`

Major Project Report

on

"STOCK AND SHOP: A FULL-STACK E-COMMERCE SYSTEM USING THE MERN STACK"

Submitted in partial fulfilment for the requirements of the degree of

BACHELOR OF COMPUTER APPLICATIONS

CENTER FOR DISTANCE & ONLINE EDUCATION
CHANDIGARH UNIVERSITY, MOHALI
PUNJAB

SUBMITTED TO :

Programme Coordinator – BCA

Center for Distance s Online Education

Chandigarh University, Mohali

Punjab

SUBMITTED BY :

Saanvi Suresh Rao

O23BCA160051

BCA – Batch – Jan 2023

`

I hereby declare that this Project Report titled **"Stock And Shop: A Full-Stack E-Commerce system using the Mern stack"** is an authentic record of my own work carried out during the academic year **2025-2026**, in partial fulfillment of the requirements for the award of the degree of **Bachelor Of Computer Applications** at **Chandigarh University**.

I further declare that the work presented in this report is original and has not been previously submitted in part or full for any other degree, diploma, or similar award at any other university or institution. All sources of information and help have been duly acknowledged.

Saanvi Suresh Rao

O23BCA160051

Bachelor Of Computer Application

Center for Distance & Online Education

Date: November 29, 2025

Place: Vapi, Gujarat

`

ACKNOWLEDGEMENT

I want to sincerely thank everyone who helped me finish my project, **"Stock And Shop: A Full-Stack E-Commerce system using the Mern stack"** and for their support.

I am extremely grateful to Chandigarh University for giving me the chance to complete this project as a requirement for my Bachelor Of Computer Application degree. I am grateful to Chandigarh University's academic staff and administration for creating and implementing a thorough program that gave me the skills and information I needed to finish this work.

I also want to thank the technical support staff for helping to make sure that the learning process runs smoothly.

Lastly, I would like to express my sincere gratitude to my family for their unwavering understanding, encouragement, and support during the difficult stages of this project.

Saanvi Suresh Rao

O23BCA160051

Bachelor Of Computer Applications – CU ONLINE

`

ABSTRACT

This major project introduces **"Stock & Shop,"** an e-commerce platform developed to address the process inefficiencies inherent in traditional grocery shopping while providing a secure and scalable online marketplace for groceries, vegetables, and fruits.

The system focuses on establishing a full-stack, dynamic environment for retail, ensuring improved inventory visibility and streamlined transaction processing. The platform is built on the **MERN stack**, utilizing **MongoDB** for flexible product and order data storage, **Express.js** and **Node.js** for a high-performance backend API, and **React** for a dynamic, user-friendly shopping experience.

Key functionalities include secure user authentication, real-time cart management, and a robust **online payment gateway integrated via Stripe**. A critical design element is the focus on **transaction reliability**, ensuring orders are created, and inventory is updated only upon verified, final confirmation of payment receipt.

"Stock & Shop" provides a comprehensive, modern e-commerce solution, significantly enhancing operational proficiency and demonstrating mastery of full-stack development and third-party financial service integration.

`

TABLE OF CONTENTS

- Title Page
- Declaration / Bonafide Certificate
- Acknowledgement
- Abstract
- Table of Contents

Chapters

- Chapter 1: Background
    - Project Profile and Objectives
    - Scope of the Project
    - Existing System Limitations and Proposed Solution
    - Technology Used
- Chapter 2: System Analysis
    - Study of Requirements
    - Feasibility Study
    - Hardware and Software Requirements
    - System Planning
- Chapter 3: System Design
    - System Architecture
    - ER-Diagram
    - Data Flow Diagrams
    - Database Design and Schema
    - Input And Output Screen Design

`

- **Chapter 4: Implementation**
    - **Process Logic of Authentication Module**
    - **Process Logic of Product and Inventory Module**
    - **Process Logic of Payment Gateway**
        - **Secure Webhook Handler Implementation**
        - **Order Fulfillment and Cart Clearance Logic**
    - **Code implementation Details**
- **Chapter 5: Testing And Maintenance**
    - **Testing Methodology Adopted**
    - **Test Cases and Expected Results**
    - **System Evaluation and Future Scope**
    - **Cost and Benefit Analysis**
    - **User/Operational Manual**
        - ***CONLUSION***
        - ***BIBLIOGRAPHY / REFERENCES***
        - ***APPENDICES***

`

CHAPTER – 1

BACKGROUND

PROJECT PROFILE AND OBJECTIVE:

The project, titled **"E-COMMERCE SYSTEM USING MERN STACK"** (named **Stock & Shop** in the application), is a robust, full-stack application developed to serve as a modern online platform for purchasing groceries, fresh vegetables, and fruits. The system is architected around the highly scalable **MERN (MongoDB, Express.js, React, Node.js) technology stack**.

Project Profile

DETAIL	DESCRIPTION
Project Name	Stock & Shop
Domain	E-commerce for Groceries and Inventory Management
Architecture	Full-Stack, Three-Tier (MERN)
Key Technology	MongoDB, React, Node.js/Express, Stripe Payment Gateway
Target Users	Retail Customers and System Administrators

Project Objectives

The core objectives of developing the Stock & Shop E-commerce System are as follows:

1. **To Implement a Scalable MERN Architecture:** To design and develop a high-performance application utilizing a component-based React frontend and a modular Node.js/Express backend, ensuring the system can handle a growing user base and product catalog.

`

2. **To Ensure Secure and Controlled Access:** To establish a complete user lifecycle management system, including secure authentication, Role-Based Access Control (RBAC) to differentiate Customer and Administrator privileges, and a secure Forgot Password feature utilizing OTP (One-Time Password) email verification.

3. **To Provide Real-Time Inventory Control:** To implement an efficient Admin Dashboard allowing for CRUD (Create, Read, Update, Delete) operations on categories, sub-categories, and products, ensuring accurate and real-time tracking of stock quantity.

4. **To Integrate a Reliable Payment Solution:** To integrate the Stripe Payment Gateway for secure, third-party processing of all online credit/debit card transactions, maintaining strict security and PCI compliance.

5. **To Guarantee Transaction Reliability:** To design and implement the complex secure order fulfillment logic on the backend to ensure that permanent order records are created and customer carts are cleared only after a verified, successful payment signal is received from Stripe.

`

Scope of the Project:

The project scope defines the functional boundaries of the "Stock & Shop" E-commerce System, covering the complete lifecycle from inventory management to order fulfillment using the MERN stack and Stripe.

1. **Authentication and Security:** Includes secure user **Login, Register**, and **Forgot Password** with **OTP** verification. The system implements **Role-Based Access Control (RBAC)** to distinguish between Customer and Administrator privileges.
2. **Inventory Management:** The Administrator module is fully in scope, enabling **CRUD (Create, Read, Update, Delete)** operations on all **Groceries, Categories, and Sub-Categories**. The system tracks and updates **real-time stock quantity**.
3. **Customer Shopping Experience:** The scope covers the user interface for product **browsing, searching, cart management**, and the ability to save multiple **shipping addresses**.
4. **Secure Payment and Fulfillment:** The system integrates the **Stripe API** to initiate online payment sessions. Crucially, the backend implements **secure server-to-server confirmation logic** to ensure orders are created and the customer's cart is cleared **only** upon verified payment success.

`

Existing System Limitations and Proposed Solution:

This section analyzes the shortcomings of traditional or older retail systems, which the Stock & Shop E-commerce platform is designed to address through its modern technology stack.

A. Limitations of the Existing/Manual System

LIMITATION	DESCRIPTION
Inaccurate Inventory	Manual tracking of grocery and produce stock is prone to human error, leading to inaccurate availability displayed to the customer.
Limited Accessibility	Physical stores or basic websites are not accessible 24/7, restricting sales opportunities and customer convenience.
Payment Insecurity	Systems relying on cash on delivery or non-integrated payment methods lack security and real-time transaction verification, posing fraud risks.
Lack of Data Centralization	Customer data, orders, and sales history are often scattered, making trend analysis and efficient resource planning difficult.

B. The Proposed Solution

The "Stock & Shop" E-commerce System overcomes these limitations by implementing a modern, integrated, and scalable solution:

1. Real-Time Data Management: The MERN Stack (MongoDB) provides a centralized database for all inventory, customer, and order data, enabling the Admin to view and update stock quantities in real-time.

`

2. 24/7 Accessibility: The React frontend provides a dynamic, accessible online platform that operates continuously, significantly expanding the market reach.
3. Secure Financial Transactions: Integration with the Stripe Payment Gateway offloads sensitive card processing, ensuring PCI compliance and providing instant, verified payment confirmation via a secure server-to-server connection.
4. Automated Fulfillment: The backend logic automatically manages order creation, inventory updates, and cart clearance upon verified payment, maximizing efficiency and minimizing human error.

`

Technology Used:

The "Stock & Shop" E-commerce System is a Full-Stack application built on the MERN technology stack, supplemented by the Stripe Payment Gateway. This modern architecture was chosen for its flexibility, scalability, and ability to handle high-performance web applications.

COMPONENT	TECHNOLOGY	ROLE IN THE SYSTEM
M	MongoDB	The **NoSQL Database** used for flexible and scalable data storage. It manages all application data, including user profiles, product catalogs, and order records.
E	Express.js	The **Web Application Framework** for Node.js. It handles routing, middleware, and API endpoints, forming the robust backbone of the server.
R	React	The **JavaScript Library** used to build the dynamic, component-based user interface (UI) for both the Customer shopping experience and the Administrator dashboard.
N	Node.js	The **JavaScript Runtime Environment** that executes the Express.js server logic. It enables fast, non-blocking I/O operations necessary for a high-traffic E-commerce API.
Payment	Stripe API	The integrated third-party service that securely manages all credit/debit card transactions and provides the necessary **webhook** signals for reliable order fulfillment.

`

Development Tools and Libraries

The project leverages several critical tools beyond the core stack:

- **JWT (JSON Web Tokens):** Used for secure, stateless user authentication and session management.
- **Bcrypt:** Utilized for secure hashing of user passwords before storage in MongoDB.
- **Stripe CLI:** Essential for local development, enabling the simulation of live webhook events on the local machine for testing the order fulfillment logic.
- **NodeMailer:** Used for sending the **OTP (One-Time Password)** via email during the Forgot Password process.

`

CHAPTER – 2

SYSTEM ANALYSIS

Study of Requirements

The Study of Requirements defines the functional, non-functional, and operational needs of the "Stock & Shop" E-commerce System. This ensures that the system aligns with user expectations, performance goals, security needs, and platform constraints. The requirements were analyzed from the perspectives of customers, administrators, and system behavior, forming the foundation for all design and implementation activities.

A. Functional Requirements

Functional requirements define *what the system must do*. These are directly derived from the objectives of the Stock & Shop platform.

1. User Authentication and Authorization

- Secure User Registration and Login functionality.
- Role-Based Access Control (RBAC) distinguishing **Customer** and **Admin** users.
- Forgot Password workflow using OTP email verification.
- Authenticated session handling using JWT tokens.

2. Product and Inventory Management (Admin Module)

- CRUD operations for Categories, Sub-categories, and Products.

`

- Real-time updating of product stock quantities.
- Uploading product images, setting prices, descriptions, and category associations.
- Dashboard displaying inventory status.

3. Customer Shopping Experience (User Module)

- Product browsing, searching, and filtering.
- Add-to-cart, update quantity, and remove-from-cart functionalities.
- Management of multiple shipping addresses.
- Viewing order history and order details.

4. Order Management and Payment Integration

- Creating a temporary order session before payment.
- Integration with Stripe for secure payment processing.
- Receiving secure **webhook events** from Stripe.
- Creating final order records only after verified payment success.
- Auto-clearing the user's cart upon successful payment.

B. Non-Functional Requirements

Non-functional requirements ensure performance, availability, reliability, and ease of use.

1. Performance Requirements

- The system must handle multiple concurrent user sessions.
- API responses must be optimized using asynchronous, non-blocking Node.js operations.
- Database queries must be index-optimized for faster reads.

`

2. Security Requirements

- Passwords must be encrypted using Bcrypt.
- JWT-based session authentication ensures secure access.
- Stripe ensures PCI-compliant payment processing.
- Sensitive logic (order creation, inventory update) executed **only** on secure server-side webhook confirmation.

3. Reliability Requirements

- Webhook handler must ensure idempotency to prevent duplicate orders.
- MongoDB collections must guarantee atomic updates on inventory count.
- The application must ensure uptime with minimal failure points.

4. Usability Requirements

- Intuitive navigation for both customer and admin dashboards.
- Mobile-responsive UI using React components.
- Clear visual feedback for cart operations and checkout flow.

5. Scalability Requirements

- MERN architecture allows horizontal scaling of backend and database.
- React frontend supports modular expansion for future features.

C. User Requirements

User requirements define what **end users expect** from the system.

Customer Requirements

- Ability to create an account and login securely.

`

- Browse products easily with clear categories.
- Add items to cart and complete payments online.
- Track order status and view past orders.

Administrator Requirements

- Ability to manage inventory in real time.
- Add, update, or delete categories and products.
- Monitor product availability and update stock accordingly.
- View and manage customer orders.

D. System Requirements

These specify the expected system behavior and constraints.

- The application must be accessible 24/7 through the internet.
- Backend must expose RESTful API endpoints for all operations.
- MongoDB must handle structured and unstructured product data efficiently.
- The system must integrate third-party services (Stripe, NodeMailer) without compromising performance.

`

Feasibility Study

The feasibility study evaluates whether the "Stock & Shop" E-commerce System can be successfully developed and operated using the MERN stack and Stripe payment integration. It covers technical, operational, and economic aspects to ensure that the proposed solution is practical, achievable, and beneficial.

A. Technical Feasibility

The technologies selected for the project–MongoDB, Express.js, React, Node.js, and Stripe–are modern, well-documented, and widely used for scalable web applications. The MERN stack provides high performance, non-blocking operations, and flexible data handling, making it technically feasible to implement real-time inventory updates, authentication workflows, and payment processing. All required tools and development environments are freely accessible and supported by extensive online resources, ensuring smooth project development.

B. Operational Feasibility

The system addresses the limitations of traditional grocery shopping by providing 24/7 accessibility, real-time stock visibility, and secure online payments. The customer interface is designed for easy browsing, cart management, and checkout, while the admin dashboard simplifies product

`

and inventory management. The solution is practical for real-world use, supports seamless online ordering, and can be operated by both customers and administrators without specialized training.

C. Economic Feasibility

The project is economically viable as it uses open-source technologies that eliminate licensing costs. Tools such as MongoDB, Node.js, React, and Express are completely free to use. Stripe follows a pay-as-you-go model, which means there are no upfront charges for integration or testing. The overall development and maintenance cost remains low, making the system cost-effective for deployment, scaling, and long-term use.

`

Hardware And Software Requirements

This section outlines the minimum hardware and software resources required to develop and operate the "Stock & Shop" E-commerce System. Since the application is MERN-based and uses Stripe for payment integration, the requirements are lightweight and suitable for modern development environments.

A. Hardware Requirements

For Development

- **Processor:** Intel i5 or equivalent
- **RAM:** 8 GB (minimum)
- **Storage:** 20 GB free space
- **System:** Windows / macOS / Linux

For Deployment (Server)

- **Processor:** Dual-core CPU
- **RAM:** 4–8 GB
- **Storage:** SSD preferred
- **Network:** Stable broadband connection

`

B. Software Requirements

Backend

- Node.js
- Express.js
- MongoDB
- Stripe API
- Postman (for API testing)

Frontend

- React
- HTML, CSS, JavaScript
- Axios for API requests

Development Tools

- VS Code
- Git and GitHub
- MongoDB Compass
- Stripe CLI
- npm packages (JWT, Bcrypt, Nodemailer, etc.)

`

System Planning

System planning involves defining the structure and workflow of the Stock & Shop platform to ensure smooth development and efficient operation.

1. **Modular Architecture:**

   The system is divided into modules such as Authentication, Product Management, Cart, Orders, and Payment Integration for easier development and maintenance.

2. **Three-Tier MERN Structure:**

   - Frontend (React): Handles UI and customer interactions.
   - Backend (Node/Express): Manages business logic and API routes.
   - Database (MongoDB): Stores users, products, orders, and inventory data.

3. **API Planning:**

   REST-based endpoints were planned for user actions, admin operations, cart updates, and order processing.

4. **Database Planning:**
   Collections were planned for Users, Products, Categories, Orders, and Cart Items.

5. **Payment Workflow Planning:**
   Stripe Checkout and Webhook handling were designed to ensure secure order creation only after valid payment confirmation.

`

CHAPTER – 3

SYSTEM DESIGN

System Design defines the structural, architectural, and data-oriented blueprint of the "Stock & Shop" E-commerce System. The purpose of this chapter is to illustrate how different components of the application interact, how data flows between modules, and how the system ensures secure and efficient operations.

System Requirements

The "Stock & Shop" application follows a Three-Tier MERN Architecture, ensuring modularity, scalability, and efficient separation of concerns.

A. Presentation Layer (Frontend – React)

- Displays categories, products, cart, checkout, and admin dashboard.
- Manages user interactions through reusable components.
- Communicates with backend APIs using Axios.
- Implements client-side routing for smooth navigation.

B. Application Layer (Backend – Node.js + Express.js)

- Handles all business logic for authentication, products, orders, and payments.

`

- Validates user requests and enforces role-based permissions.
- Connects to Stripe for payment session creation and webhook verification.
- Generates JWTs and processes encrypted credentials.

C. Data Layer (Database – MongoDB)

- Stores Users, Products, Categories, Orders, and Cart items.
- Maintains real-time stock quantities.
- Ensures consistency during order confirmation via secure server-side logic.

Data Flow Summary:

React → Express API → MongoDB

Stripe Payment → Webhook → Express → MongoDB (Order creation)

`

ER Diagram (Entity–Relationship Overview)

The ER model of the system identifies key entities and relationships required for managing the e-commerce workflow.

Major Entities

1. **User**
    - Fields: name, email, password, role, addresses
    - Relationship: User → Orders (1-to-many)
2. **Product**
    - Fields: name, categoryId, price, stock, image
    - Relationship: Category → Products (1-to-many)
3. **Category / Sub-category**
    - Grouping for products
    - Helps streamline admin management and product filtering
4. **Order**
    - Fields: userId, items, totalAmount, paymentStatus, timestamp
    - Relationship: Order → User (many-to-1)
5. **Cart**
    - Temporary storage of selected products before checkout
    - Related to User (1-to-1)

`

High-Level Relationships

- User places Orders
- User has Cart
- Category contains Products
- Order contains multiple Products

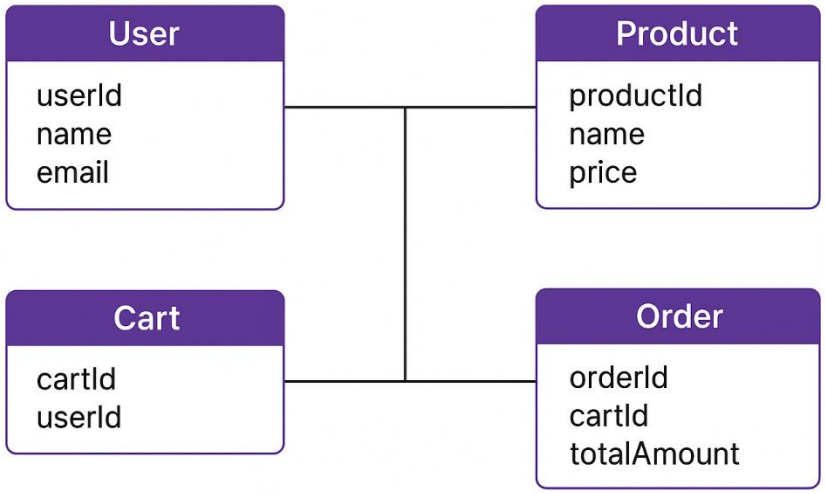


Figure (a) : Entity Relationship Diagram of Stock and Shop

`

Data Flow Diagrams (DFD)

DFDs illustrate how data moves within the system.

A. Level o DFD – Context Diagram

Shows the system as a single process interacting with:

- Customer
- Administrator
- Stripe Payment Gateway

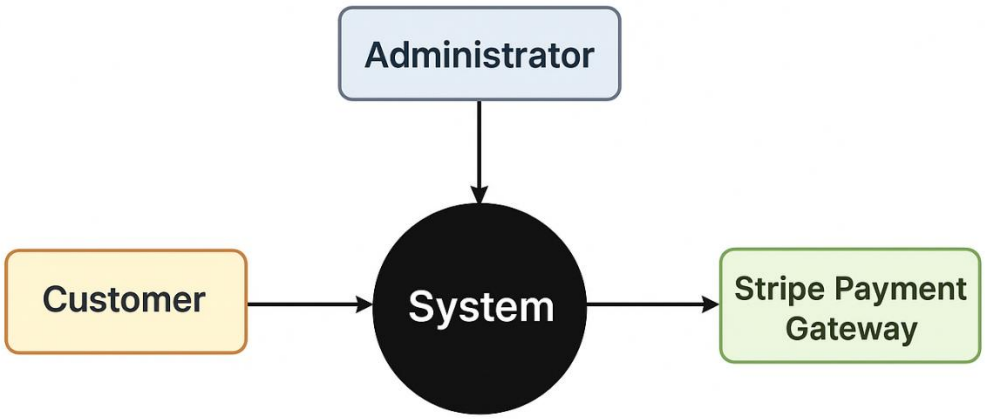


Figure (b) : Level o DFD

`

B. Level 1 DFD – Expanded Processes

1. User Process:

   Login → Browse Products → Add to Cart → Checkout

2. Admin Process:

   Manage Categories → Manage Products → Update Stock

3. Payment Process:

   Initiate Payment → Stripe Checkout → Webhook Response → Order Creation

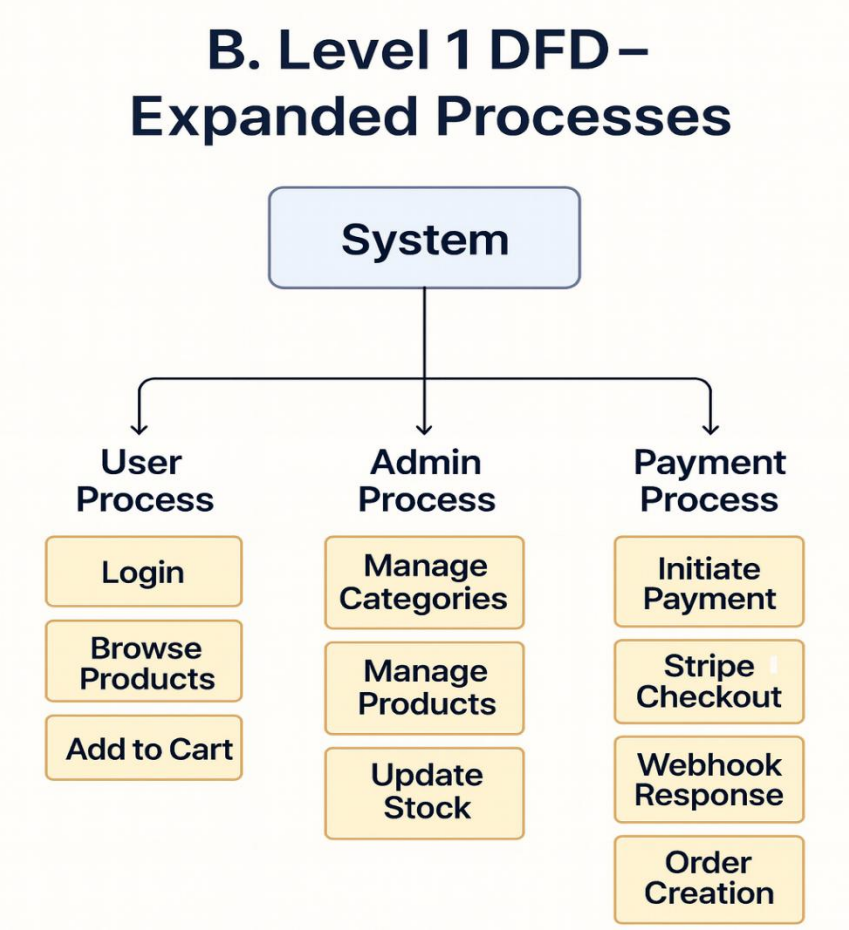


Figure (c) : Level 1 DFD

`

C. Level 2 DFD – Deep Payment Flow

- Backend creates payment session
- Stripe processes card
- Stripe webhook confirms payment
- Backend updates inventory and creates order
- Cart items are cleared

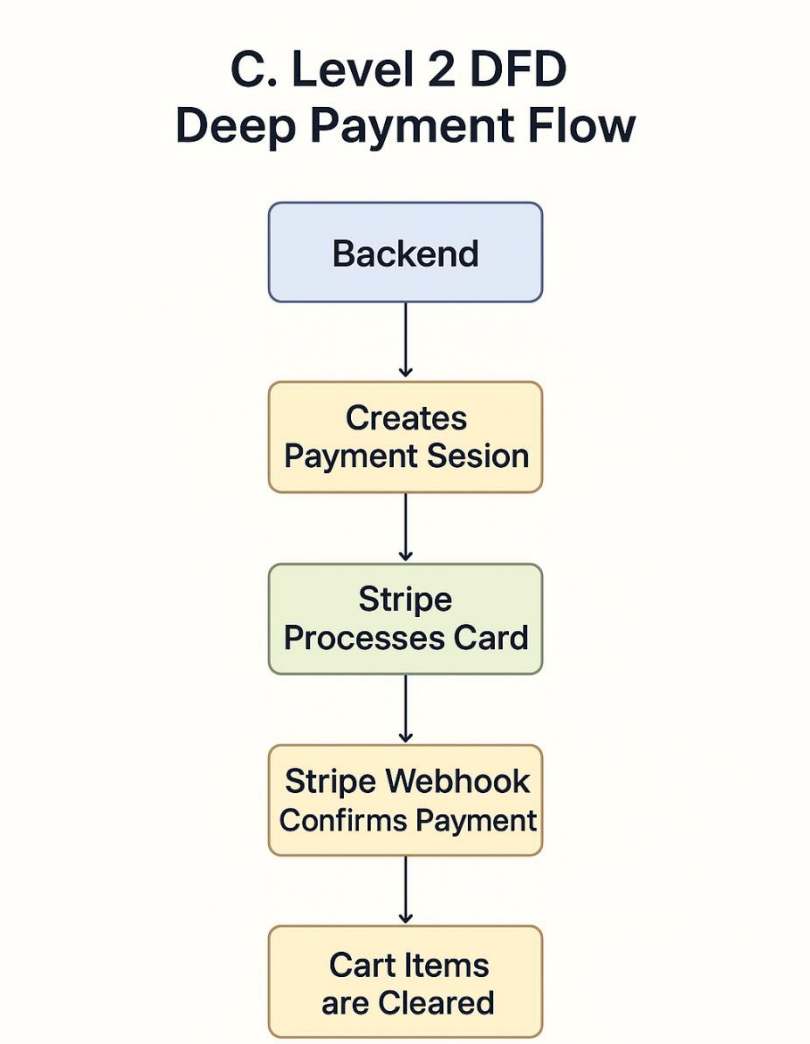


Figure (d) : Level 2 DFD

`

Database design and schema

The database uses a collection-based schema, ideal for flexible e-commerce data.

Main Collections

1. **Users**
    - name, email, role, hashed password, addresses
2. **Products**
    - title, description, category, stock, price, image URL
3. **Categories / Subcategories**
    - name, parentCategoryId (optional)
4. **Orders**
    - userId, orderItems, totalAmount, paymentIntentId, orderStatus
5. **Cart Items**
    - userId, productId, quantity

Design Principles

- Use of ObjectId references for relationships
- Inventory decremented only after verified payment
- Orders linked to both user and Stripe payment session
- Timestamped records for tracking and sorting

`

Input and Output Screen Design

The screen designs define the visual interaction points for both customers and administrators.

A. Customer Screens

- Home Page: Categories and product listings
- Product Page: Product details, stock, and Add-to-Cart option
- Cart Page: Update quantity, remove items
- Checkout Page: Payment initiation via Stripe
- Order History: View previous orders

B. Admin Screens

- Dashboard: Summary of products and stock
- Add/Edit Product: Form-based product management
- Category Management: CRUD operations
- View Orders: Complete order details and statuses

C. Output Screens

- Payment Success Screen
- Order Confirmation Screen
- Error or Validation Messages

`

CHAPTER – 4

IMPLEMENTATION

This chapter describes the practical implementation of the **"Stock & Shop" E-Commerce System**, developed using the MERN stack. The implementation covers authentication, admin inventory management, customer modules, secure payment gateway integration, and backend business logic.

Process Logic of Authentication Module

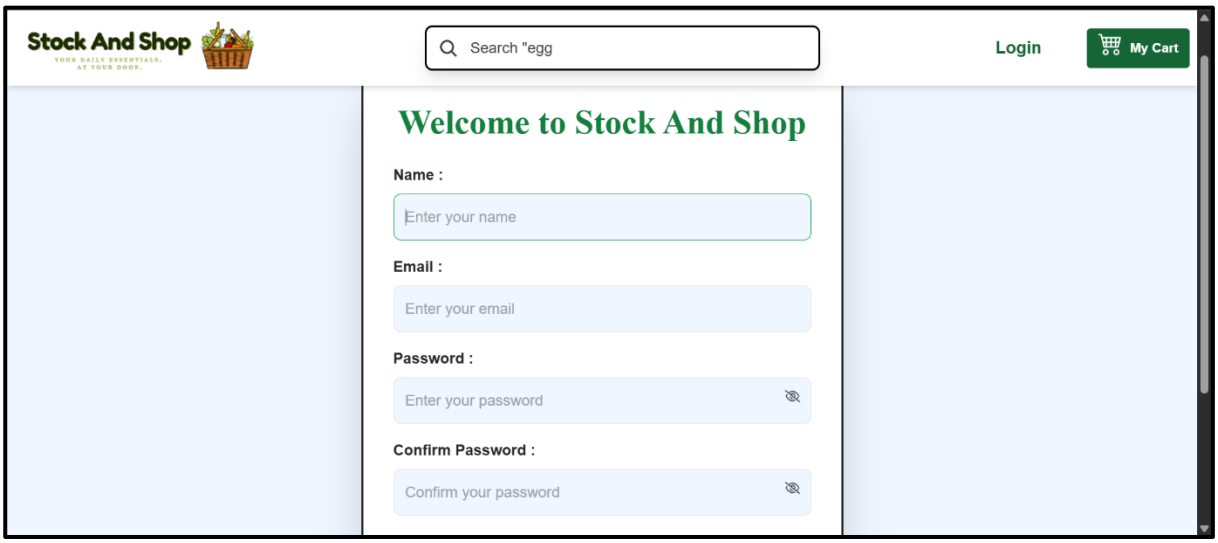
The Authentication Module handles user registration, login, and secure password reset using OTP. It also identifies whether the user is an Admin or Customer and routes them accordingly.

4.1.1 Registration Logic

Process:

- User enters name, email, and password.
- Backend validates email uniqueness.
- Password is hashed using bcrypt.
- New user is saved with default role "customer".

`



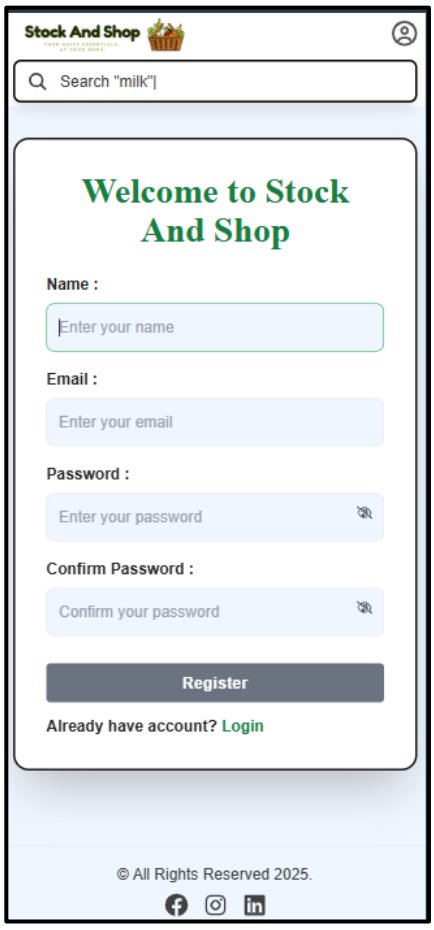


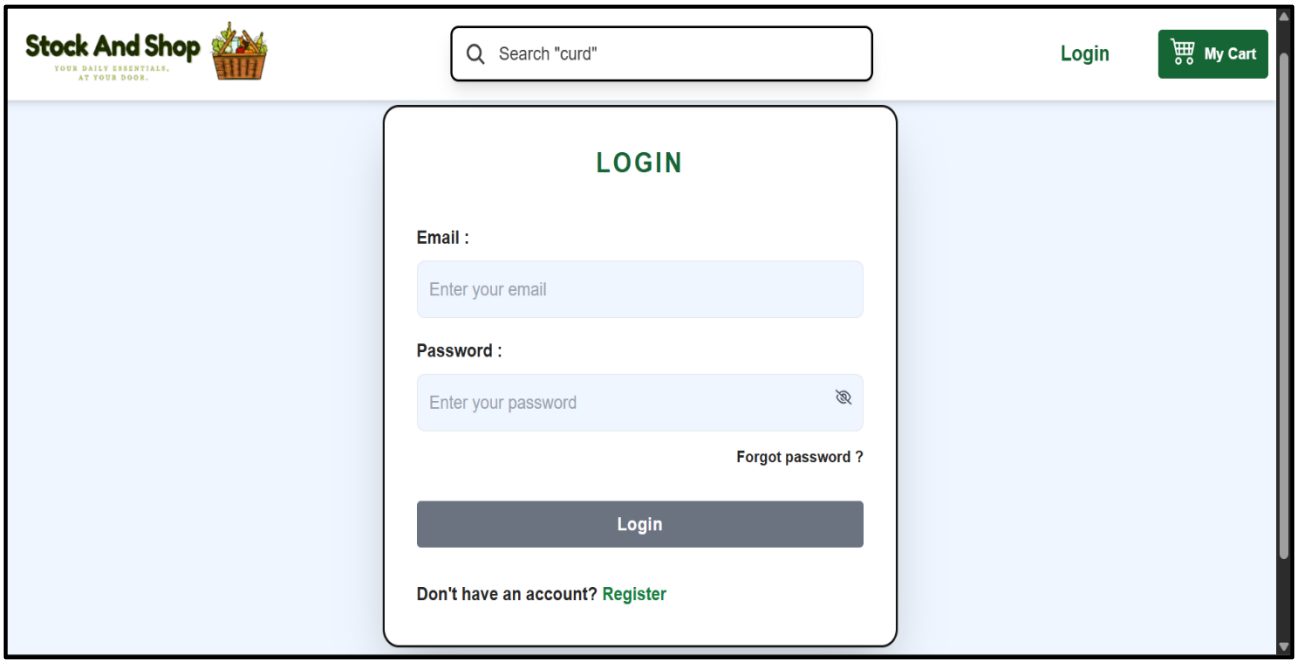
Figure 4.1: User Registration Page

`

4.1.2 Login Logic

Process:

- User enters credentials.
- Server verifies hashed password.
- JWT token is generated.
- If role = Admin → redirect to Admin Dashboard.
- If role = Customer → redirect to User Homepage



`

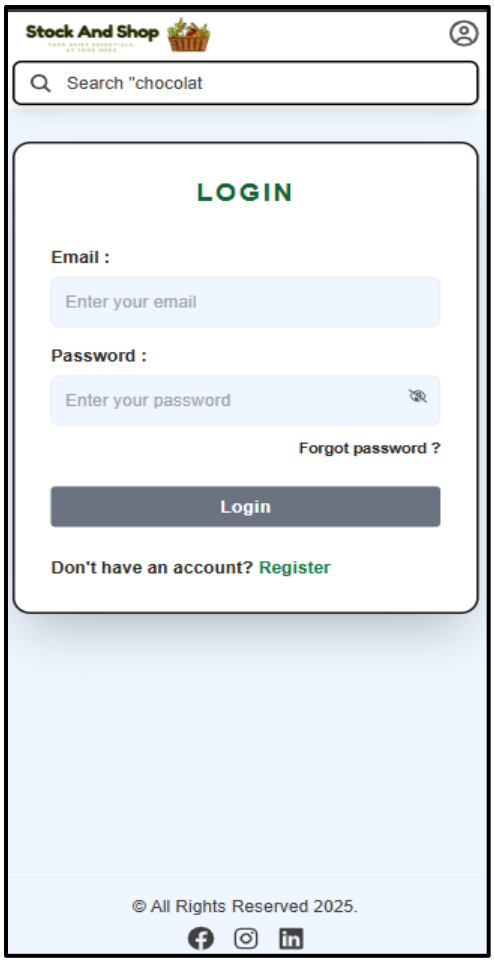


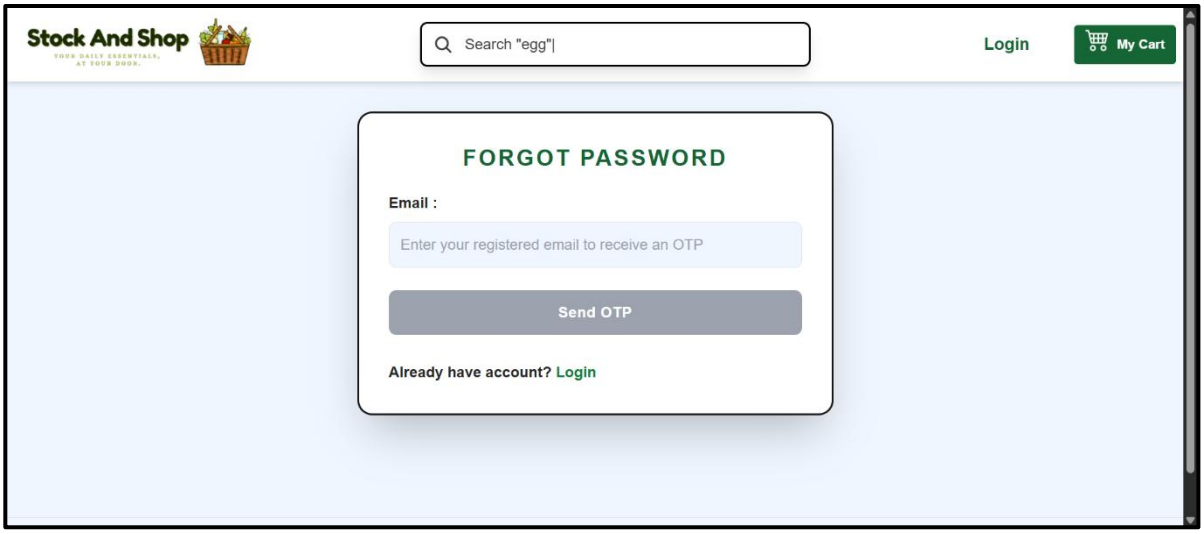
Figure 4.2: User Login Page

4.1.3 Forgot Password (OTP) Logic

Process:

- User enters registered email.
- OTP is generated and emailed using **Nodemailer**.
- User verifies OTP.
- System allows new password creation.

`



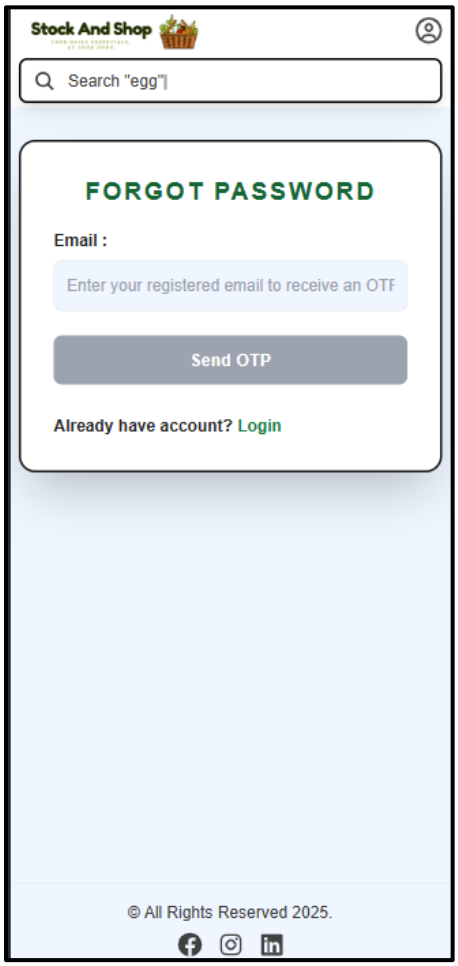


Figure 4.3: Forgot Password

`

4.1.4 Authenticated User Details Page

Both Admin and Customer have profile pages displaying personal details.

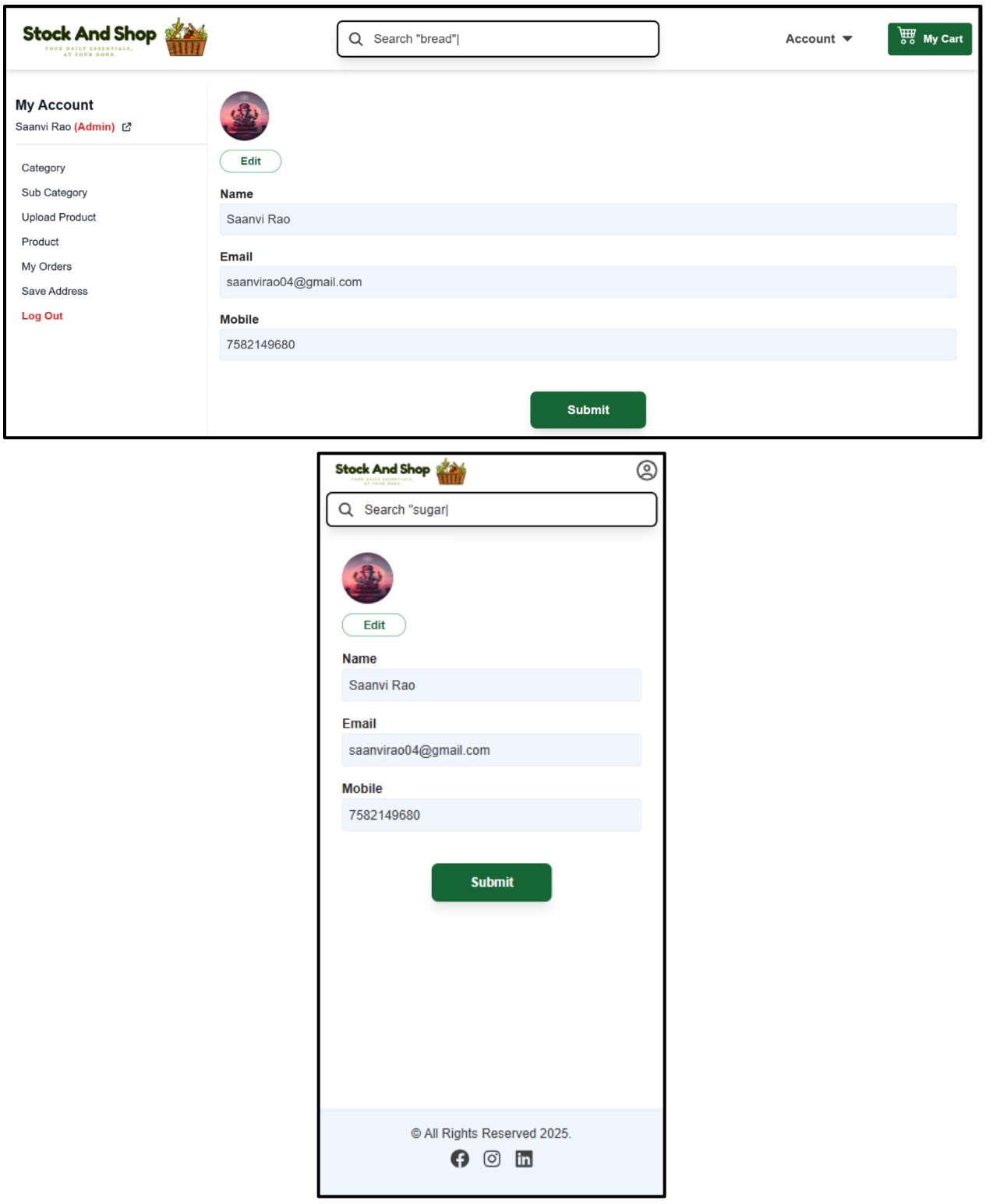
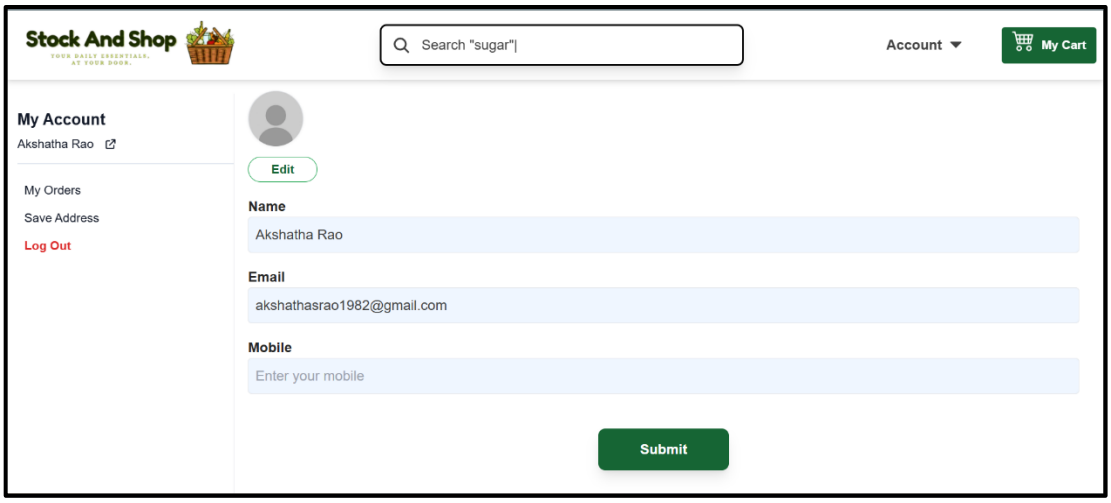


Figure 4.4: Admin Profile Page

`



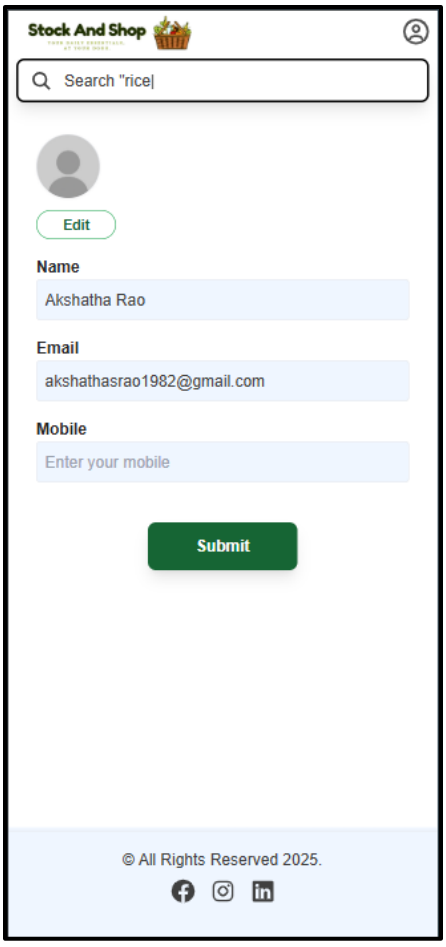


Figure 4.5: Customer Profile Page

`

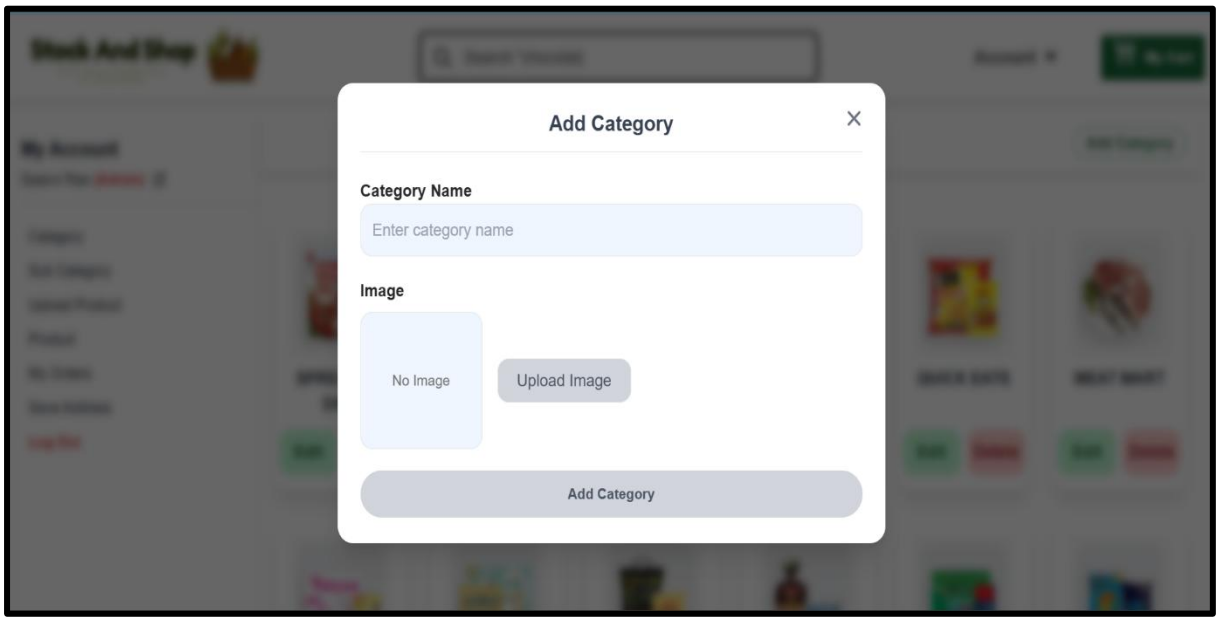
4.2 Process Logic of Product & Inventory Module (Admin)

This module is accessible only to the **Administrator** and provides complete control over categories, sub-categories, products, and inventory stock.

4.2.1 Category Management

Process:

- Admin enters category name and image.
- Category is stored in MongoDB.
- Categories are displayed dynamically during product upload.



`

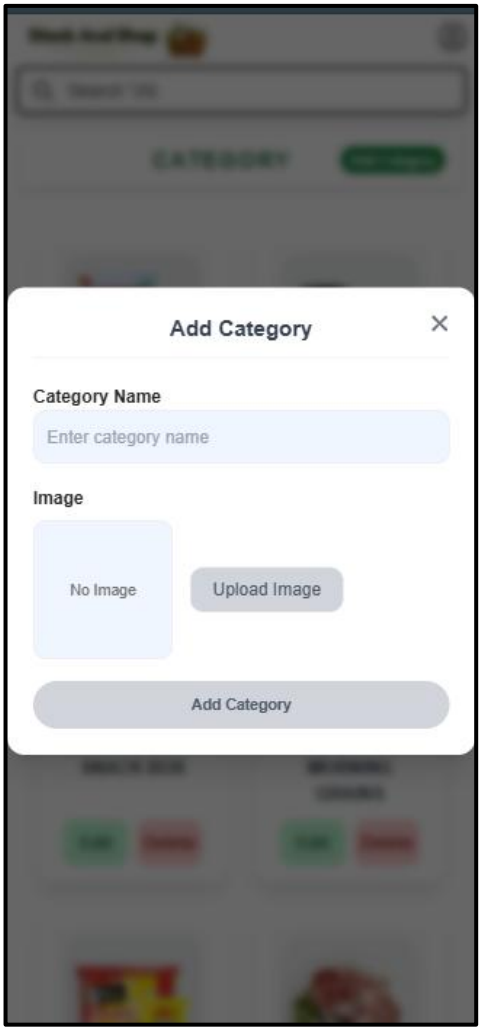
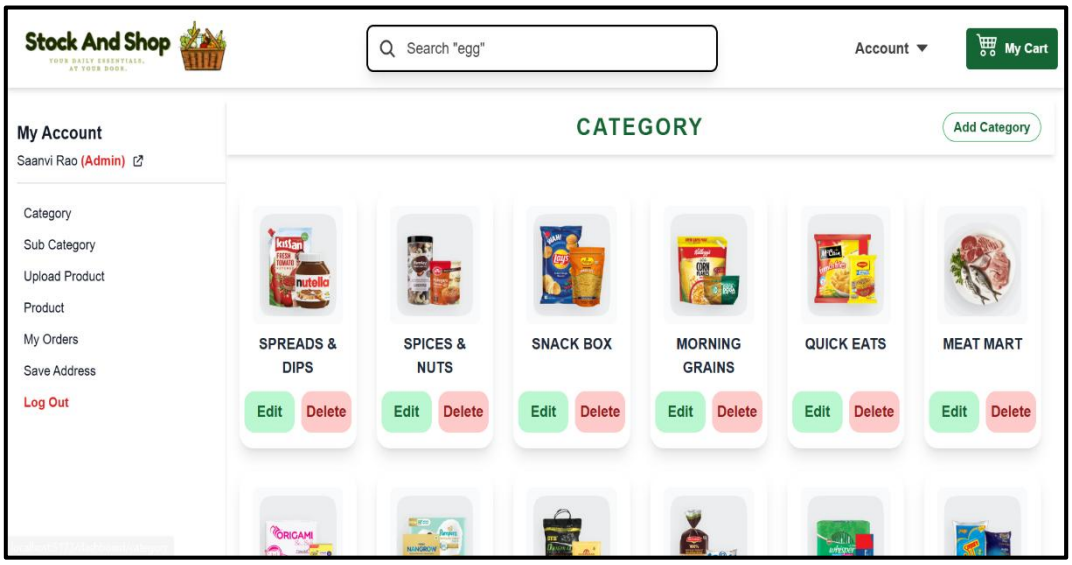


Figure 4.6: Admin – Add Category Page



`

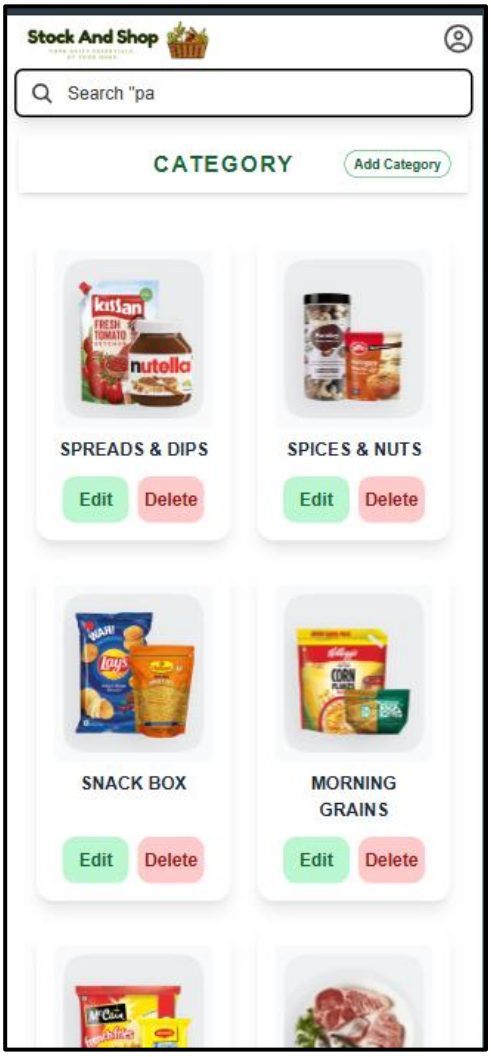


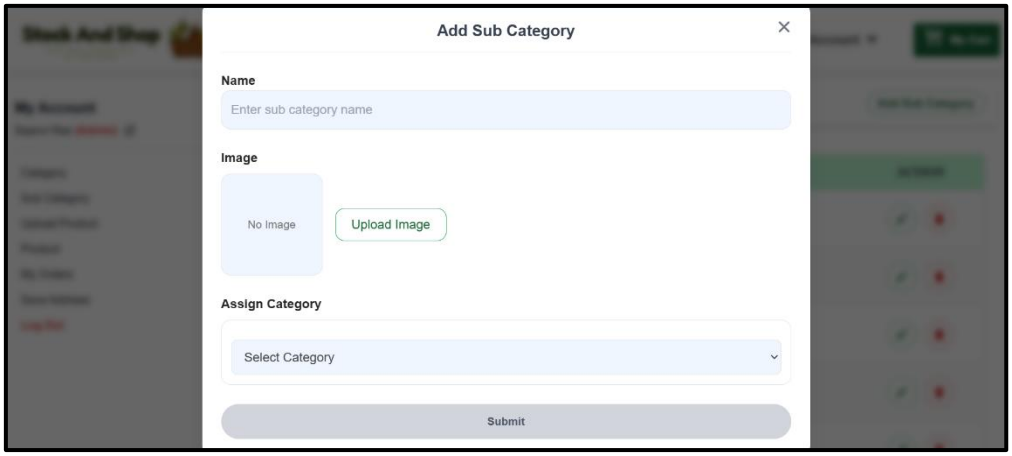
Figure 4.7: Admin – Category List Page

4.2.2 Sub-Category Management

Process:

- Admin selects a category.
- Adds sub-category name and image.
- Sub-category stored with reference to parent category.

`



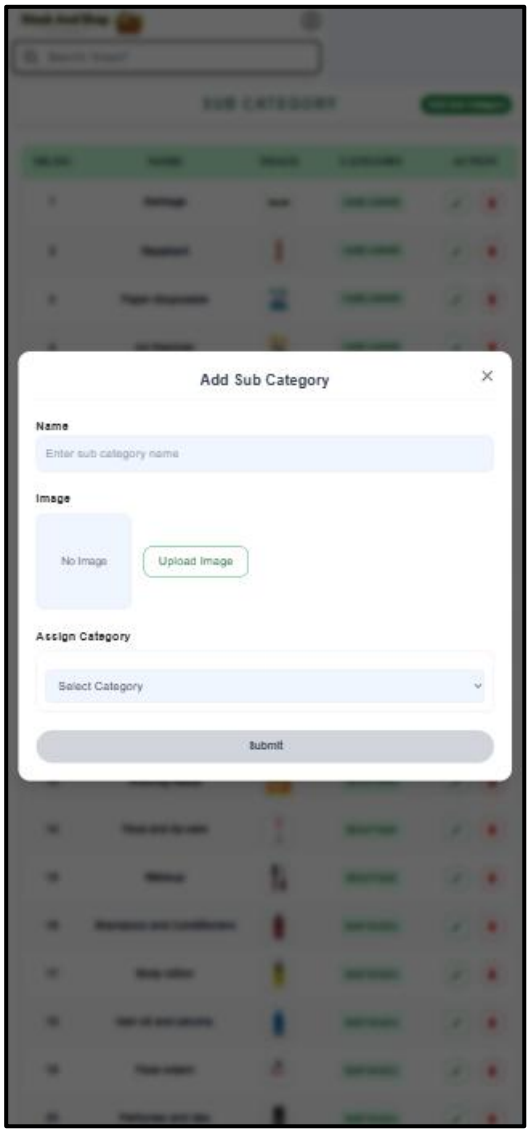
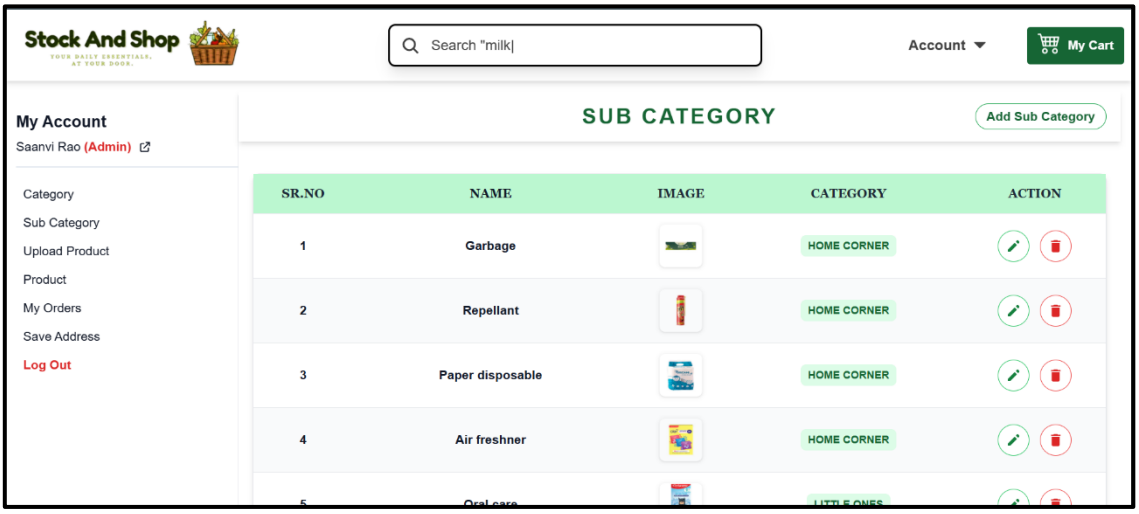


Figure 4.8: Admin – Add Sub-Category Page

`



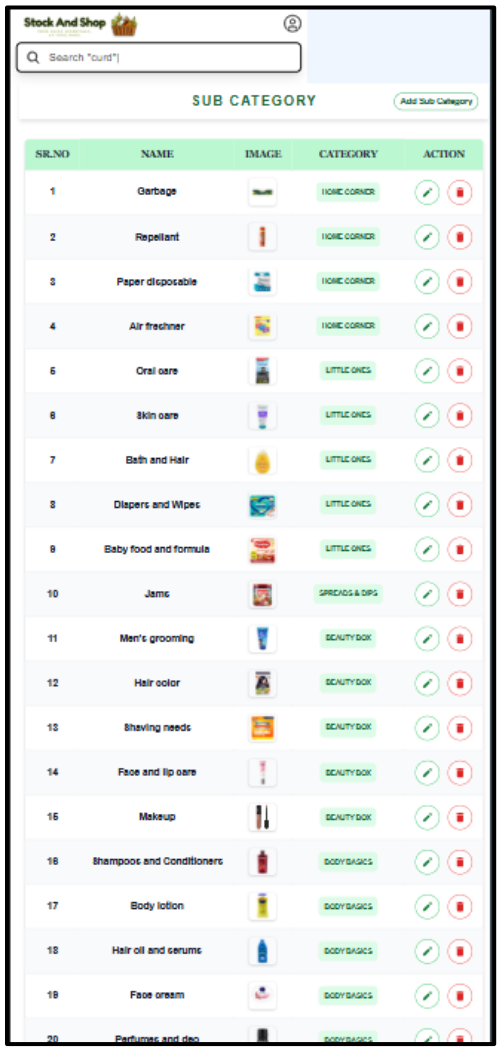


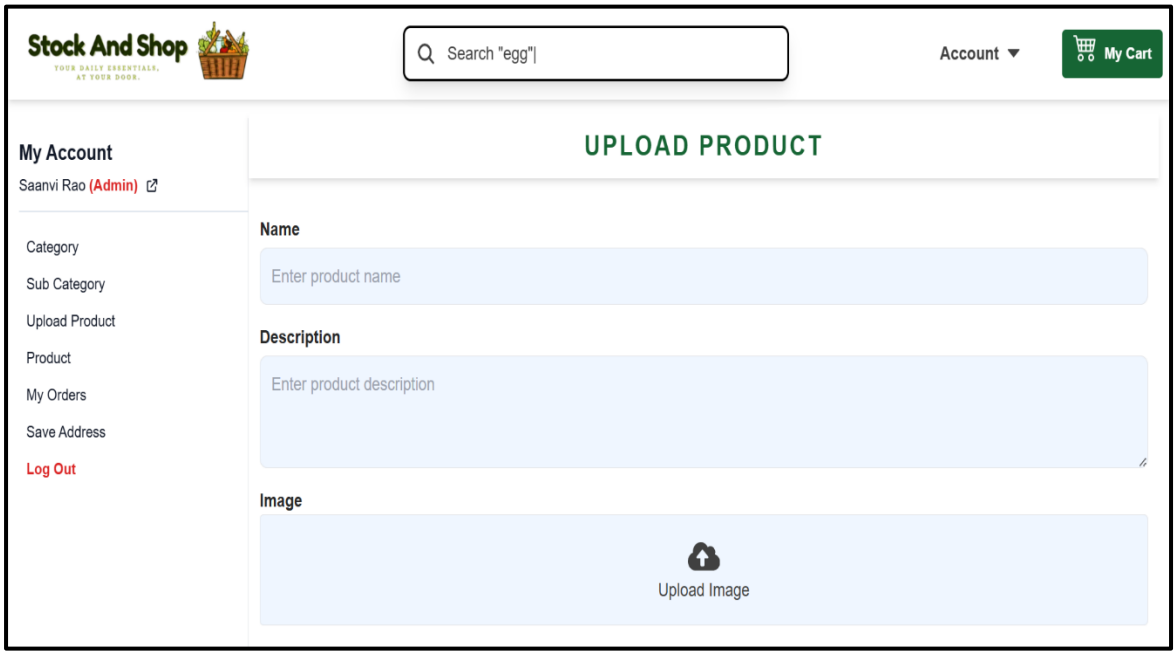
Figure 4.9: Admin – Sub-Category List Page

`

4.2.3 Product Upload & Listing

Process:

- Admin selects Category → Sub-Category.
- Enters product details.
- Uploads product image.
- Product stored with category and sub-category references.



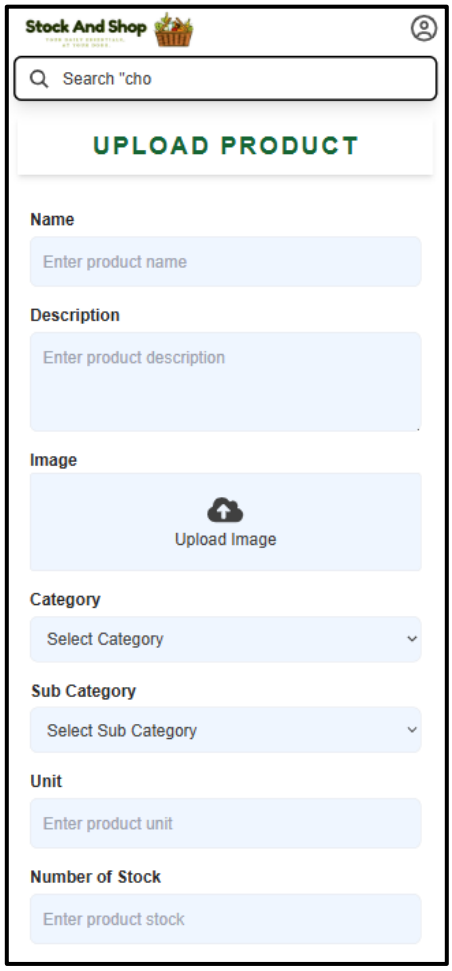
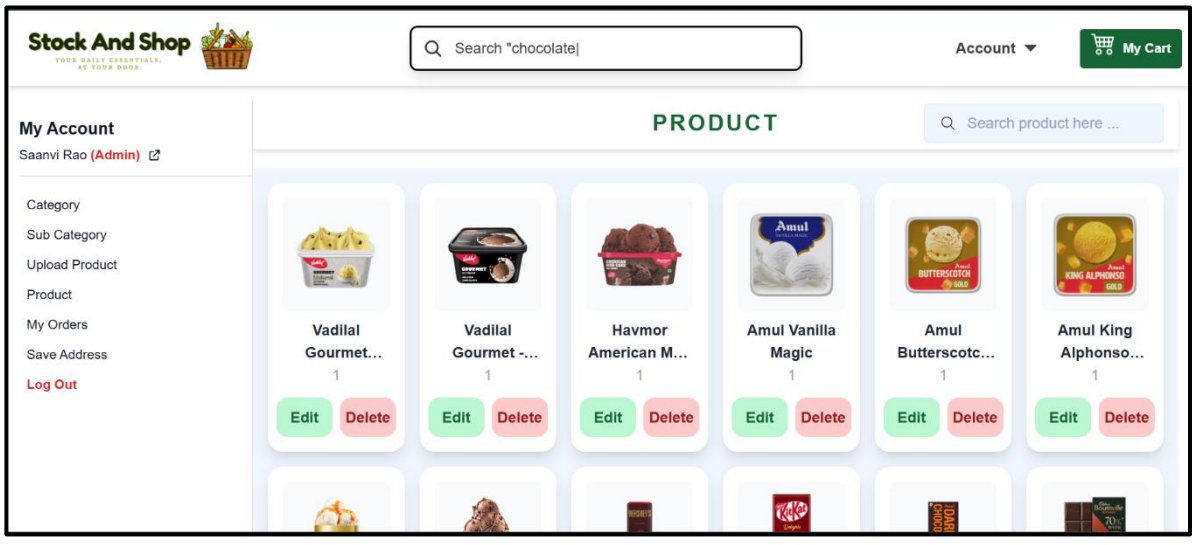


Figure 4.10: Admin – Upload Product Page



`

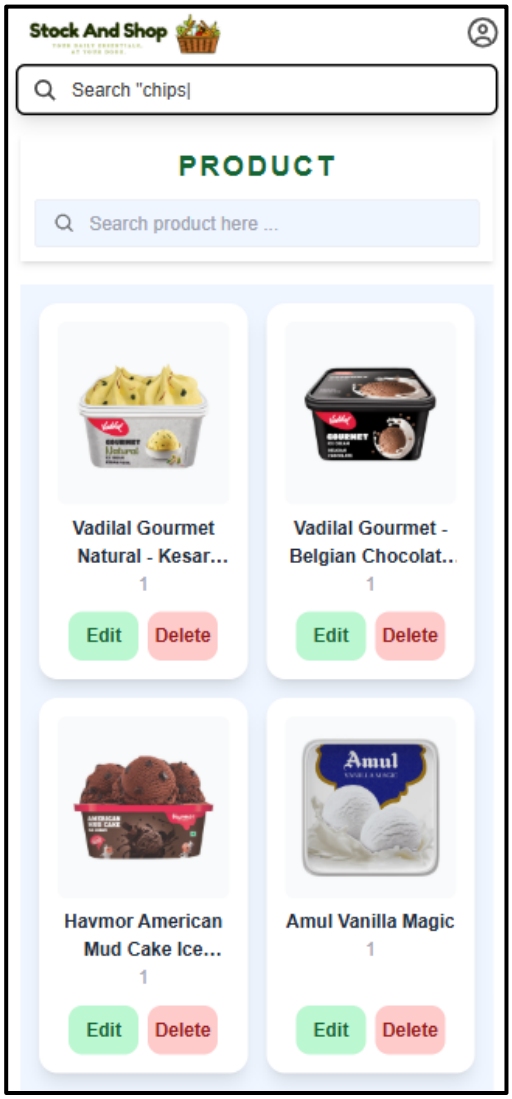
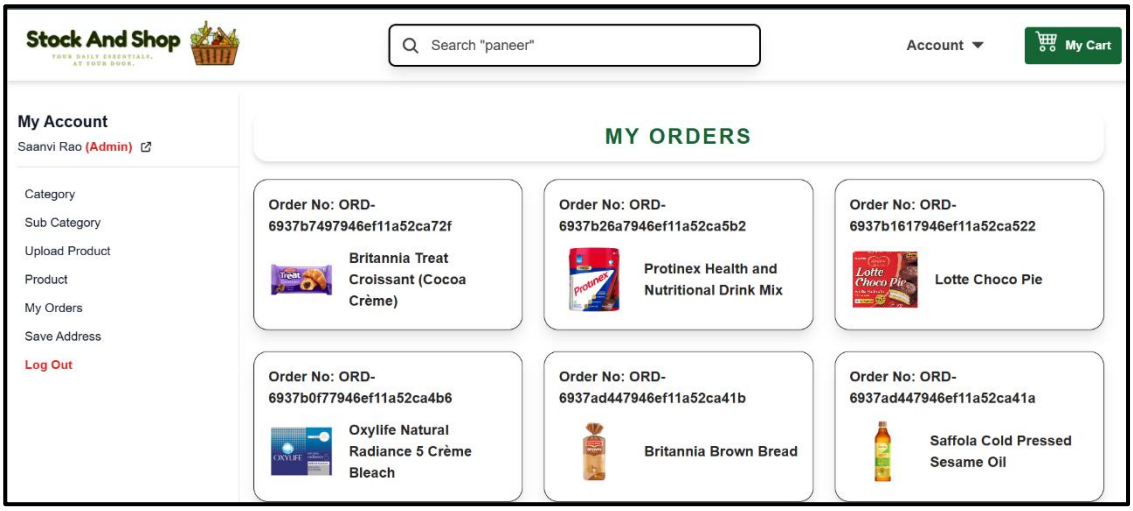


Figure 4.11: Admin – Product Page

4.2.4 Admin – Orders & Addresses

Admin can view:

- All customer orders
- Saved delivery addresses



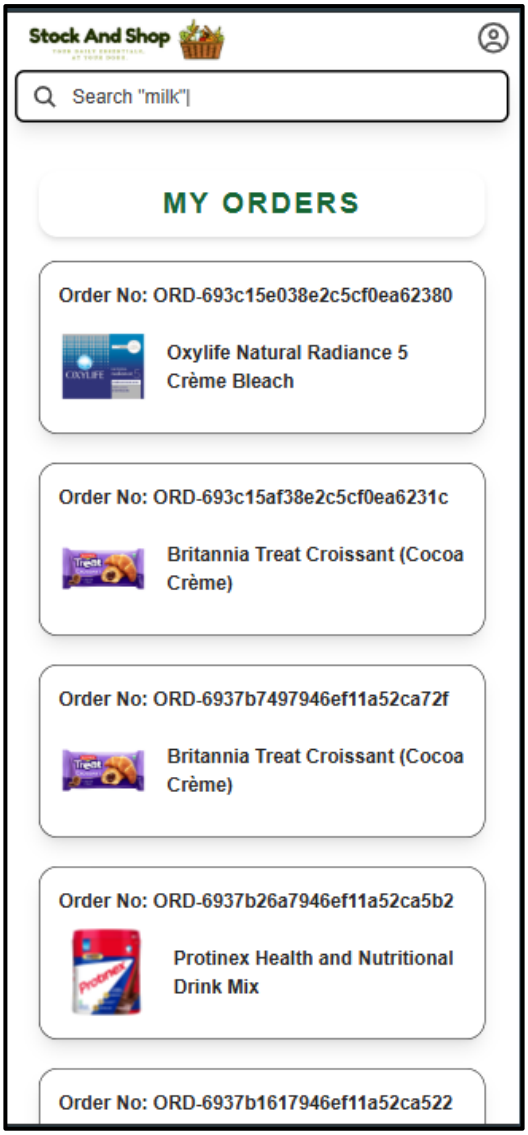
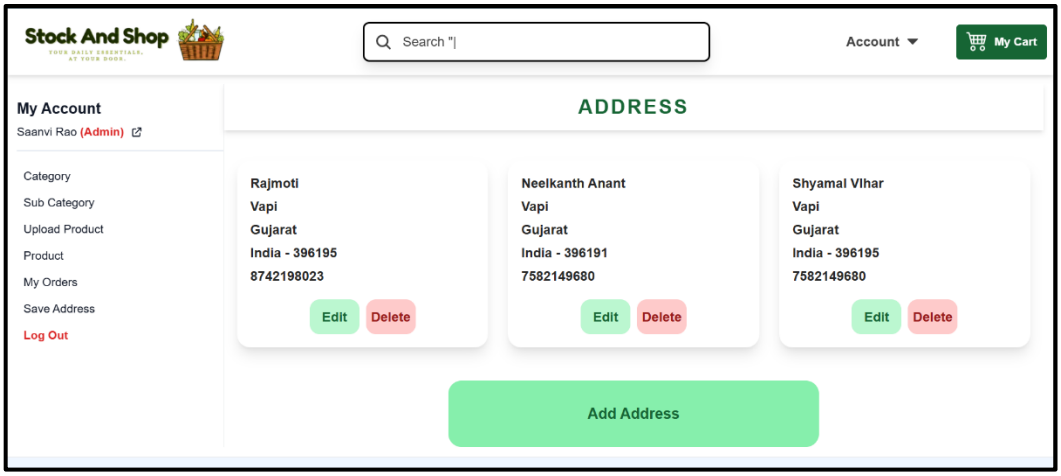


Figure 4.12: Admin – My Orders Page



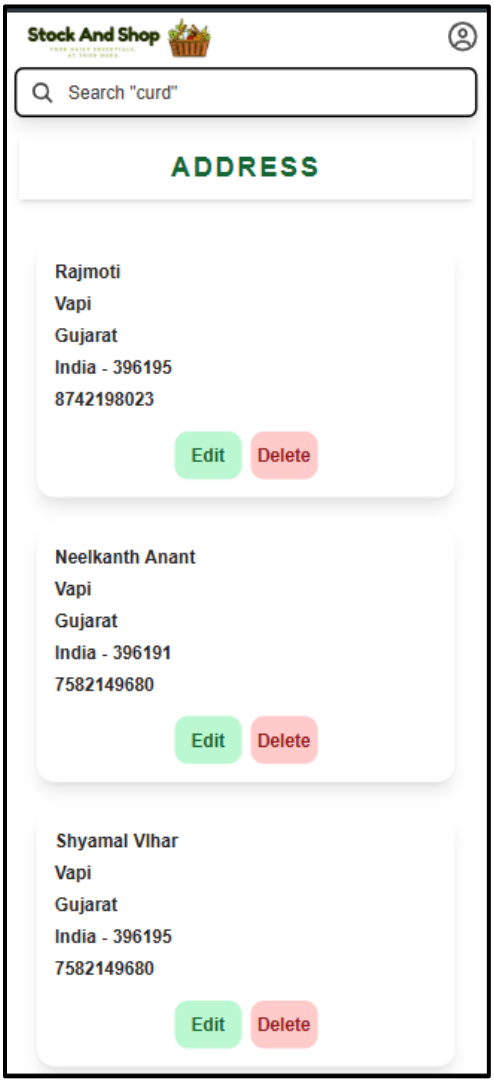


Figure 4.13: Admin – Saved Addresses Page

`

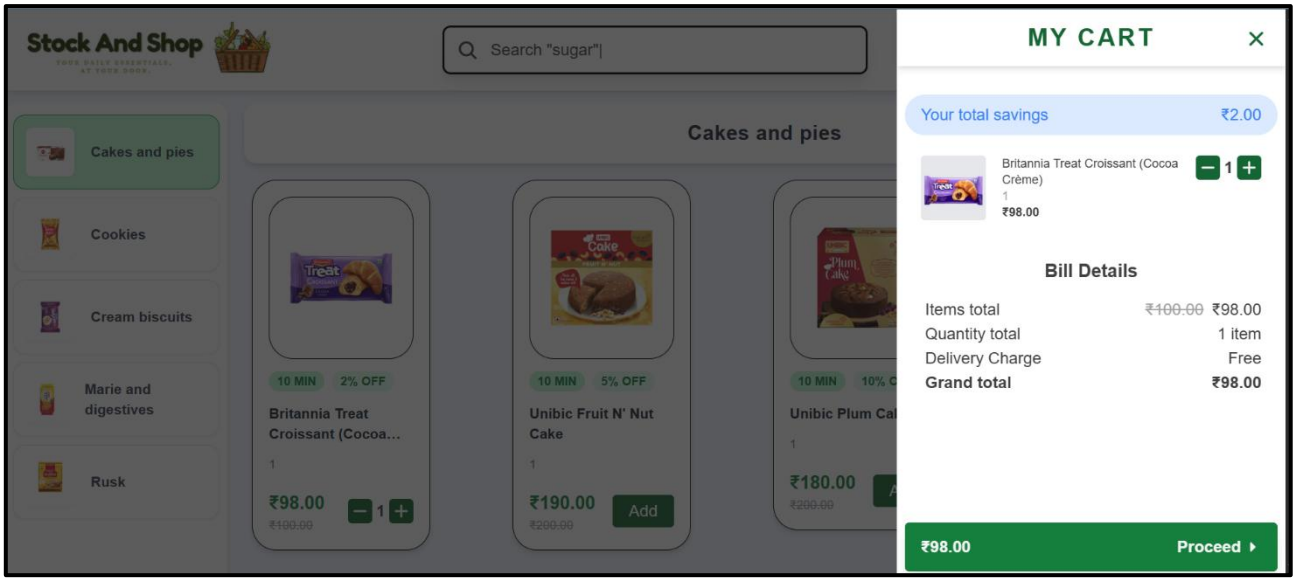
4.3 Process Logic of Payment Gateway (Stripe Integration)

The system integrates **Stripe Payment Gateway** to ensure secure, reliable, and verified online transactions.

4.3.1 Cart Management (Customer)

Process:

- Customer adds products to cart.
- Cart quantity updates dynamically.
- Cart data stored in MongoDB.



`

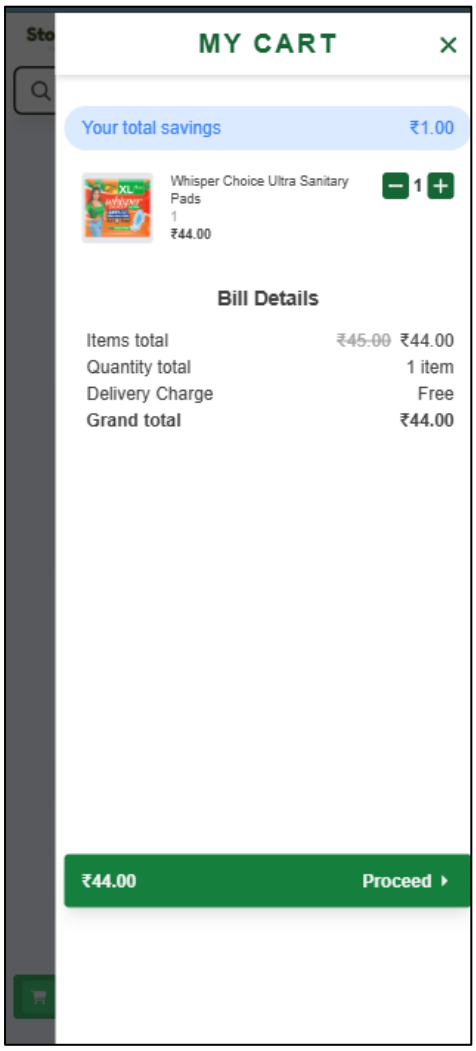
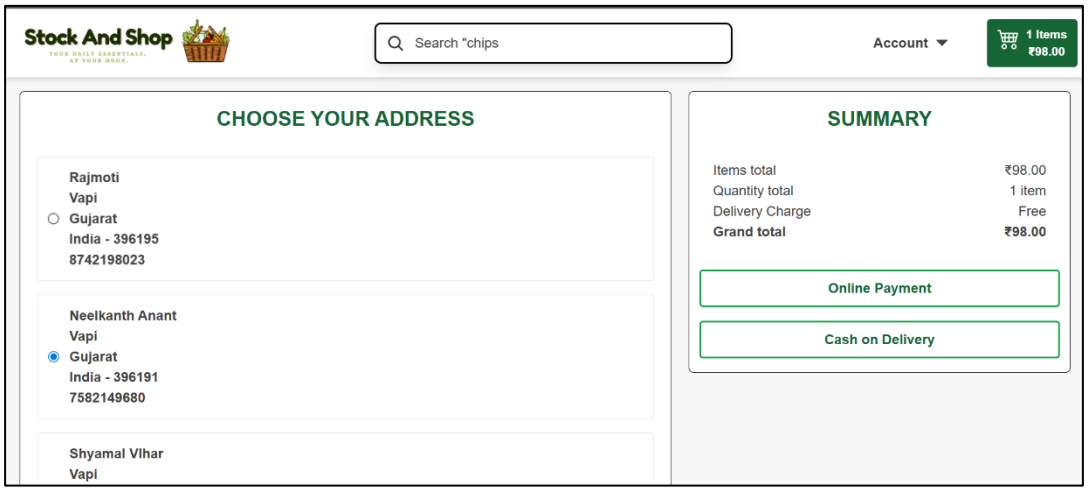


Figure 4.14: Customer – My Cart Page

4.3.2 Checkout Flow

Process:

- User clicks checkout.
- Backend prepares line items.
- Stripe Checkout Session is created.
- User redirected to Stripe payment page.



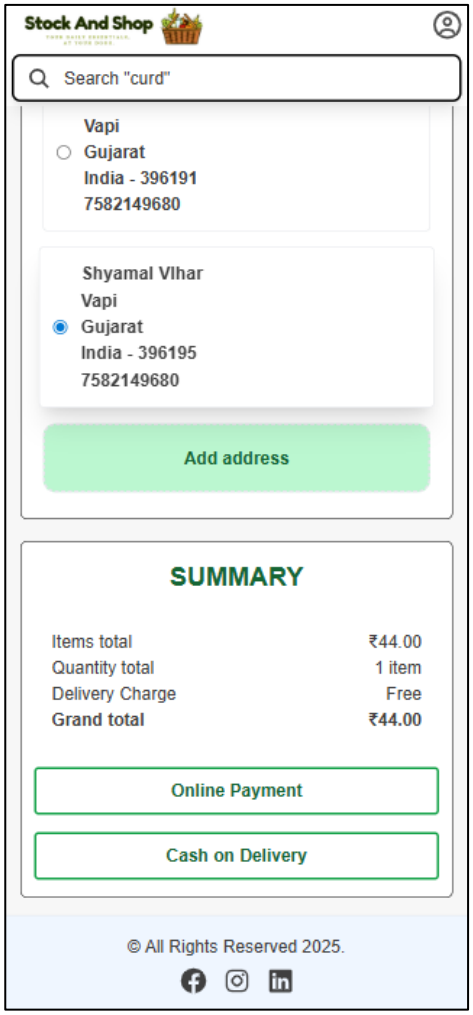


Figure 4.15: Checkout Page

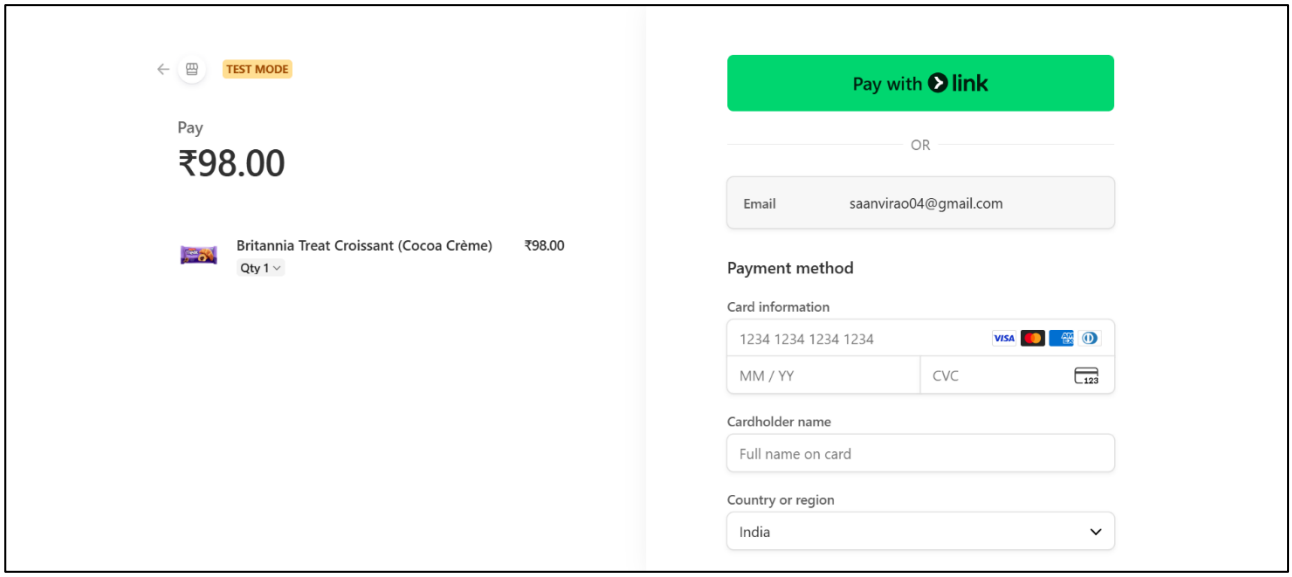
`

4.3.3 Secure Webhook Handler Implementation

After payment completion, Stripe sends a webhook to the backend.

Process:

- Stripe verifies payment.
- Sends event to /api/webhook/stripe.
- Backend validates webhook signature.
- Metadata (userId, cartId) is extracted.
- Order Fulfillment Logic is triggered.



`

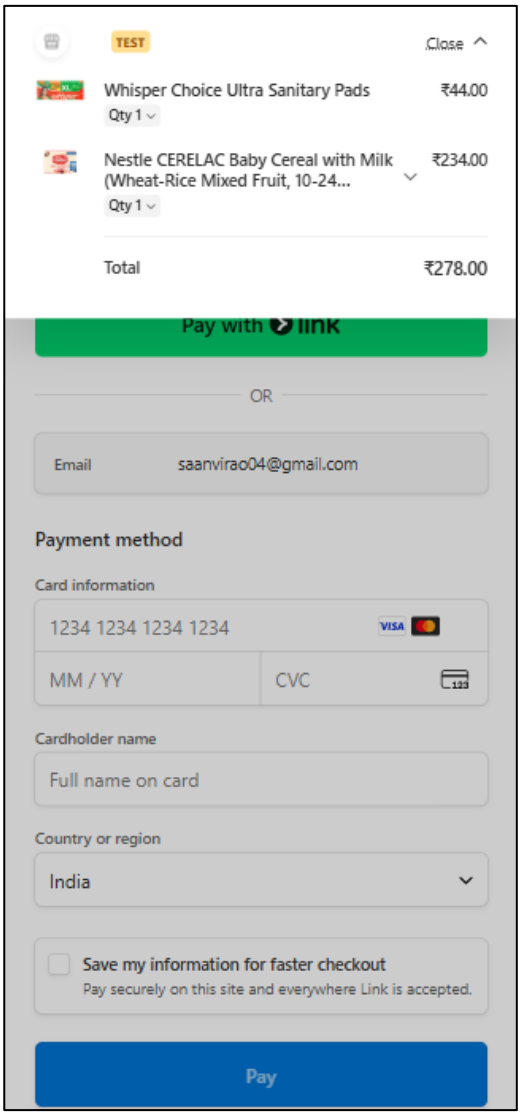


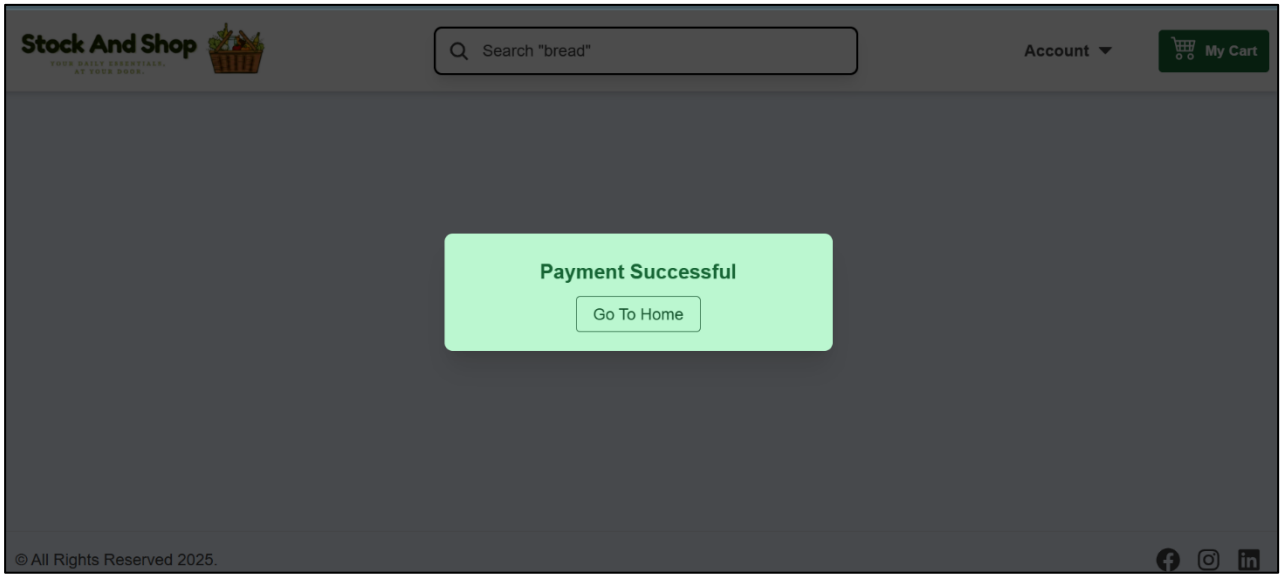
Figure 4.16: Stripe Page

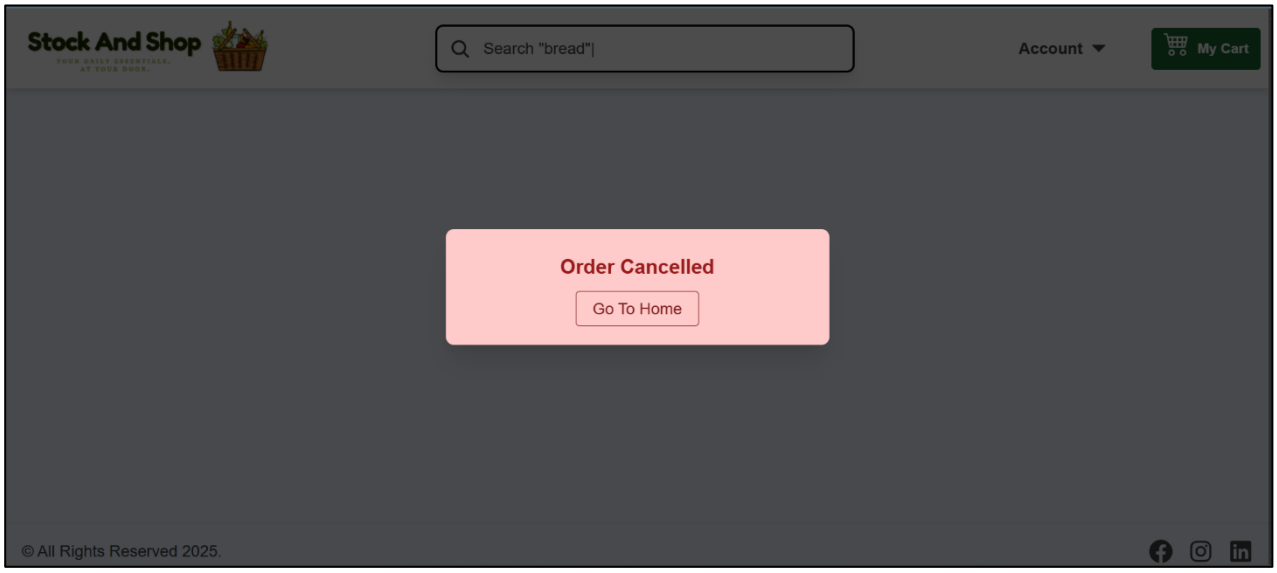
4.3.4 Order Fulfillment & Cart Clearance Logic

Process:

- Order record created in database.
- Product stock is updated.
- Cart items are cleared.
- Order confirmation response sent to user.

`


Stock And Shop
Search "bread"
Account
My Cart
Payment Successful
Go To Home
© All Rights Reserved 2025.


Stock And Shop
Search "bread"
Account
My Cart
Order Cancelled
Go To Home
© All Rights Reserved 2025.

`

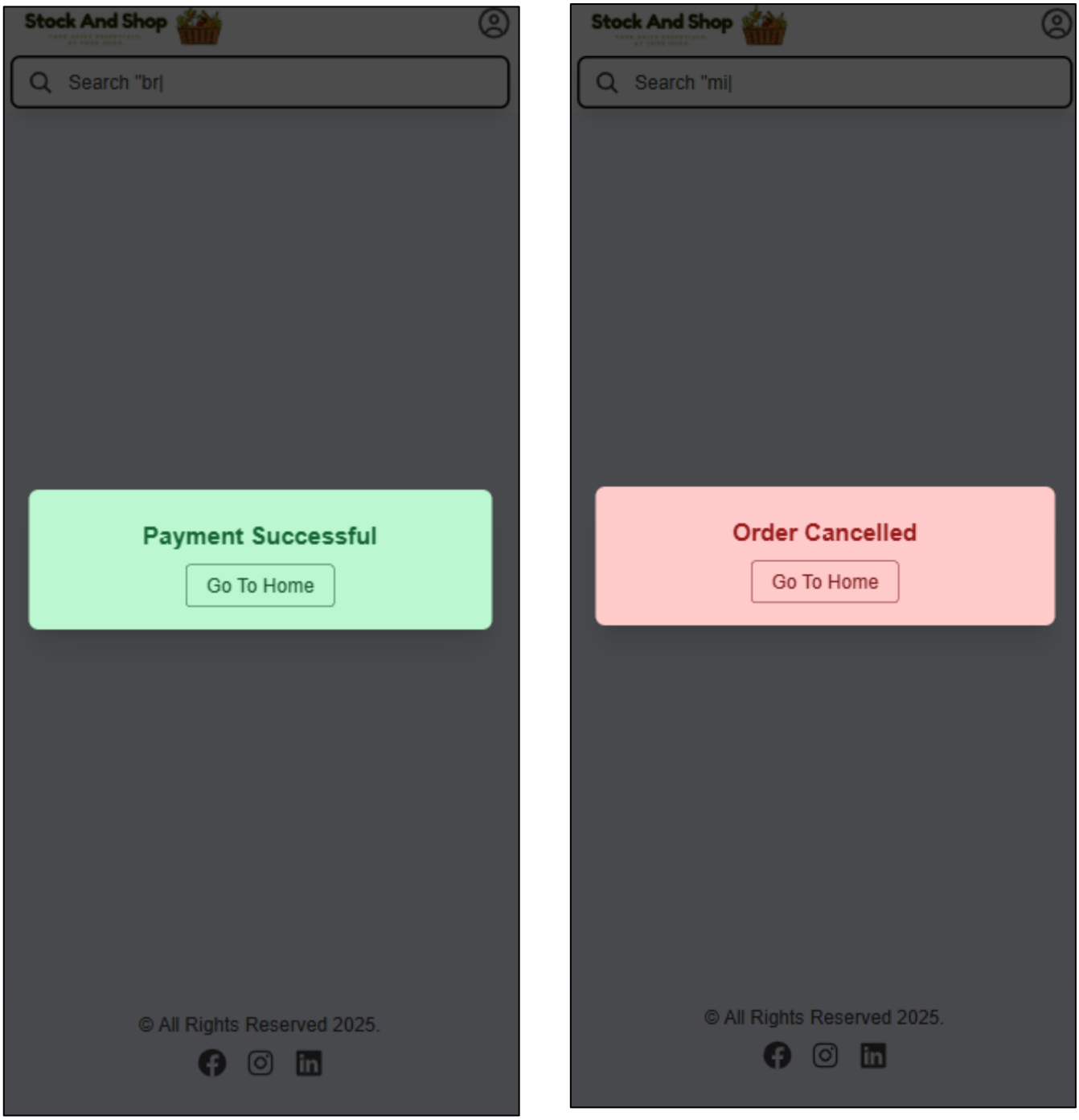


Figure 4.17: Order Confirmation Page

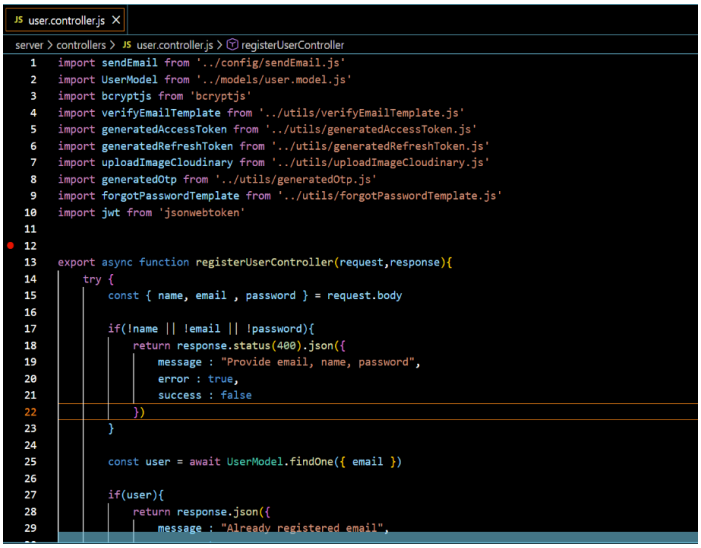
`

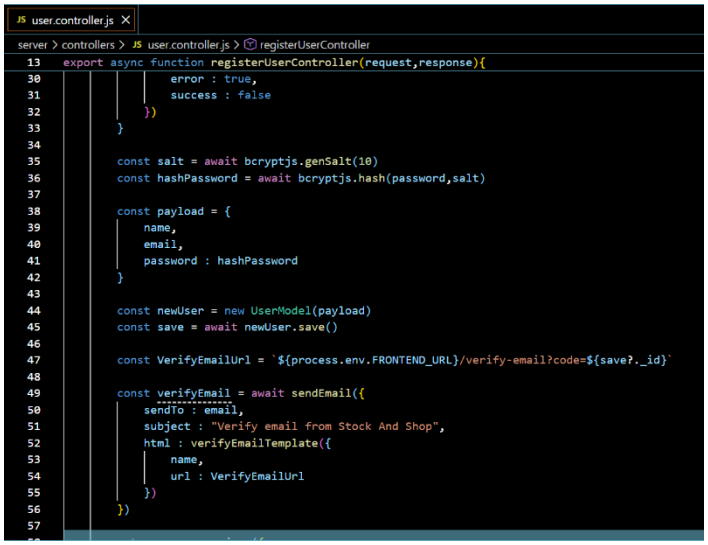
4.4 Code Implementation Details

This section highlights the core code files used in implementing authentication, inventory management, cart operations, and payment processing. Only essential files are included to maintain clarity.

4.4.1 Authentication Code

- **user.controller.js** – Handles user registration, login, and OTP-based password reset
- **auth.js** – JWT authentication middleware





user.controller.js

server > controllers > user.controller.js > registerUserController

```
13  export async function registerUserController(request,response){
58          return response.json({
59              message : "User registered successfully",
60              error : false,
61              success : true,
62              data : save
63          })
64
65      } catch (error) {
66          return response.status(500).json({
67              message : error.message || error,
68              error : true,
69              success : false
70          })
71      }
72  }
73
74  export async function verifyEmailController(request,response){
75      try {
76          const { code } = request.body
77
78          const user = await UserModel.findOne({ _id : code})
79
80          if(!user){
81              return response.status(400).json({
82                  message : "Invalid code",
83                  error : true,
84                  success : false
85              })
```

user.controller.js

server > controllers > user.controller.js > verifyEmailController

```
74  export async function verifyEmailController(request,response){
86          }
87
88          const updateUser = await UserModel.updateOne({ _id : code },{
89              verify_email : true
90          })
91
92          return response.json({
93              message : "Email verification successful.",
94              success : true,
95              error : false
96          })
97      } catch (error) {
98          return response.status(500).json({
99              message : error.message || error,
100             error : true,
101             success : true
102         })
103     }
104 }
105
106 //login controller
107 export async function loginController(request,response){
108     try {
109         const { email , password } = request.body
110
111
112         if(!email || !password){
113             return response.status(400).json({
```

function loginController(request: any, response: any): Promise<any>

user.controller.js

server > controllers > user.controller.js > verifyEmailController

```
107 export async function loginController(request,response){
112         if(!email || !password){
113             return response.status(400).json({
114                 message : "Provide email, password",
115                 error : true,
116                 success : false
117             })
118         }
119
120         const user = await UserModel.findOne({ email })
121
122         if(!user){
123             return response.status(400).json({
124                 message : "User not registered",
125                 error : true,
126                 success : false
127             })
128         }
129
130         if(user.status !== "Active"){
131             return response.status(400).json({
132                 message : "Contact to the Admin",
133                 error : true,
134                 success : false
135             })
136         }
137
138         const checkPassword = await bcryptjs.compare(password,user.password)
139
```

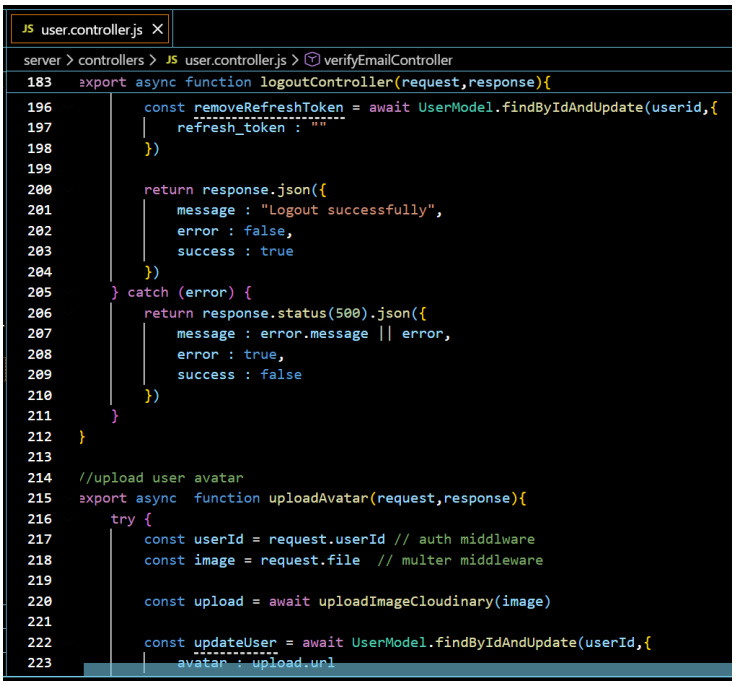
user.controller.js

server > controllers > user.controller.js > verifyEmailController

```
107 export async function loginController(request,response){
140         if(!checkPassword){
141             return response.status(400).json({
142                 message : "Check your password",
143                 error : true,
144                 success : false
145             })
146         }
147
148         const accesstoken = await generatedAccessToken(user._id)
149         const refreshToken = await generatedRefreshToken(user._id)
150
151         const updateUser = await UserModel.findByIdAndUpdate(user?._id,{
152             last_login_date : new Date()
153         })
154
155         const cookiesOption = {
156             httpOnly : true,
157             secure : true,
158             sameSite : "None"
159         }
160         response.cookie('accessToken',accesstoken,cookiesOption)
161         response.cookie('refreshToken',refreshToken,cookiesOption)
162
163         return response.json({
164             message : "Login successfully",
165             error : false,
166             success : true,
167             data : {
```

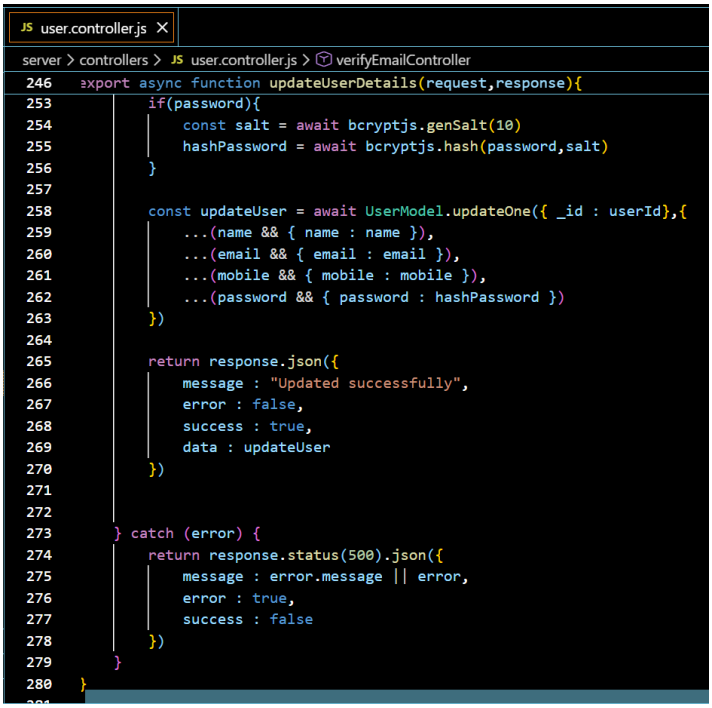
`

```
server > controllers > user.controller.js > verifyEmailController
export async function loginController(request,response){
            data : {
                accesstoken,
                refreshToken
            }
        })

    } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
    }
}

//logout controller
export async function logoutController(request,response){
    try {
        const userid = request.userId //middleware

        const cookiesOption = {
            httpOnly : true,
            secure : true,
            sameSite : "None"
        }

        response.clearCookie("accessToken",cookiesOption)
        response.clearCookie("refreshToken",cookiesOption)
```



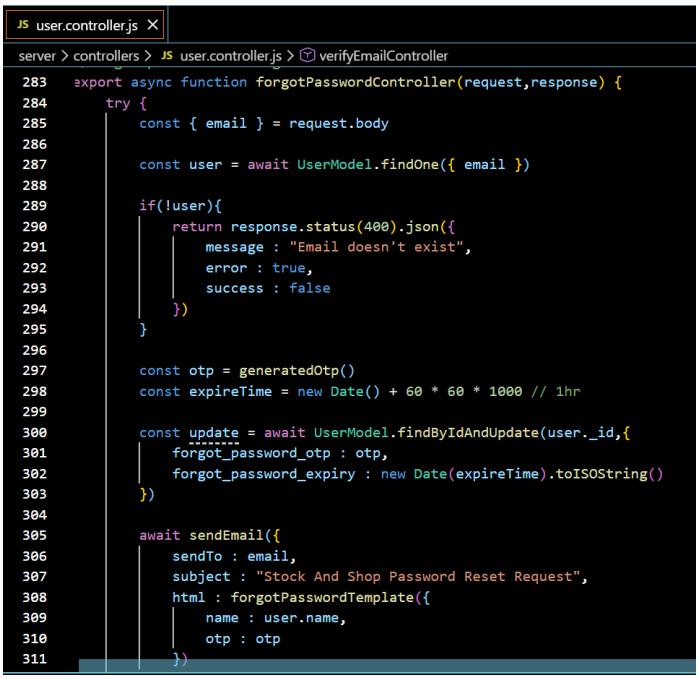
```
server > controllers > user.controller.js > verifyEmailController
export async function logoutController(request,response){
        const removeRefreshToken = await UserModel.findByIdAndUpdate(userid,{
            refresh_token : ""
        })

        return response.json({
            message : "Logout successfully",
            error : false,
            success : true
        })
    } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
    }
}

//upload user avatar
export async  function uploadAvatar(request,response){
    try {
        const userId = request.userId // auth middlware
        const image = request.file  // multer middleware

        const upload = await uploadImageCloudinary(image)

        const updateUser = await UserModel.findByIdAndUpdate(userId,{
            avatar : upload.url
```

```
server > controllers > user.controller.js > verifyEmailController
export async  function uploadAvatar(request,response){
        return response.json({
            message : "upload profile",
            success : true,
            error : false,
            data : {
                _id : userId,
                avatar : upload.url
            }
        })

    } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
    }
}

//update user details
export async function updateUserDetails(request,response){
    try {
        const userId = request.userId //auth middleware
        const { name, email, mobile, password } = request.body

        let hashPassword = ""

        if(password){
```



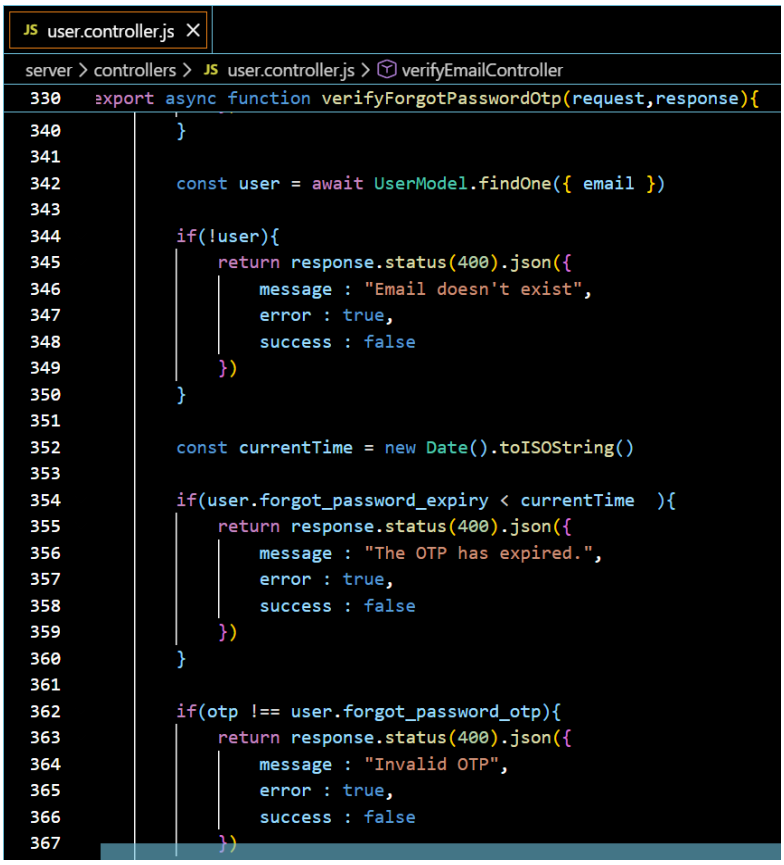
```
server > controllers > user.controller.js > verifyEmailController
export async function updateUserDetails(request,response){
        if(password){
            const salt = await bcryptjs.genSalt(10)
            hashPassword = await bcryptjs.hash(password,salt)
        }

        const updateUser = await UserModel.updateOne({ _id : userId},{
            ...(name && { name : name }),
            ...(email && { email : email }),
            ...(mobile && { mobile : mobile }),
            ...(password && { password : hashPassword })
        })

        return response.json({
            message : "Updated successfully",
            error : false,
            success : true,
            data : updateUser
        })


    } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
    }
}
```

`



```
user.controller.js
server > controllers > user.controller.js > verifyEmailController
283  export async function forgotPasswordController(request,response) {
284      try {
285          const { email } = request.body
286
287          const user = await UserModel.findOne({ email })
288
289          if(!user){
290              return response.status(400).json({
291                  message : "Email doesn't exist",
292                  error : true,
293                  success : false
294              })
295          }
296
297          const otp = generatedOtp()
298          const expireTime = new Date() + 60 * 60 * 1000 // 1hr
299
300          const update = await UserModel.findByIdAndUpdate(user._id,{
301              forgot_password_otp : otp,
302              forgot_password_expiry : new Date(expireTime).toISOString()
303          })
304
305          await sendEmail({
306              sendTo : email,
307              subject : "Stock And Shop Password Reset Request",
308              html : forgotPasswordTemplate({
309                  name : user.name,
310                  otp : otp
311              })
```

```
user.controller.js
server > controllers > user.controller.js > verifyEmailController
283  export async function forgotPasswordController(request,response) {
312          })
313
314          return response.json({
315              message : "Please check your email",
316              error : false,
317              success : true
318          })
319
320      } catch (error) {
321          return response.status(500).json({
322              message : error.message || error,
323              error : true,
324              success : false
325          })
326      }
327  }
328
329  //verify forgot password otp
330  export async function verifyForgotPasswordOtp(request,response){
331      try {
332          const { email , otp }  = request.body
333
334          if(!email || !otp){
335              return response.status(400).json({
336                  message : "Required fields: Email, OTP",
337                  error : true,
338                  success : false
339              })
```

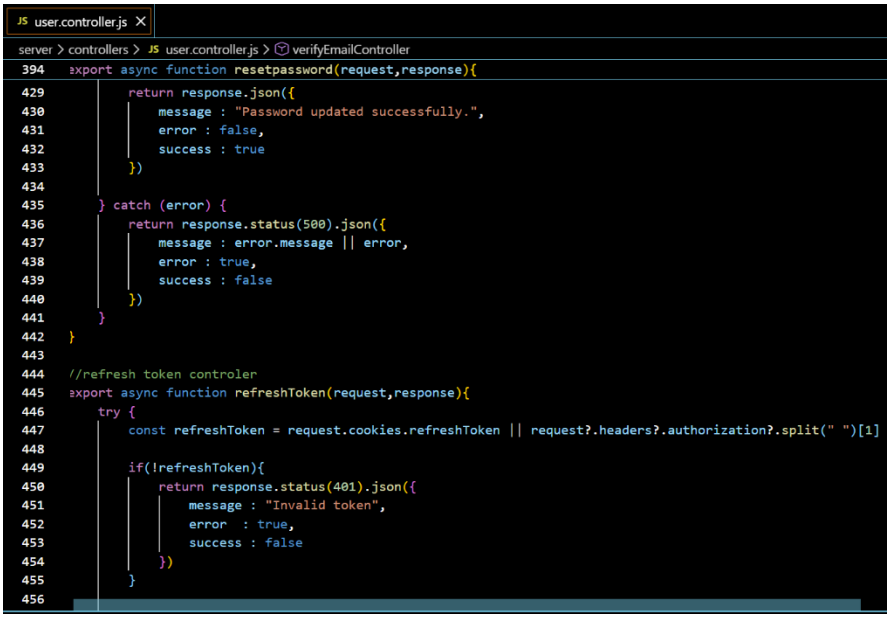


```
user.controller.js
server > controllers > user.controller.js > verifyEmailController
330  export async function verifyForgotPasswordOtp(request,response){
340      }
341
342      const user = await UserModel.findOne({ email })
343
344      if(!user){
345          return response.status(400).json({
346              message : "Email doesn't exist",
347              error : true,
348              success : false
349          })
350      }
351
352      const currentTime = new Date().toISOString()
353
354      if(user.forgot_password_expiry < currentTime  ){
355          return response.status(400).json({
356              message : "The OTP has expired.",
357              error : true,
358              success : false
359          })
360      }
361
362      if(otp !== user.forgot_password_otp){
363          return response.status(400).json({
364              message : "Invalid OTP",
365              error : true,
366              success : false
367          })
```

`

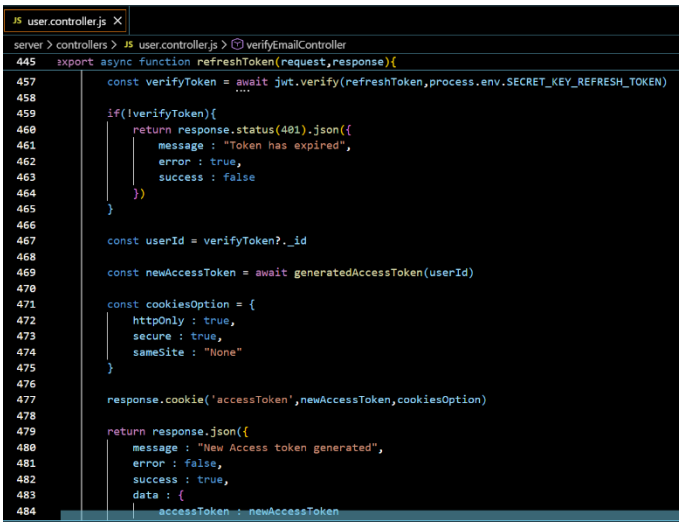
```
user.controller.js
server > controllers > user.controller.js > verifyEmailController
export async function verifyForgotPasswordOtp(request,response){
        const updateUser = await UserModel.findByIdAndUpdate(user?._id,{
            forgot_password_otp : "",
            forgot_password_expiry : ""
        })

        return response.json({
            message : "OTP Verified Successfully",
            error : false,
            success : true
        })

    } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
    }
}

//reset the password
export async function resetpassword(request,response){
    try {
        const { email , newPassword, confirmPassword } = request.body

        if(!email || !newPassword || !confirmPassword){
            return response.status(400).json({
                message : "Please provide all required fields: Email, New Password, and Confirm Password."
```

```
user.controller.js
server > controllers > user.controller.js > verifyEmailController
export async function resetpassword(request,response){
            })
        }

        const user = await UserModel.findOne({ email })

        if(!user){
            return response.status(400).json({
                message : "Email doesn't exist",
                error : true,
                success : false
            })
        }

        if(newPassword !== confirmPassword){
            return response.status(400).json({
                message : "New Password and Confirm Password must match.",
                error : true,
                success : false,
            })
        }

        const salt = await bcryptjs.genSalt(10)
        const hashPassword = await bcryptjs.hash(newPassword,salt)

        const update = await UserModel.findOneAndUpdate(user._id,{
            password : hashPassword
        })
```

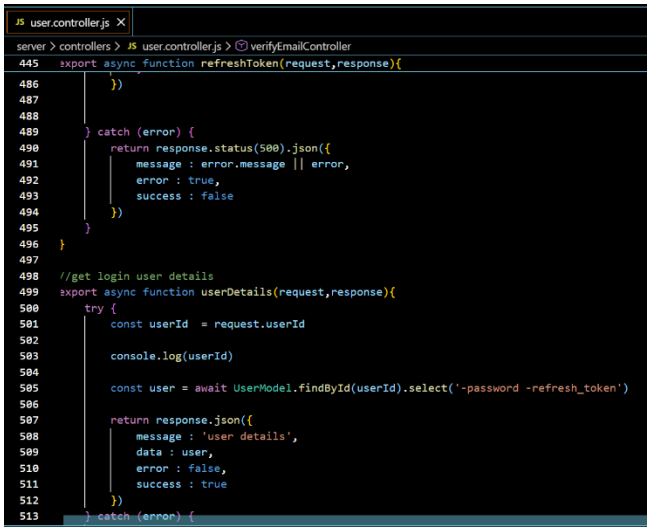


```
user.controller.js
server > controllers > user.controller.js > verifyEmailController
export async function resetpassword(request,response){
            return response.json({
                message : "Password updated successfully.",
                error : false,
                success : true
            })

        } catch (error) {
            return response.status(500).json({
                message : error.message || error,
                error : true,
                success : false
            })
        }
    }

    //refresh token controler
    export async function refreshToken(request,response){
        try {
            const refreshToken = request.cookies.refreshToken || request?.headers?.authorization?.split(" ")[1]

            if(!refreshToken){
                return response.status(401).json({
                    message : "Invalid token",
                    error  : true,
                    success : false
                })
            }
```

`



```
user.controller.js
server > controllers > user.controller.js > verifyEmailController
445  export async function refreshToken(request,response){
457      const verifyToken = await jwt.verify(refreshToken,process.env.SECRET_KEY_REFRESH_TOKEN)
458
459      if(!verifyToken){
460          return response.status(401).json({
461              message : "Token has expired",
462              error : true,
463              success : false
464          })
465      }
466
467      const userId = verifyToken?._id
468
469      const newAccessToken = await generatedAccessToken(userId)
470
471      const cookiesOption = {
472          httpOnly : true,
473          secure : true,
474          sameSite : "None"
475      }
476
477      response.cookie('accessToken',newAccessToken,cookiesOption)
478
479      return response.json({
480          message : "New Access token generated",
481          error : false,
482          success : true,
483          data : {
484              accessToken : newAccessToken
```



```
user.controller.js
server > controllers > user.controller.js > verifyEmailController
445  export async function refreshToken(request,response){
486          })
487
488
489      } catch (error) {
490          return response.status(500).json({
491              message : error.message || error,
492              error : true,
493              success : false
494          })
495      }
496  }
497
498  //get login user details
499  export async function userDetails(request,response){
500      try {
501          const userId  = request.userId
502
503          console.log(userId)
504
505          const user = await UserModel.findById(userId).select('-password -refresh_token')
506
507          return response.json({
508              message : 'user details',
509              data : user,
510              error : false,
511              success : true
512          })
513      } catch (error) {
```

```
user.controller.js
server > controllers > user.controller.js > verifyEmailController
499  export async function userDetails(request,response){
513      } catch (error) {
514          return response.status(500).json({
515              message : "Something is wrong",
516              error : true,
517              success : false
518          })
519      }
520  }
```

```
auth.js
server > middleware > auth.js > ...
1   import jwt from 'jsonwebtoken'
2   const auth = async(request,response,next)=>{
3       try {
4           const token = request.cookies.accessToken || request?.headers?.authorization?.split(" ")[1]
5
6           if(!token){
7              return response.status(401).json({
8                  message : "Provide token"
9              })
10          }
11          const decode = await jwt.verify(token,process.env.SECRET_KEY_ACCESS_TOKEN)
12          if(!decode){
13              return response.status(401).json({
14                  message : "unauthorized access",
15                  error : true,
16                  success : false
17              })
18          }
19          request.userId = decode.id
20          next()
21      } catch (error) {
22          return response.status(500).json({
23              message : "You have not logged in",///error.message || error,
24              error : true,
25              success : false
26          })
27      }
28  }
29  export default auth
```

Figure 4.18: Authentication Controller and JWT Middleware

`

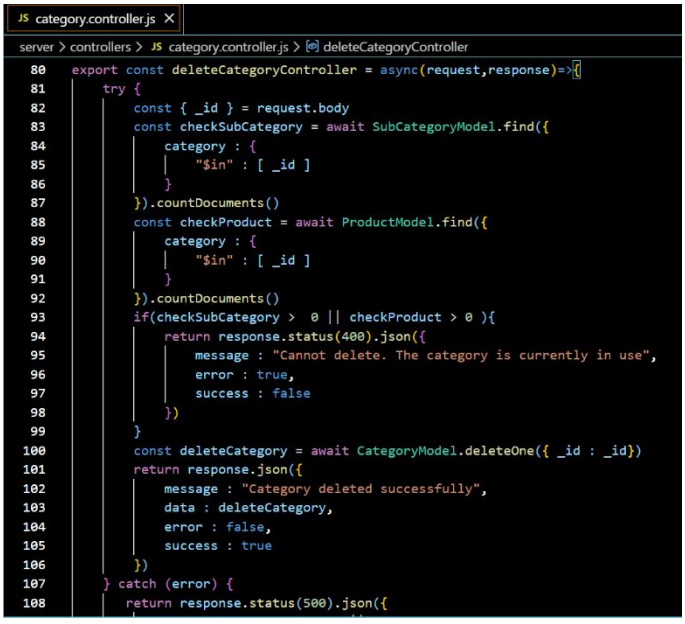
4.4.2 Product & Inventory Code

- **category.controller.js** – Category CRUD operations
- **subCategory.controller.js** – Sub-category management
- **product.controller.js** – Product upload and stock updates

```
server > controllers > category.controller.js > deleteCategoryController
import CategoryModel from "../models/category.model.js";
import SubCategoryModel from "../models/subCategory.model.js";
import ProductModel from "../models/product.model.js";
export const AddCategoryController = async(request,response)=>{
    try {
        const { name , image } = request.body

        if(!name || !image){
            return response.status(400).json({
                message : "Enter required fields",
                error : true,
                success : false
            })
        }
        const addCategory = new CategoryModel({
            name,
            image
        })
        const saveCategory = await addCategory.save()
        if(!saveCategory){
            return response.status(500).json({
                message : "Not Created",
                error : true,
                success : false
            })
        }
        return response.json({
            message : "Add Category",
            data : saveCategory,
```

```
server > controllers > category.controller.js > deleteCategoryController
export const AddCategoryController = async(request,response)=>{
            data : saveCategory,
            success : true,
            error : false
        })
    } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
    }
}
export const getCategoryController = async(request,response)=>{
    try {
        const data = await CategoryModel.find().sort({ createdAt : -1 })
        return response.json({
            data : data,
            error : false,
            success : true
        })
    } catch (error) {
        return response.status(500).json({
            message : error.messsage || error,
            error : true,
            success : false
        })
    }
}
```

```
server > controllers > category.controller.js > deleteCategoryController
export const updateCategoryController = async(request,response)=>{
    try {
        const { _id ,name, image } = request.body
        const update = await CategoryModel.updateOne({
            _id : _id
        },{
            name,
            image
        })
        return response.json({
            message : "Category updated",
            success : true,
            error : false,
            data : update
        })
    } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
    }
}
```



```
server > controllers > category.controller.js > deleteCategoryController
export const deleteCategoryController = async(request,response)=>{
    try {
        const { _id } = request.body
        const checkSubCategory = await SubCategoryModel.find({
            category : {
                "$in" : [ _id ]
            }
        }).countDocuments()
        const checkProduct = await ProductModel.find({
            category : {
                "$in" : [ _id ]
            }
        }).countDocuments()
        if(checkSubCategory >  0 || checkProduct > 0 ){
            return response.status(400).json({
                message : "Cannot delete. The category is currently in use",
                error : true,
                success : false
            })
        }
        const deleteCategory = await CategoryModel.deleteOne({ _id : _id})
        return response.json({
            message : "Category deleted successfully",
            data : deleteCategory,
            error : false,
            success : true
        })
    } catch (error) {
        return response.status(500).json({
```

`

```
        return response.status(500).json({
            message : error.message || error,
            success : false,
            error : true
        })
    }
}
```

product.controller.js

server > controllers > product.controller.js > ...

```
import ProductModel from "../models/product.model.js";

export const createProductController = async(request,response)=>{
    try {
        const {
            name ,
            image ,
            category,
            subCategory,
            unit,
            stock,
            price,
            discount,
            description,
            more_details,
        } = request.body

        if(!name || !image[0] || !category[0] || !subCategory[0] || !unit || !price || !description ){
            return response.status(400).json({
                message : "Enter required fields",
                error : true,
                success : false
            })
        }

        const product = new ProductModel({
            name ,
            image ,
            category,
```

product.controller.js

server > controllers > product.controller.js > ...

```
export const createProductController = async(request,response)=>{
        const product = new ProductModel({
            subCategory,
            unit,
            stock,
            price,
            discount,
            description,
            more_details,
        })
        const saveProduct = await product.save()

        return response.json({
            message : "Product Created Successfully",
            data : saveProduct,
            error : false,
            success : true
        })

    } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
    }
}

export const getProductController = async(request,response)=>{
```

product.controller.js

server > controllers > product.controller.js > ...

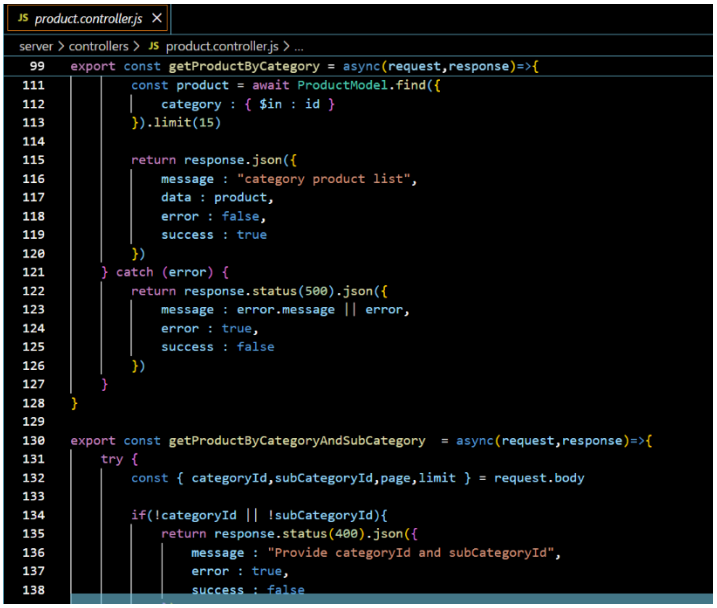
```
export const getProductController = async(request,response)=>{
    try {

        let { page, limit, search } = request.body

        if(!page){
            page = 1
        }

        if(!limit){
            limit = 10
        }

        const query = search ? {
            $text : {
                $search : search
            }
        } : {}

        const skip = (page - 1) * limit

        const [data,totalCount] = await Promise.all([
            ProductModel.find(query).sort({createdAt : -1 }).skip(skip).limit(limit).populate('category subCategory'),
            ProductModel.countDocuments(query)
        ])

        return response.json({
            message : "Product data",
            error : false,
```

product.controller.js

server > controllers > product.controller.js > ...

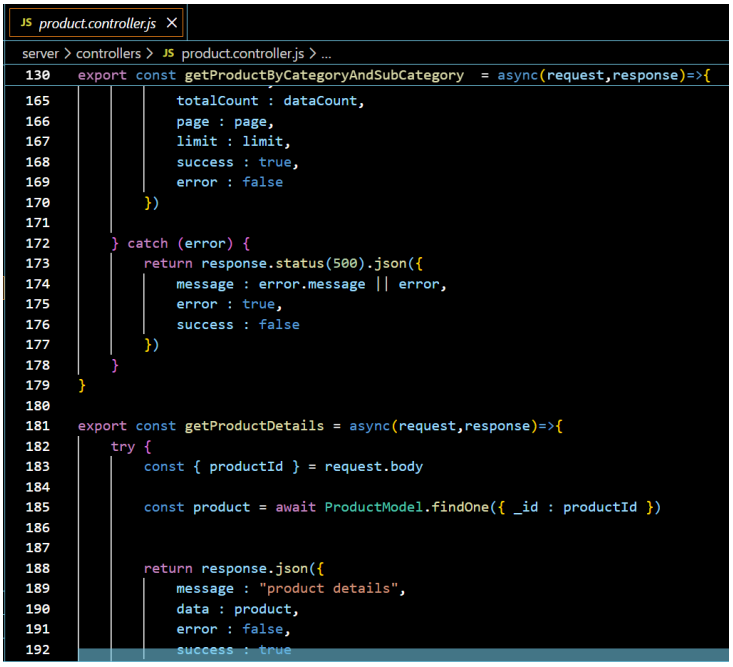
```
export const getProductController = async(request,response)=>{
            success : true,
            totalCount : totalCount,
            totalNoPage : Math.ceil( totalCount / limit),
            data : data
        })
    } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
    }
}

export const getProductByCategory = async(request,response)=>{
    try {
        const { id } = request.body

        if(!id){
            return response.status(400).json({
                message : "provide category id",
                error : true,
                success : false
            })
        }
```

`



```
JS product.controller.js
server > controllers > JS product.controller.js > ...
 99  export const getProductByCategory = async(request,response)=>{
111      const product = await ProductModel.find({
112          category : { $in : id }
113      }).limit(15)
114
115      return response.json({
116          message : "category product list",
117          data : product,
118          error : false,
119          success : true
120      })
121  } catch (error) {
122      return response.status(500).json({
123          message : error.message || error,
124          error : true,
125          success : false
126      })
127  }
128  }
129
130  export const getProductByCategoryAndSubCategory  = async(request,response)=>{
131      try {
132          const { categoryId,subCategoryId,page,limit } = request.body
133
134          if(!categoryId || !subCategoryId){
135              return response.status(400).json({
136                  message : "Provide categoryId and subCategoryId",
137                  error : true,
138                  success : false
```

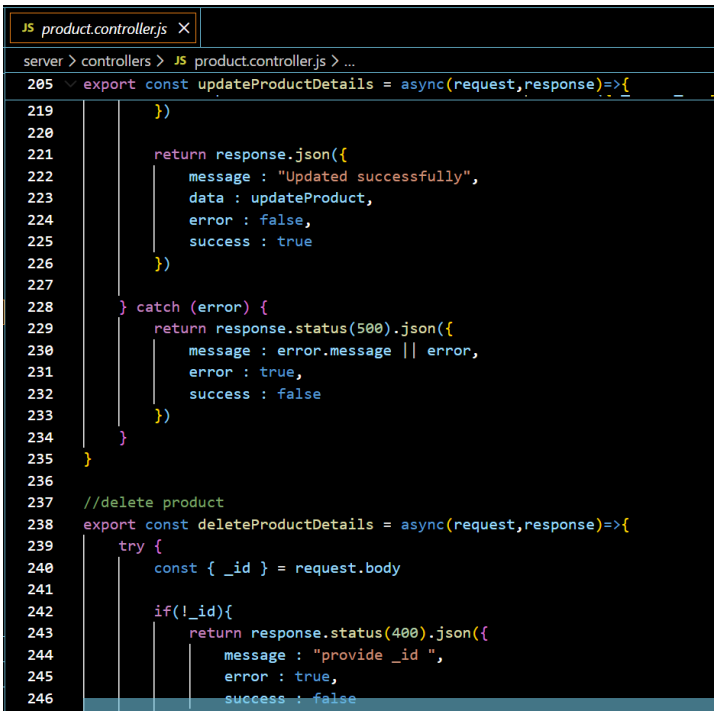
```
JS product.controller.js
server > controllers > JS product.controller.js > ...
130  export const getProductByCategoryAndSubCategory  = async(request,response)=>{
137              error : true,
138              success : false
139          })
140      }
141
142      if(!page){
143          page = 1
144      }
145
146      if(!limit){
147          limit = 10
148      }
149
150      const query = {
151          category : { $in :categoryId  },
152          subCategory : { $in : subCategoryId }
153      }
154
155      const skip = (page - 1) * limit
156
157      const [data,dataCount] = await Promise.all([
158          ProductModel.find(query).sort({createdAt : -1 }).skip(skip).limit(limit),
159          ProductModel.countDocuments(query)
160      ])
161
162      return response.json({
163          message : "Product list",
164          data : data,
```



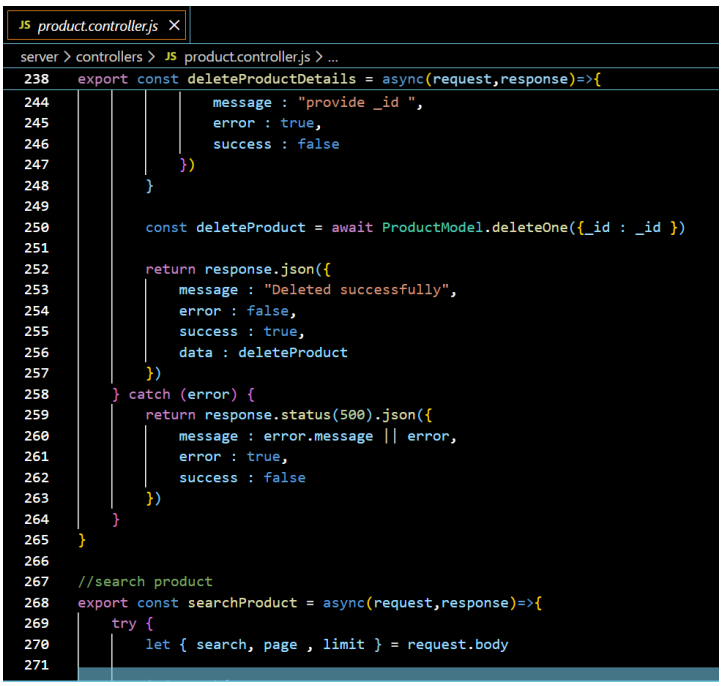
```
JS product.controller.js
server > controllers > JS product.controller.js > ...
130  export const getProductByCategoryAndSubCategory  = async(request,response)=>{
165              totalCount : dataCount,
166              page : page,
167              limit : limit,
168              success : true,
169              error : false
170          })
171
172      } catch (error) {
173          return response.status(500).json({
174              message : error.message || error,
175              error : true,
176              success : false
177          })
178      }
179  }
180
181  export const getProductDetails = async(request,response)=>{
182      try {
183          const { productId } = request.body
184
185          const product = await ProductModel.findOne({ _id : productId })
186
187
188          return response.json({
189              message : "product details",
190              data : product,
191              error : false,
192              success : true
```

```
JS product.controller.js
server > controllers > JS product.controller.js > ...
181  export const getProductDetails = async(request,response)=>{
191              error : false,
192              success : true
193          })
194
195      } catch (error) {
196          return response.status(500).json({
197              message : error.message || error,
198              error : true,
199              success : false
200          })
201      }
202  }
203
204  //update product
205  export const updateProductDetails = async(request,response)=>{
206      try {
207          const { _id } = request.body
208
209          if(!_id){
210              return response.status(400).json({
211                  message : "provide product _id",
212                  error : true,
213                  success : false
214              })
215          }
216
217          const updateProduct = await ProductModel.updateOne({ _id : _id },{
218              ...request.body
```

`



```
JS product.controller.js
server > controllers > JS product.controller.js > ...
205 export const updateProductDetails = async(request,response)=>{
219         })
220
221         return response.json({
222             message : "Updated successfully",
223             data : updateProduct,
224             error : false,
225             success : true
226         })
227
228     } catch (error) {
229         return response.status(500).json({
230             message : error.message || error,
231             error : true,
232             success : false
233         })
234     }
235 }
236
237 //delete product
238 export const deleteProductDetails = async(request,response)=>{
239     try {
240         const { _id } = request.body
241
242         if(!_id){
243             return response.status(400).json({
244                 message : "provide _id ",
245                 error : true,
246                 success : false
```



```
JS product.controller.js
server > controllers > JS product.controller.js > ...
238 export const deleteProductDetails = async(request,response)=>{
244                 message : "provide _id ",
245                 error : true,
246                 success : false
247             })
248         }
249
250         const deleteProduct = await ProductModel.deleteOne({_id : _id })
251
252         return response.json({
253             message : "Deleted successfully",
254             error : false,
255             success : true,
256             data : deleteProduct
257         })
258     } catch (error) {
259         return response.status(500).json({
260             message : error.message || error,
261             error : true,
262             success : false
263         })
264     }
265 }
266
267 //search product
268 export const searchProduct = async(request,response)=>{
269     try {
270         let { search, page , limit } = request.body
271
```

```
JS product.controller.js
server > controllers > JS product.controller.js > ...
267 //search product
268 export const searchProduct = async(request,response)=>{
269     try {
270         let { search, page , limit } = request.body
271
272         if(!page){
273             page = 1
274         }
275         if(!limit){
276             limit  = 10
277         }
278
279         const query = search ? {
280             $text : {
281                 $search : search
282             }
283         } : {}
284
285         const skip = ( page - 1) * limit
286
287         const [data,dataCount] = await Promise.all([
288             ProductModel.find(query).sort({ createdAt  : -1 }).skip(skip).limit(limit).populate('category subCategory
289             ProductModel.countDocuments(query)
290         ])
291
292         return response.json({
293             message : "Product data",
294             error : false,
295             success : true,
```

`

product.controller.js

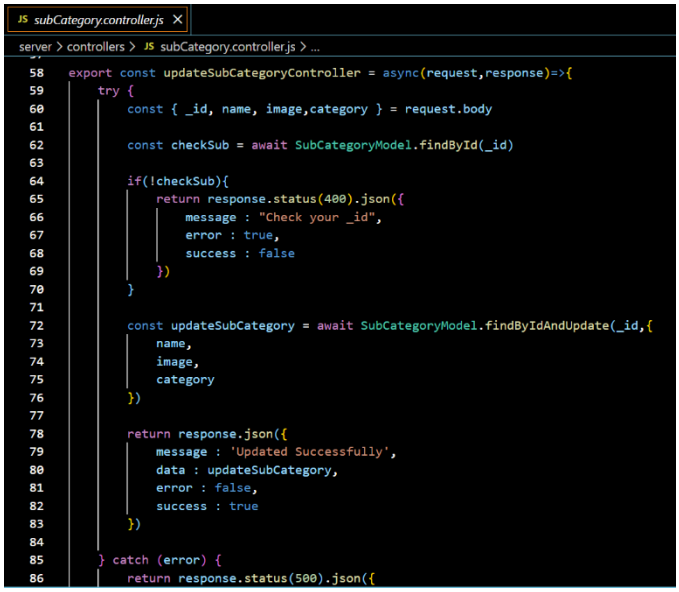
server > controllers > product.controller.js > ...

```
268  export const searchProduct = async(request,response)=
290              ])
291
292              return response.json({
293                  message : "Product data",
294                  error : false,
295                  success : true,
296                  data : data,
297                  totalCount :dataCount,
298                  totalPage : Math.ceil(dataCount/limit),
299                  page : page,
300                  limit : limit
301              })
302
303
304          } catch (error) {
305              return response.status(500).json({
306                  message : error.message || error,
307                  error : true,
308                  success : false
309              })
310          }
311      }
```

subCategory.controller.js

server > controllers > subCategory.controller.js > ...

```
1   import SubCategoryModel from "../models/subCategory.model.js";
2
3   export const AddSubCategoryController = async(request,response)=>{
4       try {
5           const { name, image, category } = request.body
6
7           if(!name && !image && !category[0] ){
8               return response.status(400).json({
9                   message : "Provide name, image, category",
10                  error : true,
11                  success : false
12              })
13          }
14
15          const payload = {
16              name,
17              image,
18              category
19          }
20
21          const createSubCategory = new SubCategoryModel(payload)
22          const save = await createSubCategory.save()
23
24          return response.json({
25              message : "Sub Category Created",
26              data : save,
27              error : false,
28              success : true
29          })
```



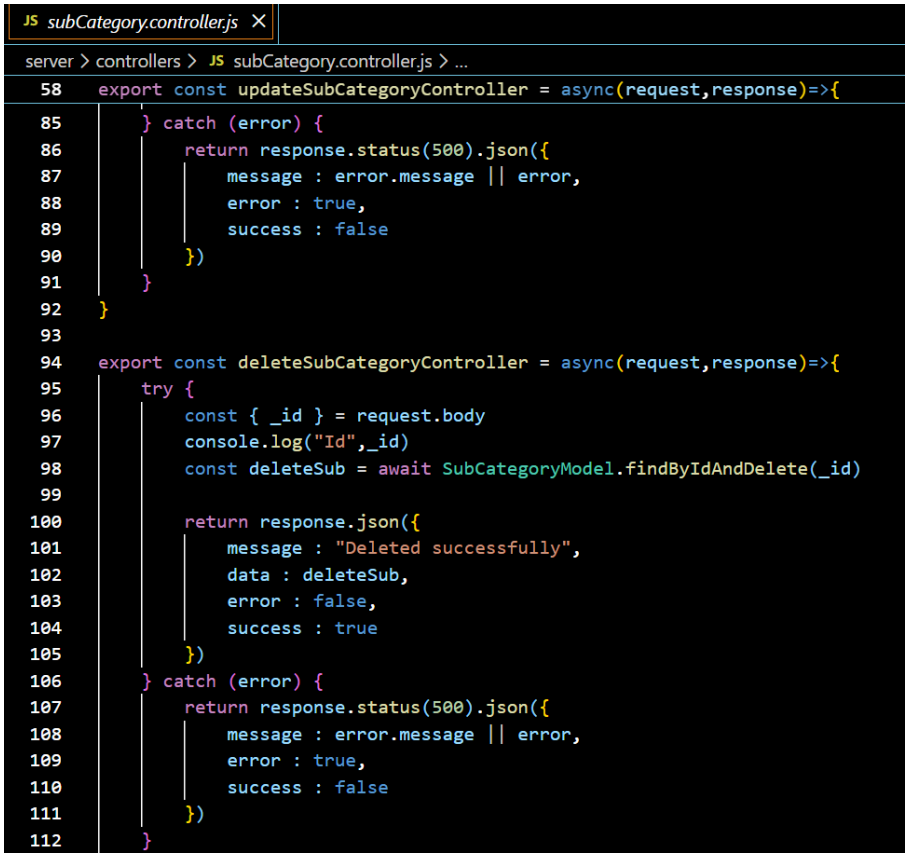


Figure 4.19: Product and Inventory Controllers

`

4.4.3 Cart & Order Code

- **cart.controller.js** – Add, update, and remove cart items
- **order.controller.js** – Order creation and order history

```
import CartProductModel from "../models/cartproduct.model.js";
import UserModel from "../models/user.model.js";
export const addToCartItemController = async(request,response)=>{
    try {
        const  userId = request.userId
        const { productId } = request.body
        if(!productId){
            return response.status(402).json({
                message : "Provide productId",
                error : true,
                success : false
            })
        }
        const checkItemCart = await CartProductModel.findOne({
            userId : userId,
            productId : productId
        })
        if(checkItemCart){
            return response.status(400).json({
                message : "Item is already in the cart"
            })
        }
        const cartItem = new CartProductModel({
            quantity : 1,
            userId : userId,
            productId : productId
        })
        const save = await cartItem.save()
        const updateCartUser = await UserModel.updateOne({ _id : userId},{
```

```
export const addToCartItemController = async(request,response)=>{
        const updateCartUser = await UserModel.updateOne({ _id : userId},{
            $push : {
                shopping_cart : productId
            }
        })
        return response.json({
            data : save,
            message : "Item added successfully",
            error : false,
            success : true
        })
    } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
    }
}
export const getCartItemController = async(request,response)=>{
    try {
        const userId = request.userId

        const cartItem =  await CartProductModel.find({
            userId : userId
        }).populate('productId')

        return response.json({
```

```
export const getCartItemController = async(request,response)=>{
            data : cartItem,
            error : false,
            success : true
        })
    } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
    }
}
export const updateCartItemQtyController = async(request,response)=>{
    try {
        const userId = request.userId
        const { _id,qty } = request.body

        if(!_id ||  !qty){
            return response.status(400).json({
                message : "provide _id, qty"
            })
        }
        const updateCartitem = await CartProductModel.updateOne({
            _id : _id,
            userId : userId
        },{
            quantity : qty
        })
```

```
export const updateCartItemQtyController = async(request,response)=>{
        return response.json({
            message : "Update cart",
            success : true,
            error : false,
            data : updateCartitem
        })
    } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
    }
}
export const deleteCartItemQtyController = async(request,response)=>{
    try {
      const userId = request.userId // middleware
      const { _id } = request.body
      if(!_id){
        return response.status(400).json({
            message : "Provide _id",
            error : true,
            success : false
        })
      }
      const deleteCartItem  = await CartProductModel.deleteOne({_id : _id, userId : userId })
      return response.json({
        message : "Item removed",
```

`

cart.controller.js

server > controllers > cart.controller.js > deleteCartItemQtyController

```
export const deleteCartItemQtyController = async(request,response)=>{
        const deleteCartItem  = await CartProductModel.deleteOne({_id : _id, userId : userId })
        return response.json({
          message : "Item removed",
          error : false,
          success : true,
          data : deleteCartItem
        })
      } catch (error) {
        return response.status(500).json({
            message : error.message || error,
            error : true,
            success : false
        })
      }
}
```

order.controller.js

server > controllers > order.controller.js > ...

```
import Stripe from "../config/stripe.js";
import CartProductModel from "../models/cartproduct.model.js";
import OrderModel from "../models/order.model.js";
import UserModel from "../models/user.model.js";
import mongoose from "mongoose";

// Helper: calculate price after discount
export const priceWithDiscount = (price, discount = 0) => {
  const p = Number(price) || 0;
  const dis = Number(discount) || 0;
  const discountAmount = Math.ceil((p * dis) / 100);
  return p - discountAmount;
};

// --- Cash on Delivery Order ---
export async function CashOnDeliveryOrderController(req, res) {
  try {
    const userId = req.userId;
    const { list_items, totalAmt, addressId, subTotalAmt } = req.body;

    const payload = list_items.map(el => ({
      userId,
      orderId: `ORD-${new mongoose.Types.ObjectId()}`,
      productId: el.productId._id,
      product_details: {
        name: el.productId.name,
        image: el.productId.image
      },
      paymentId: "",
```

order.controller.js

server > controllers > order.controller.js > ...

```
export async function CashOnDeliveryOrderController(req, res) {
    const payload = list_items.map(el => ({
      paymentId: "",
      payment_status: "CASH ON DELIVERY",
      delivery_address: addressId,
      subTotalAmt,
      totalAmt
    }));

    const generatedOrder = await OrderModel.insertMany(payload);

    await CartProductModel.deleteMany({ userId });
    await UserModel.updateOne({ _id: userId }, { shopping_cart: [] });

    return res.json({
      message: "Order successfully placed",
      success: true,
      error: false,
      data: generatedOrder
    });
  } catch (err) {
    return res.status(500).json({
      message: err.message || err,
      success: false,
      error: true
    });
  }
}
```

order.controller.js

server > controllers > order.controller.js > ...

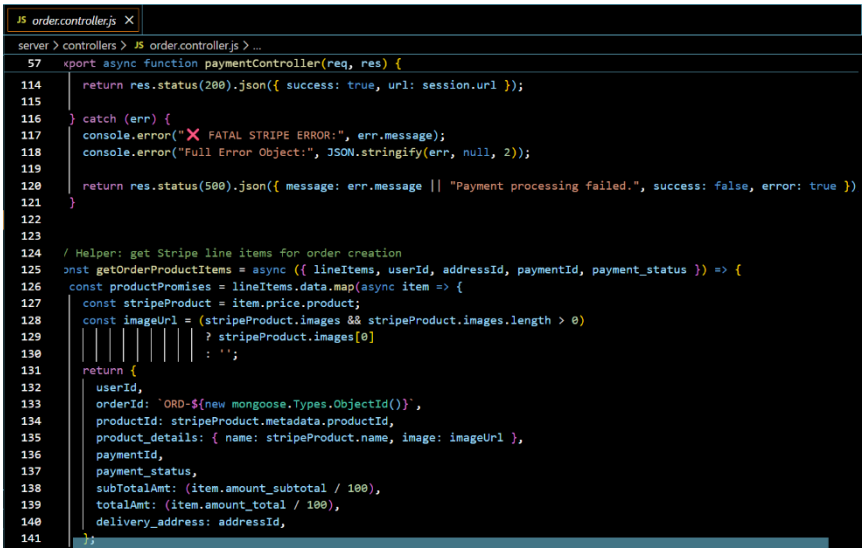
```
// --- Stripe Online Payment Controller ---
export async function paymentController(req, res) {
  const DUMMY_IMAGE = 'https://dummyimage.com/600x400/000/fff&text=No+Image';

  try {
    const userId = req.userId;
    const { list_items, addressId } = req.body;

    const user = await UserModel.findById(userId);
    if (!user?.email) {
      return res.status(400).json({ message: "User email not found", success: false, error: true })
    }

    const line_items = list_items.map(item => {
      if (!item.productId) {
        console.error("❌ Cart item missing product data:", item);
        throw new Error("Missing product details for an item in the cart.");
      }

      const product = item.productId;
      const finalPriceInRupees = priceWithDiscount(product.price, product.discount);
      const calculatedPriceInPaise = Math.max(1, Math.round(finalPriceInRupees * 100));

      let imageUrls = [];
      if (product.image) {
        const imagesArray = Array.isArray(product.image) ? product.image : [product.image];
        imageUrls = imagesArray
          .filter(img => img)
          .map(img => img.startsWith('http://') ? img.replace('http://', 'https://') : img);
```

order.controller.js

server > controllers > order.controller.js > ...

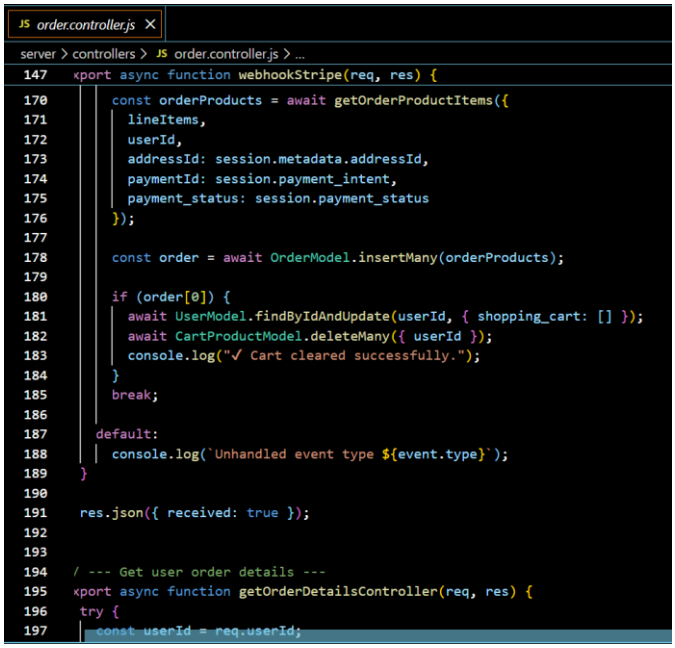
```
export async function paymentController(req, res) {
    const line_items = list_items.map(item => {
      const finalImages = imageUrls.length > 0 ? imageUrls : [DUMMY_IMAGE];

      return {
        price_data: {
          currency: "inr",
          product_data: {
            name: product.name,
            images: finalImages,
            metadata: { productId: product._id.toString() }
          },
          unit_amount: calculatedPriceInPaise
        },
        adjustable_quantity: { enabled: true, minimum: 1 },
        quantity: item.quantity
      }
    });

    const session = await Stripe.checkout.sessions.create({
      submit_type: "pay",
      mode: "payment",
      payment_method_types: ["card"],
      customer_email: user.email,
      metadata: { userId, addressId },
      line_items,
      success_url: `${process.env.FRONTEND_URL}/success`,
      cancel_url: `${process.env.FRONTEND_URL}/cancel`
    });
```

`



```
server > controllers > order.controller.js > ...
57  xport async function paymentController(req, res) {
114     return res.status(200).json({ success: true, url: session.url });
115
116   } catch (err) {
117     console.error("❌ FATAL STRIPE ERROR:", err.message);
118     console.error("Full Error Object:", JSON.stringify(err, null, 2));
119
120     return res.status(500).json({ message: err.message || "Payment processing failed.", success: false, error: true })
121   }
122
123
124  / Helper: get Stripe line items for order creation
125  onst getOrderProductItems = async ({ lineItems, userId, addressId, paymentId, payment_status }) => {
126    const productPromises = lineItems.data.map(async item => {
127      const stripeProduct = item.price.product;
128      const imageUrl = (stripeProduct.images && stripeProduct.images.length > 0)
129                       ? stripeProduct.images[0]
130                       : '';
131      return {
132        userId,
133        orderId: `ORD-${new mongoose.Types.ObjectId()}`,
134        productId: stripeProduct.metadata.productId,
135        product_details: { name: stripeProduct.name, image: imageUrl },
136        paymentId,
137        payment_status,
138        subTotalAmt: (item.amount_subtotal / 100),
139        totalAmt: (item.amount_total / 100),
140        delivery_address: addressId,
141      };
```

```
server > controllers > order.controller.js > ...
125  onst getOrderProductItems = async ({ lineItems, userId, addressId, paymentId, payment_status }) => {
126    const productPromises = lineItems.data.map(async item => {
142    });
143    return Promise.all(productPromises);
144  ;
145
146  / --- Stripe Webhook ---
147  xport async function webhookStripe(req, res) {
148    const sig = req.headers['stripe-signature'];
149    const endpointSecret = process.env.STRIPE_WEBHOOK_SECRET_KEY;
150
151    if (!endpointSecret || typeof endpointSecret !== 'string') {
152      console.error("❌ STRIPE_WEBHOOK_SECRET_KEY not configured.");
153      return res.status(400).send('Webhook secret not configured.');
154    }
155
156    let event;
157    try {
158      event = Stripe.webhooks.constructEvent(req.body, sig, endpointSecret);
159    } catch (err) {
160      console.log(`❌ Webhook Signature Error: ${err.message}`);
161      return res.status(400).send(`Webhook Error: ${err.message}`);
162    }
163
164    switch (event.type) {
165      case "checkout.session.completed":
166        const session = event.data.object;
167        const lineItems = await Stripe.checkout.sessions.listLineItems(session.id, { expand: ['data.price.product'] });
168        const userId = session.metadata.userId;
```



```
server > controllers > order.controller.js > ...
147  xport async function webhookStripe(req, res) {
170      const orderProducts = await getOrderProductItems({
171        lineItems,
172        userId,
173        addressId: session.metadata.addressId,
174        paymentId: session.payment_intent,
175        payment_status: session.payment_status
176      });
177
178      const order = await OrderModel.insertMany(orderProducts);
179
180      if (order[0]) {
181        await UserModel.findByIdAndUpdate(userId, { shopping_cart: [] });
182        await CartProductModel.deleteMany({ userId });
183        console.log("✓ Cart cleared successfully.");
184      }
185      break;
186
187    default:
188      console.log(`Unhandled event type ${event.type}`);
189  }
190
191  res.json({ received: true });
192
193
194  / --- Get user order details ---
195  xport async function getOrderDetailsController(req, res) {
196    try {
197      const userId = req.userId;
```

```
server > controllers > order.controller.js > ...
147  xport async function webhookStripe(req, res) {
189    }
190
191    res.json({ received: true });
192
193
194  / --- Get user order details ---
195  xport async function getOrderDetailsController(req, res) {
196    try {
197      const userId = req.userId;
198      const orderList = await OrderModel.find({ userId }).sort({ createdAt: -1 }).populate("delivery_address");
199
200      return res.json({
201        message: "Order list fetched successfully",
202        data: orderList,
203        success: true,
204        error: false
205      });
206    } catch (err) {
207      return res.status(500).json({ message: err.message || err, success: false, error: true });
208    }
209
210
```

Figure 4.20: Cart and Order Controllers

`

4.4.4 Payment Gateway Code

- **stripe.js** – Stripe configuration and checkout session creation
- **Webhook Logic (order.controller.js)** – Payment verification and order confirmation

```
stripe.js
server > config > stripe.js > ...
1  import Stripe from "stripe";
2
3  const stripeClient = new Stripe(process.env.STRIPE_SECRET_KEY);
4
5  export default stripeClient;
6
```

Figure 4.21: Stripe Checkout

4.4.5 Frontend State Management Code

- **store.js** – Redux store configuration
- **userSlice.js** – User state management
- **cartProduct.js** – Cart state management

```
store.js
client > src > store > store.js > store > reducer > orders
1  import { configureStore } from '@reduxjs/toolkit'
2  import userReducer from './userSlice'
3  import productReducer from './productSlice'
4  import cartReducer from './cartProduct'
5  import addressReducer from './addressSlice'
6  import orderReducer from './orderSlice'
7
8  export const store = configureStore({
9    reducer: {
10     user : userReducer,
11     product : productReducer,
12     cartItem : cartReducer,
13     addresses : addressReducer,
14     orders : orderReducer
15   },
16 })
```

`

```
userSlice.js
client > src > store > userSlice.js > default
import { createSlice } from "@reduxjs/toolkit";
const initialValue = {
    _id : "",
    name : "",
    email : "",
    avatar : "",
    mobile : "",
    verify_email : "",
    last_login_date : "",
    status : "",
    address_details : [],
    shopping_cart : [],
    orderHistory : [],
    role : "",
}
const userSlice  = createSlice({
    name : 'user',
    initialState : initialValue,
    reducers : {
        setUserDetails : (state,action) =>{
            state._id = action.payload?._id
            state.name  = action.payload?.name
            state.email = action.payload?.email
            state.avatar = action.payload?.avatar
            state.mobile = action.payload?.mobile
            state.verify_email = action.payload?.verify_email
            state.last_login_date = action.payload?.last_login_date
            state.status = action.payload?.status
            state.address_details = action.payload?.address_details
```

```
userSlice.js
client > src > store > userSlice.js > default
const userSlice  = createSlice({
    reducers : {
        setUserDetails : (state,action) =>{
            state.shopping_cart = action.payload?.shopping_cart
            state.orderHistory = action.payload?.orderHistory
            state.role = action.payload?.role
        },
        updatedAvatar : (state,action)=>{
            state.avatar = action.payload
        },
        logout : (state,action)=>{
            state._id = ""
            state.name  = ""
            state.email = ""
            state.avatar = ""
            state.mobile = ""
            state.verify_email = ""
            state.last_login_date = ""
            state.status = ""
            state.address_details = []
            state.shopping_cart = []
            state.orderHistory = []
            state.role = ""
        },
    }
})
export const { setUserDetails, logout ,updatedAvatar} = userSlice.actions
export default userSlice.reducer
```

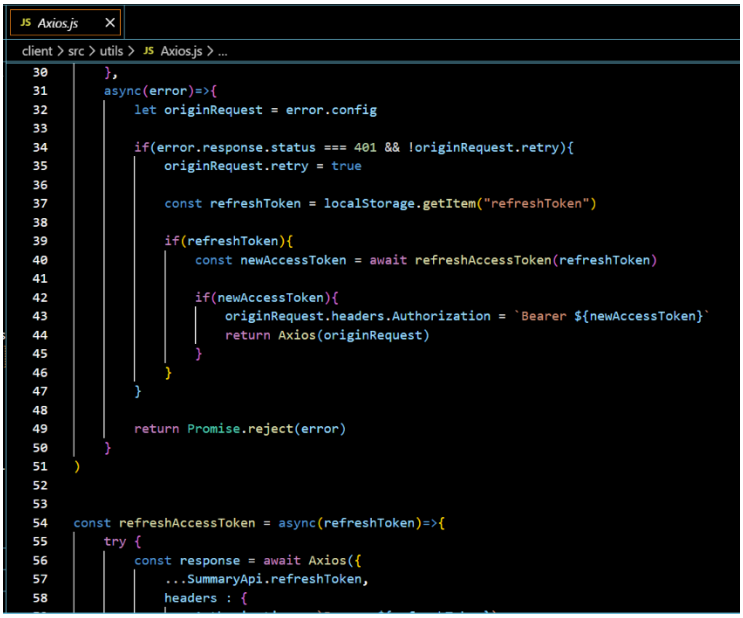
```
cartProduct.js
client > src > store > cartProduct.js > default
import { createSlice } from "@reduxjs/toolkit";

const initialState = {
    cart : []
}

const cartSlice = createSlice({
    name : "cartItem",
    initialState : initialState,
    reducers : {
        handleAddItemCart : (state,action)=>{
            state.cart = [...action.payload]
        },
    }
})

export const { handleAddItemCart } = cartSlice.actions

export default cartSlice.reducer
```

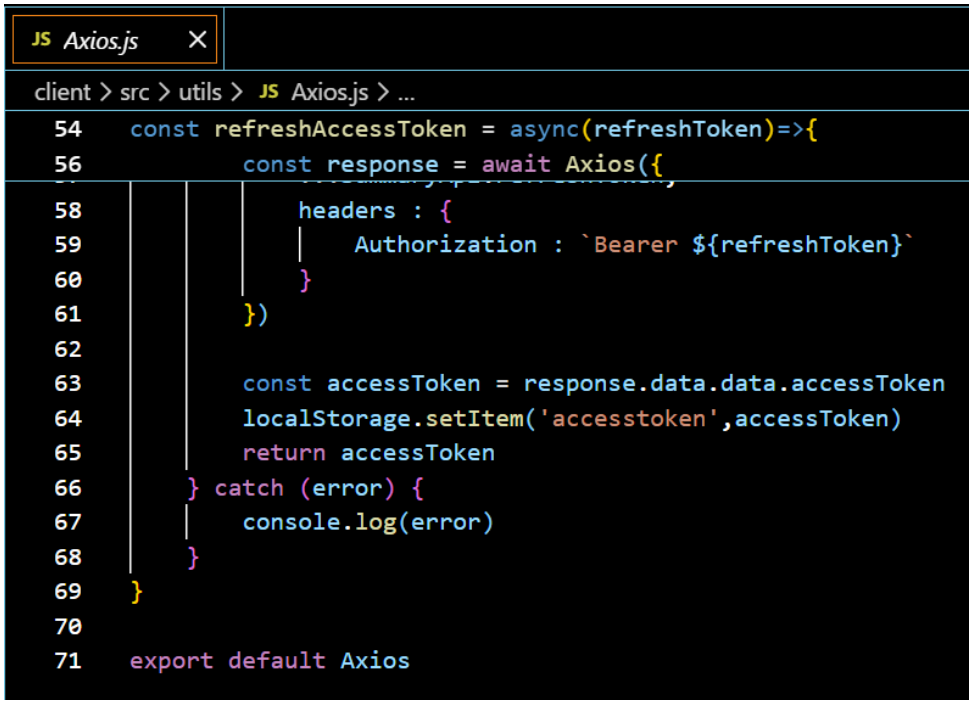
Figure 4.22: Redux Store and Slice Implementation

`

4.4.6 Utility Code

- **axios.js** – API request configuration
- **addToCartProduct.js** – Client-side cart handling logic





`

addToCartProduct.js

client > src > utils > addToCartProduct.js > addToCartProduct > response

```
1  import toast from "react-hot-toast"
2  import SummaryApi from "../common/SummaryApi"
3  import Axios from "./Axios"
4  import AxiosToastError from "./AxiosToastError"
5
6  export const addToCartProduct = async(productId,qty)=>{
7      try {
8          const response = await Axios({
9              ...SummaryApi.addToCart,
10             data : {
11                 quantity : qty,
12                 productId : productId
13             }
14         })
15
16         const { data : responseData} = response
17
18         console.log(responseData)
19         if(responseData.success){
20             toast.success(responseData.message)
21         }
22         return responseData
23
24     } catch (error) {
25         AxiosToastError(error)
26
27         return {}
28     }
29 }
```

addToCartProduct.js

client > src > utils > addToCartProduct.js > addToCartProduct > resp

```
30
31 export const getCartItems = async()=>{
32     try {
33         const response = await Axios({
34             ...SummaryApi.getCartItems
35         })
36
37         const { data : responseData } = response
38
39         if(responseData.success){
40             return responseData
41         }
42     } catch (error) {
43         AxiosToastError(error)
44         return error
45     }
46 }
```

Figure 4.23: Frontend Utility Functions

`

CHAPTER – 5

TESTING AND MAINTENANCE

This chapter describes the testing strategies adopted for the "**Stock & Shop**" E-Commerce System, the test cases executed to verify system functionality, system evaluation, future enhancements, cost–benefit analysis, and a brief user/operational manual.

Testing Methodology Adopted

The **Black Box Testing** methodology was primarily used to test the system. This approach focuses on validating the functionality of the application without considering the internal code structure. Each module was tested based on expected inputs and outputs to ensure correctness, reliability, and usability.

In addition, integration testing was carried out to ensure seamless communication between frontend, backend, database, and third-party services such as Stripe.

`

Test Cases and Expected Results

Test Case ID	Module	Input	Expected Result	Status
TC-01	User Registration	Valid user details	User successfully registered	Pass
TC-02	User Login	Valid credentials	Login successful with JWT token	Pass
TC-03	Forgot Password	Registered email	OTP sent to email	Pass
TC-04	Category Creation	Valid category details	Category added successfully	Pass
TC-05	Product Upload	Valid product data	Product stored in database	Pass
TC-06	Add to Cart	Selected product	Product added to cart	Pass
TC-07	Checkout	Valid cart items	Redirected to Stripe Checkout	Pass
TC-08	Payment Success	Valid card details	Order created and cart cleared	Pass

`

System Evaluation

The "Stock & Shop" system was evaluated based on performance, security, scalability, and usability.

- **Performance:** The MERN stack ensures fast API responses and dynamic UI rendering.
- **Security:** JWT authentication, bcrypt password hashing, and Stripe's PCI-compliant payment processing provide strong security.
- **Scalability:** MongoDB and modular backend architecture allow the system to scale with increased users and products.
- **Usability:** The user-friendly interface makes navigation simple for both Admin and Customers.

Overall, the system meets all the defined objectives efficiently.

Future Scope

The system can be further enhanced with the following features:

- Deployment on cloud platforms (AWS / Vercel)
- Mobile application integration
- Advanced analytics for admin dashboard
- Product recommendations using AI
- Multiple payment gateway support
- Real-time order tracking

`

Cost and Benefit Analysis

Cost Factors

- Development time and effort
- Internet and hardware resources
- Stripe transaction fees (only on successful payments)

Benefits

- 24/7 online availability
- Reduced manual inventory errors
- Secure digital payments
- Improved customer experience
- Efficient order management

The benefits significantly outweigh the costs, making the system economically viable.

User / Operational Manual

For Admin

1. Login using admin credentials.
2. Add categories and sub-categories.
3. Upload and manage products.
4. Monitor orders and inventory levels.
5. View saved customer addresses.

`

For Customer

1. Register and login.
2. Browse products by category.
3. Add products to cart.
4. Complete checkout using Stripe.
5. View order history and saved addresses.

`

CONCLUSION

The "**Stock & Shop**" E-Commerce System successfully demonstrates the design and implementation of a full-stack web application using the MERN stack. The project fulfills its primary objective of providing a secure, scalable, and user-friendly platform for online grocery shopping while supporting efficient inventory and order management for administrators.

The system integrates essential features such as role-based authentication, real-time inventory control, cart and order management, and secure online payments using the Stripe payment gateway. The implementation of a secure webhook mechanism ensures reliable order creation and cart clearance only after verified payment confirmation, reflecting real-world e-commerce practices.

Through this project, practical experience was gained in full-stack development, RESTful API design, database modeling, frontend state management using Redux, and third-party service integration. Overall, the project meets all defined requirements and serves as a strong foundation for future enhancements such as cloud deployment, mobile application support, and advanced analytics.

`

REFERENCES

The development of the "Stock & Shop" E-Commerce System leveraged official documentation and online resources related to its core full-stack technologies, frontend framework, backend runtime environment, database management system, and payment gateway integration. The primary references consulted include:

1. MongoDB Documentation. (n.d.). MongoDB Manual.
   Retrieved from https://www.mongodb.com/docs/
   (Accessed: December 16, 2025).
2. Express.js Documentation. (n.d.). Express.js Official Documentation.
   Retrieved from https://expressjs.com/
   (Accessed: December 16, 2025).
3. React.js Documentation. (n.d.). React – A JavaScript Library for Building User Interfaces.
   Retrieved from https://react.dev/
   (Accessed: December 16, 2025).
4. Node.js Documentation. (n.d.). Node.js Official Documentation.
   Retrieved from https://nodejs.org/en/docs/
   (Accessed: December 16, 2025).
5. Stripe Documentation. (n.d.). Stripe API Documentation.
   Retrieved from https://stripe.com/docs
   (Accessed: December 16, 2025).

`

6. JSON Web Token Documentation. (n.d.). Introduction to JSON Web Tokens.

   Retrieved from https://jwt.io/introduction

   (Accessed: December 16, 2025).

7. Redux Toolkit Documentation. (n.d.). Redux Toolkit Official Documentation.

   Retrieved from https://redux-toolkit.js.org/

   (Accessed: December 16, 2025).

8. *Nodemailer Documentation. (n.d.). Nodemailer Usage and Configuration Guide.*

   Retrieved from https://nodemailer.com/about/

   (Accessed: December 16, 2025).

`

APPENDICES

APPENDIX A

INSTALLATION AND EXECUTION PROCEDURE

The following steps describe the procedure to install and execute the "Stock & Shop" E-Commerce System on a local machine for evaluation and demonstration purposes.

Software Requirements

- Node.js (v16 or above)
- MongoDB
- Web Browser (Google Chrome / Microsoft Edge)
- Stripe CLI

Backend Setup

1. Navigate to the server folder.
2. Install required dependencies:

```
npm install
```

3. Create a .env file and configure the following:
    - MongoDB connection string
    - JWT secret key
    - Stripe secret key

`

4. Start the backend server:

```
npm run dev
```

The backend server runs on localhost:8080.

Frontend Setup

1. Navigate to the **client** folder.
2. Install required dependencies:

```
npm install
```

3. Start the frontend application:

```
npm run dev
```

Stripe Webhook Setup

To ensure secure and reliable payment confirmation during local development, **Stripe CLI** is used for webhook event forwarding.

1. Login to Stripe CLI:

```
stripe login
```

2. Forward webhook events to the backend server:

```
stripe listen --forward-to localhost:8080/api/webhook/stripe
```

This step is mandatory for successful order creation and cart clearance after payment.

Database Setup

MongoDB is used as the database to store user, product, cart, and order data. Database collections are created automatically when the application is executed.

`

APPENDIX B

GITHUB REPOSITORY

The complete source code for the "Stock & Shop" E-Commerce System is maintained using GitHub for version control and project management.

- **Repository Name**: *Stock & Shop – MERN E-Commerce System*
- **GitHub Link**:

  https://github.com/Saanvi-Rao/stock-and-shop-mern-ecommerce

The repository follows a modular structure with separate frontend and backend folders, including controllers, routes, database schemas, Redux state management files, and utility functions required for system execution.

APPENDIX C

SAMPLE LOGIN CREDENTIALS AND SYSTEM ACCESS

The following login credentials and access instructions are provided strictly for evaluation and demonstration purposes.

Administrator Access

Faculty members can use the following credentials to access the Admin Panel and evaluate administrative functionalities such as category management, product upload, inventory control, and order monitoring.

- **Admin Email:** admin@stockshop.com
- **Admin Password:** Admin@123

`

Upon successful login, the system automatically redirects the administrator to the Admin Dashboard.

Customer (User) Access

To evaluate the Customer Panel, faculty members may register as a new user using their own email address.

Procedure:

1. Navigate to the Register page.
2. Enter name, email, and password.
3. Complete registration.
4. If the Forgot Password feature is tested, an OTP will be sent to the registered email address.
5. The OTP can be used to reset the password and continue further operations.

This ensures secure verification and demonstrates the OTP-based authentication flow implemented in the system.

Note:

The provided credentials and user registration flow are intended only for project evaluation and do not represent real or permanent user data.