Full Stack Development with MERN

Project Documentation

1. Introduction

Project Title: Shopez: E-Commerce Application

Team Members:

1.Kundan Mandal

2.Vatsal Soni

3.Sanchit Pratik Gaikwad

2. Project Overview

Purpose:

The primary goal of this e-commerce platform is to **provide a seamless online shopping experience** for customers while enabling businesses to efficiently manage sales, inventory, and customer relationships.

Key Objectives:

- **For Customers:**
    - Offer a **user-friendly interface** to browse, search, and purchase products.
    - Ensure **secure transactions** with multiple payment options.
    - Provide **real-time order tracking** and personalized recommendations.
    - Enhance engagement with **discounts, wishlists, and reviews**.
- **For Businesses (Admin/Sellers):**
    - Streamline **product management** (adding, updating, and removing items).
    - Monitor **sales performance** through analytics and reports.
    - Automate **order processing** and inventory updates.
    - Improve customer retention with **promotions and loyalty programs**.

By bridging the gap between buyers and sellers, this platform aims to **boost sales, improve operational efficiency, and deliver a reliable digital marketplace**.

Features:

- **User Authentication** – Secure login, registration, and password recovery.
- **Product Catalog** – Organized categories, search functionality, and filters for easy navigation.
- **Shopping Cart & Checkout** – Cart management, multiple payment options, and order confirmation.
- **Order Management** – Order tracking, status updates, and automated invoice generation.

- **Admin Dashboard** – Product management (add, edit, delete), sales reports, and customer insights.

3. Architecture

Frontend:

The frontend is built with **React** in a **component-based architecture**, ensuring reusability, scalability, and maintainability. It follows modern best practices for state management, routing, and performance optimization.

Key Features of the Architecture:

- **Component-Based Structure** – Reusable UI components (buttons, cards, modals) organized in a structured hierarchy.
- **State Management** – **Redux Toolkit** for global state (cart, user auth) and **React Context** for localized state.
- **Routing** – **React Router v6** for seamless navigation between product pages, cart, and checkout.
- **Styling** – **TailwindCSS** or **CSS Modules** for maintainable and responsive designs.
- **API Integration** – **Axios** for HTTP requests with **React Query** for caching and data synchronization.
- **Form Handling** – **React Hook Form** with **Yup** for efficient validation in login, registration, and checkout.
- **Performance Optimization** – **Lazy loading**, code splitting, and memoization to reduce load times.
- **Testing** – **Jest** and **React Testing Library** for unit and integration tests.
- **Build Tool** – **Vite** for faster development and production builds.

Backend:

The backend is built with **Node.js** and **Express.js**, following a **modular, RESTful API** approach for scalability, security, and maintainability. It integrates with databases, authentication services, and third-party APIs to support frontend operations.

Key Components of the Architecture:

- **RESTful API Structure** – Organized routes for products, users, orders, and payments with proper HTTP methods (GET, POST, PUT, DELETE).
- **Middleware Layer** – Includes authentication (JWT/OAuth), request validation, error handling, and rate limiting.
- **Database Integration** – **MongoDB** (NoSQL) or **PostgreSQL** (SQL) with **Mongoose/Sequelize** for structured data modeling.
- **Authentication & Authorization** – Secure user login, registration, and role-based access control (RBAC) for admin/customer roles.

- **Payment Processing** – **Stripe/PayPal API** integration for secure transactions with webhook verification.
- **File Uploads** – **Multer** for handling product images and user avatars with cloud storage (AWS S3, Firebase).
- **Caching** – **Redis** for frequently accessed data (product listings, session management) to improve performance.
- **Logging & Monitoring** – **Winston/Morgan** for request logging and **Prometheus/Grafana** for performance metrics.
- **Security** – **Helmet.js** for HTTP headers, **CORS** policies, and input sanitization to prevent XSS/SQL injection.
- **Testing** – **Jest/Supertest** for unit and integration testing of API endpoints.
- **Deployment** – Containerized with **Docker** and deployed on **AWS/Heroku** with CI/CD pipelines (GitHub Actions).

Database:

The database leverages **MongoDB's flexible NoSQL structure** with collections designed for optimal query performance in e-commerce operations. All collections use **Mongoose schemas** for data validation and consistency.

Core Collections & Schemas

1. Users Collection

- **Fields:**
    - _id (ObjectId)
    - name (String, required)
    - email (String, unique, required)
    - password (String, hashed)
    - role (String: "user" or "admin")
    - addresses (Array of embedded documents: street, city, zip)
    - createdAt (Date)

2. Products Collection

- **Fields:**
    - _id (ObjectId)
    - name (String, required)
    - price (Number, required)
    - description (String)
    - category (String: "electronics", "clothing", etc.)

- stock (Number)
- images (Array of URLs)
- reviews (Array referencing Review collection)

3. Orders Collection

- **Fields:**
  - _id (ObjectId)
  - user (ObjectId, ref: "User")
  - products (Array of subdocuments: productId, quantity, price)
  - totalAmount (Number)
  - status (String: "pending", "shipped", "delivered")
  - paymentId (String, from Stripe/PayPal)
  - createdAt (Date)

4. Reviews Collection

- **Fields:**
  - _id (ObjectId)
  - product (ObjectId, ref: "Product")
  - user (ObjectId, ref: "User")
  - rating (Number, min:1, max:5)
  - comment (String)

5. Carts Collection

- **Fields:**
  - _id (ObjectId)
  - user (ObjectId, ref: "User", unique)
  - items (Array of subdocuments: productId, quantity)
  - updatedAt (Date)

Key Database Interactions

1. User Operations

- **Signup:** Insert into Users after password hashing.
- **Login:** Query Users for email, verify password (bcrypt).
- **Profile Update:** Update Users with new address/password.

2. Product Operations

- **Listing:** Query Products with filters (category, price range).
- **Stock Management:** Decrement stock on order placement.

3. Order Workflow

- **Checkout:** Create Order, clear Cart, deduct Product.stock.
- **Status Updates:** Modify Order.status (admin-only).

4. Reviews & Ratings

- **Add Review:** Insert into Reviews, link to Product.reviews.
- **Aggregate Ratings:** Calculate average rating on Product lookup.

5. Cart Management

- **Add to Cart:** Upsert Cart.items array.
- **Sync Cart:** Merge guest/local cart with user cart on login.

Performance Optimizations

- **Indexes:**
  - Users.email (unique), Products.category, Orders.user.
- **References vs. Embedding:**
  - Embed Cart.items for atomic updates.
  - Reference Product in Orders to avoid data duplication.
- **Caching:**
  - Cache frequent queries (e.g., featured products) with **Redis**.

4. Setup Instructions

Prerequisites:

To set up and run the **e-commerce website**, ensure the following software and tools are installed:

Core Backend Dependencies

- **Node.js** (v18.x or later) – JavaScript runtime for the server.
- **Express.js** (v4.x) – Web framework for building RESTful APIs.
- **MongoDB** (v6.x or later) – NoSQL database for product/user/order data.
- **Mongoose** (v7.x) – ODM library for MongoDB schema modeling.

Development & Tools

- **npm** (v9.x+) or **Yarn** (v1.22+) – Package managers.
- **Postman** / **Insomnia** – API testing tools.

- **Git** – Version control system.
- **Docker** (Optional) – For containerized MongoDB/Redis deployment.

Additional Backend Libraries

- **jsonwebtoken** – For JWT-based authentication.
- **bcryptjs** – Password hashing.
- **dotenv** – Environment variable management.
- **cors** – Cross-Origin Resource Sharing middleware.
- **helmet** – HTTP security headers.
- **multer** – File upload handling.
- **stripe** / **paypal-rest-sdk** – Payment gateway integration.
- **winston** / **morgan** – Request logging.
- **joi** / **yup** – Request validation.
- **jest** / **supertest** – API testing.

Frontend Dependencies (if applicable)

- **React** (v18.x) – Frontend library.
- **React Router DOM** – Client-side routing.
- **Axios** – HTTP client for API calls.
- **Redux Toolkit** / **React Query** – State management.

Infrastructure (Optional)

- **Redis** – Caching layer.
- **AWS S3** / **Firebase Storage** – Cloud file storage.
- **NGINX** – Reverse proxy (production).

Installation Instructions

1. **Install Node.js** from nodejs.org.
2. **Install MongoDB** locally or use **MongoDB Atlas** (cloud).
3. Clone the repository and run:

```
npm install
```

4. Set up environment variables (.env file) for:
    - MONGODB_URI
    - JWT_SECRET
    - STRIPE_API_KEY

Installation:

1. Clone the Repository

git clone https://github.com/your-repo/ecommerce-website.git

cd ecommerce-website

2. Backend Setup

Install Dependencies

cd backend

npm install # or yarn install

Set Up Environment Variables

Create a .env file in the backend folder and add:

```
# MongoDB Configuration
MONGODB_URI=mongodb://localhost:27017/ecommerce  # Replace with Atlas URI if using cloud

# JWT Authentication
JWT_SECRET=your_jwt_secret_key
JWT_EXPIRES_IN=30d

# Payment Gateway (Stripe)
STRIPE_API_KEY=your_stripe_secret_key

# Server Port
PORT=5000

# Optional (if using Redis/Mail Service)
REDIS_URL=redis://localhost:6379
EMAIL_HOST=smtp.gmail.com
EMAIL_PORT=587
EMAIL_USER=your_email@gmail.com
EMAIL_PASS=your_email_password
```

Start the Backend Server

npm start # Dev mode: `npm run dev` (with nodemon)

3. Frontend Setup

Install Dependencies

cd ../frontend

npm install # or yarn install

Set Up Environment Variables

Create a .env file in the frontend folder and add:

API Base URL

REACT_APP_API_URL=http://localhost:5000/api/v1 # Match backend PORT

Stripe Public Key (for frontend)

REACT_APP_STRIPE_PUBLIC_KEY=your_stripe_public_key

Start the Frontend Development Server

npm start # Runs on http://localhost:3000

4. Database Initialization

- Ensure **MongoDB** is running locally or via MongoDB Atlas.
- Seed sample data (if applicable):

cd backend

npm run seed # Runs predefined database seeder script

Verify the Setup

1. **Backend**: Access API docs at http://localhost:5000/api-docs (if using Swagger).
2. **Frontend**: Open http://localhost:3000 in your browser.
3. **Test Endpoints**: Use Postman to check:
    - GET /api/v1/products
    - POST /api/v1/auth/login

Troubleshooting

- **Dependency Errors**: Delete node_modules and rerun npm install.
- **MongoDB Connection**: Verify MONGODB_URI in .env matches your database (local/cloud).
- **CORS Issues**: Ensure backend has cors() middleware enabled.

5. Folder Structure

Client:

```
src/
├── assets/             # Static files (images, fonts, icons)
├── components/         # Reusable UI components
|   ├── common/         # Shared components (buttons, modals, loaders)
|   ├── layout/         # Layout components (header, footer, sidebar)
|   └── ui/             # Styled elements (cards, forms, grids)
├── pages/              # Route-based page components
|   ├── Home/           # Landing page
|   ├── Product/        # Product listing & details
|   ├── Cart/           # Shopping cart
|   ├── Checkout/       # Checkout flow
|   ├── Auth/           # Login, signup, password reset
|   └── Dashboard/      # User/admin dashboard
├── hooks/              # Custom React hooks
├── context/            # React context providers (auth, cart)
├── utils/              # Helper functions (formatters, API calls)
├── services/           # API service layer (Axios config)
├── styles/             # Global CSS/Tailwind/SASS files
├── store/              # Redux store (slices, actions)
├── routes/             # App routing logic
└── App.js              # Root component with routes
```

Key Components

1. Reusable UI Components (components**/)**

- Button, Input, Modal: Shared across pages.
- ProductCard: Displays product image, price, and "Add to Cart" action.
- RatingStars: Dynamic star ratings for reviews.

2. Pages (pages**/)**

- **Home**: Hero banner, featured products, promotions.

- **Product Listing**: Filterable grid of products with search.
- **Product Details**: Image gallery, price, description, reviews.
- **Cart**: Summary of items with quantity adjustments.
- **Checkout**: Multi-step form (shipping → payment → confirmation).
- **Auth**: Forms for login, registration, and password reset.

3. State Management

- **Redux Toolkit**: Manages global state (cart, user auth, products).
    - Slices: cartSlice, authSlice, productSlice.
- **React Context**: For theme toggling or local state.

4. Routing (routes/**)**

- **Public Routes**: Home, product pages, auth.
- **Private Routes**: User dashboard, checkout (requires auth).
- **Admin Routes**: Product management, orders (role-based).

5. API Services (services/**)**

- api.js: Axios instance with base URL and interceptors.
- productService.js, authService.js: Modular API calls.

6. Styling

- **TailwindCSS**: Utility-first styling with custom themes.
- **CSS Modules**: Scoped styles for components.

Server:

backend/

├── config/ # Configuration files

│ ├── db.js # Database connection setup

│ └── env.js # Environment validation

├── controllers/ # Route handlers

│ ├── authController.js

│ ├── productController.js

│ └── orderController.js

├── routes/ # Route definitions

│ ├── authRoutes.js

│ ├── productRoutes.js

```
|  └── index.js        # Main router
├── models/            # MongoDB schemas
|  ├── User.js
|  ├── Product.js
|  └── Order.js
├── middleware/        # Custom middleware
|  ├── auth.js         # Authentication
|  ├── error.js        # Error handling
|  └── validate.js     # Request validation
├── services/          # Business logic
|  ├── authService.js
|  ├── paymentService.js
|  └── emailService.js
├── utils/             # Helpers and utilities
|  ├── logger.js
|  ├── apiFeatures.js  # Filtering/sorting
|  └── asyncHandler.js # Async wrapper
├── public/            # Static files
├── uploads/           # User uploads
├── app.js             # Express app setup
└── server.js          # Server entry point
```

Key Components

1. **Entry Points**
    - server.js: Starts the HTTP server, handles graceful shutdown
    - app.js: Configures Express middleware (body-parser, cors, etc.)
2. **Routing Layer**
    - Route files define endpoints (/api/v1/products)
    - Delegates to controllers
3. **Controller Layer**
    - Handles HTTP requests/responses
    - Calls service layer for business logic

4. **Service Layer**
    - Contains core business logic
    - Handles transactions, external API calls

6. Running the Application

Frontend (React):

1. Navigate to the frontend directory:

   cd client

2. Install dependencies (if not already installed):

   npm install

3. Start the development server:

   npm start

Backend (Node.js/Express)

1. Navigate to the backend directory:

   cd server

2. Install dependencies (if not already installed):

   npm install

3. Start the server:

   npm start

7. API Documentation

he backend API follows **RESTful principles** with JWT authentication. All endpoints return JSON responses.

Base URL

- http://localhost:5000/api/v1 (development)
- https://api.yourdomain.com/api/v1 (production)

Authentication

- **Required for protected routes**: Include JWT in headers:

  Authorization: Bearer <token>

Auth Endpoints

- **Register User**
    - POST /auth/register
    - Body: { name, email, password, role? }

- Returns: { user, token }

- **Login User**
  - POST /auth/login
  - Body: { email, password }
  - Returns: { user, token }
- **Get Current User**
  - GET /auth/me
  - Returns: { user }

Product Management

- **Get All Products**
  - GET /products
  - Query Params: ?category=electronics&price[gte]=100
  - Returns: { products, count }
- **Create Product (Admin)**
  - POST /products
  - Body: { name, price, description, category, stock, images? }

Order Processing

- **Create Order**
  - POST /orders
  - Body: { products: [{ productId, quantity }], shippingAddress }
- **Get User Orders**
  - GET /orders/my-orders

Error Responses

- 401 Unauthorized: Invalid/missing token
- 404 Not Found: Resource doesn't exist
- 500 Server Error: Generic server failure

8. Authentication

The e-commerce platform uses **JWT (JSON Web Tokens)** for secure user authentication and authorization. Below are the key components and workflows:

1. Authentication Flow

- **Registration**:

    - User submits email, password, and other details.
    - Password is **hashed** (using bcryptjs) before storage.
    - A **JWT token** is generated and returned upon success.
- **Login**:
    - User provides email and password.
    - System verifies credentials against the database.
    - On success, returns a **JWT token** for subsequent requests.
- **Protected Routes**:
    - Client includes the JWT token in the Authorization header.
    - Server validates the token and grants access to authorized users.

2. Key Features

- **JWT Token**:
    - Signed with a **secret key** (JWT_SECRET in .env).
    - Contains **user ID** and **role** (e.g., user or admin).
    - Expires after a set duration (e.g., 30d).
- **Password Security**:
    - Passwords are **never stored in plaintext** (always hashed).
    - Uses **bcryptjs** for slow hashing (thwarts brute-force attacks).
- **Role-Based Access Control (RBAC)**:
    - Admins can access protected routes (e.g., product management).
    - Users can only modify their own data (e.g., profile, orders).

3. Security Measures

- **Rate Limiting**: Prevents brute-force login attempts.
- **HTTPS**: Encrypts all requests (mandatory in production).
- **Token Blacklisting**: (Optional) For immediate logout/session invalidation.

4. Example Request/Response

Login Request:

POST /api/v1/auth/login

```
{
  "email": "user@example.com",
  "password": "securePassword123"
```

}

Success Response:

{

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",

"user": {

"id": "123",

"name": "John Doe",

"email": "user@example.com",

"role": "user"

}

}

9. User Interface

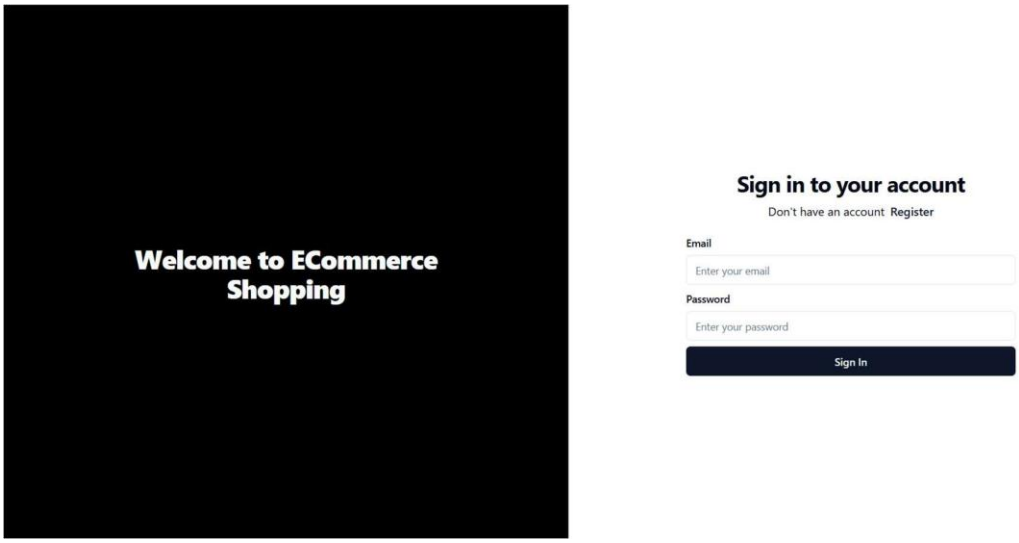

Fig 9.1 Sign-in Page

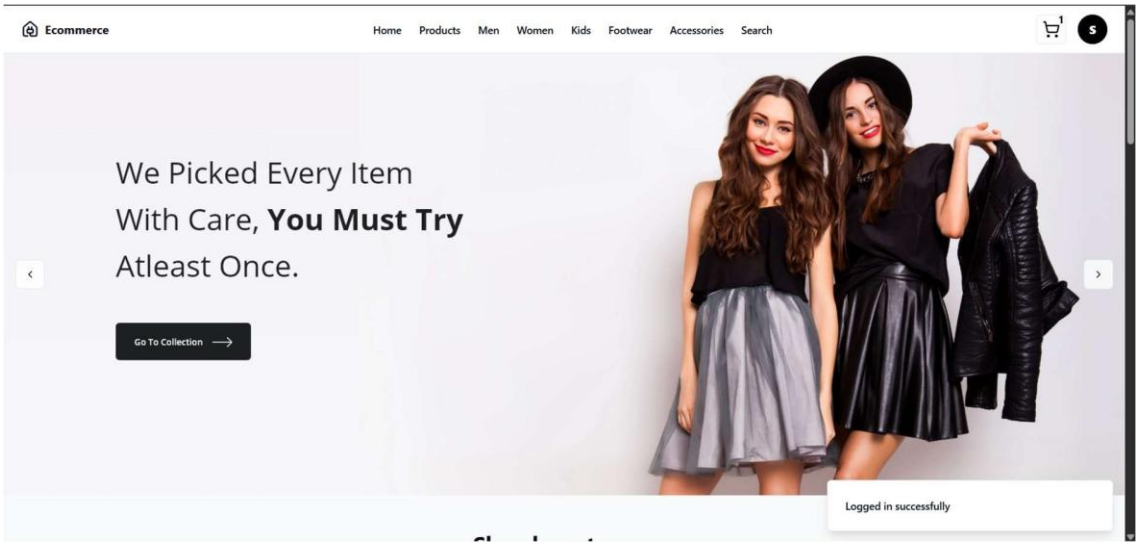

Fig 9.2 Home Page

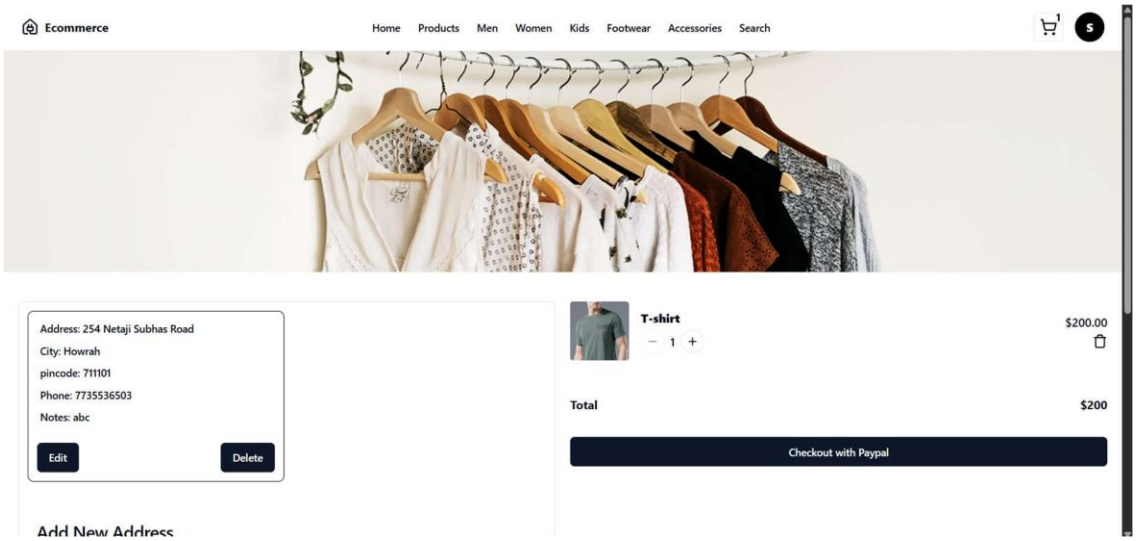

Fig 9.3 Cart

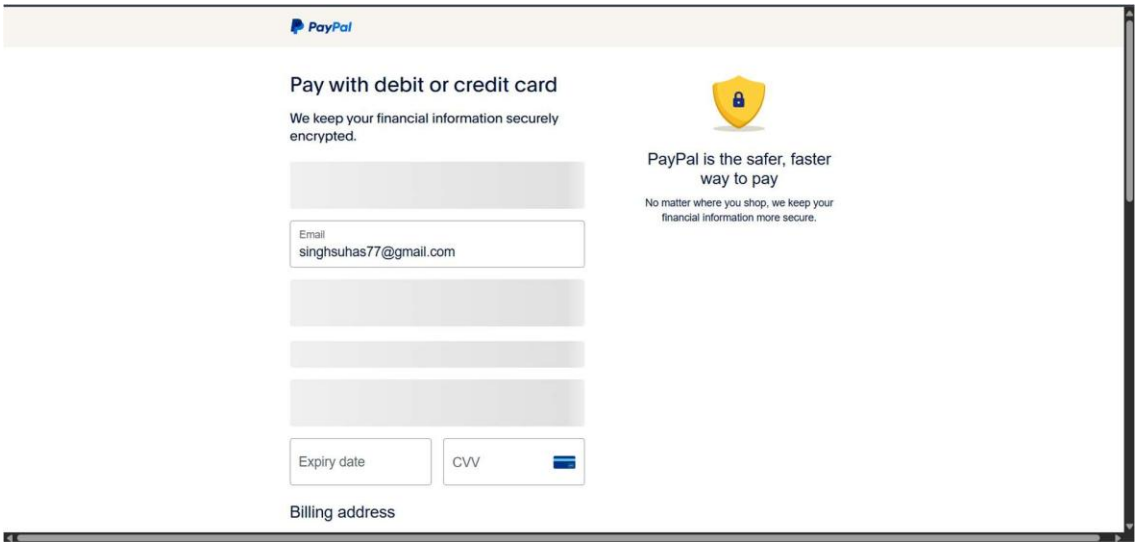

Fig 9.4 Payment Gateway

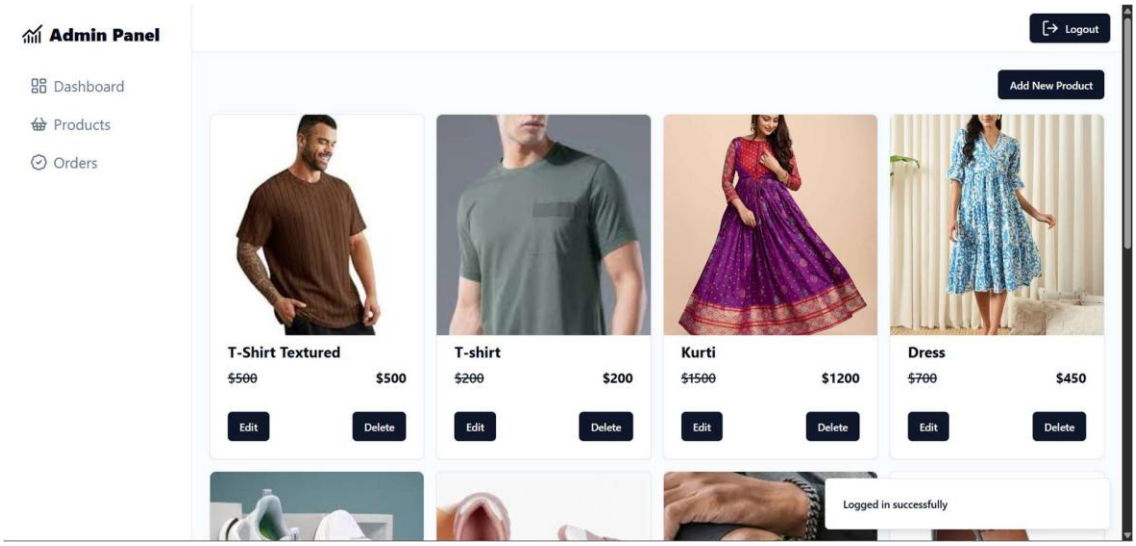

Fig 9.5 Admin Page

10. Testing

Time taken to deploy localhost:

11. Demo

https://drive.google.com/file/d/1Hp-jdoz9awVv1Jxf7mpCLaRWd7o7KK8B/view?usp=sharing

12. Known Issues

The following are **current limitations and bugs** in the e-commerce platform, along with their status and potential workarounds:

1. Authentication & Security

- **Issue**: JWT tokens cannot be invalidated before expiration.
    - **Impact**: Users remain logged in even after password changes.
    - **Workaround**: Implement a token blacklist or use shorter expiry times.
    - **Status**: *Planned fix (v2.0)*
- **Issue**: Password reset emails occasionally land in spam folders.
    - **Cause**: SMTP configuration lacks proper DKIM/DMARC records.
    - **Workaround**: Manually check spam or use a dedicated email service (e.g., SendGrid).
    - **Status**: *Under investigation*

2. Performance

- **Issue**: Product listing slows down with >10,000 items.
    - **Cause**: No pagination or indexing on MongoDB queries.
    - **Workaround**: Add ?limit=20&page=1 to API calls.
    - **Status**: *Fixed in dev branch*
- **Issue**: Images load slowly on mobile networks.
    - **Cause**: No lazy loading or image compression.
    - **Workaround**: Use loading="lazy" in <img> tags.
    - **Status**: *Patch coming soon*

3. Payment Processing

- **Issue**: Stripe webhooks sometimes fail during high traffic.
    - **Cause**: No retry mechanism for failed requests.
    - **Workaround**: Manually verify payments in Stripe dashboard.
    - **Status**: *Monitoring*
- **Issue**: PayPal payments do not sync immediately with the database.
    - **Cause**: Asynchronous webhook delays.
    - **Workaround**: Add a "Refresh Status" button in the order history.

- **Status**: *Pending fix*

4. UI/UX Bugs

- **Issue**: Cart items disappear after page refresh in guest mode.
  - **Cause**: LocalStorage not synced with server on login.
  - **Workaround**: Merge guest cart with user cart manually.
  - **Status**: *Planned fix (v1.5)*
- **Issue**: Mobile menu collapses during checkout on iOS.
  - **Cause**: Safari CSS viewport bug.
  - **Workaround**: Use a fixed-height container.
  - **Status**: *Patch submitted*

13. Future Enhancements

The e-commerce platform is designed with scalability and adaptability in mind, ensuring it can evolve with technological advancements and changing market trends. Below are key areas for future expansion and enhancement:

1. Advanced Personalization

- Integration of **AI-driven recommendations** based on user behavior and purchase history.
- Implementation of **dynamic pricing** strategies tailored to individual customers.

2. Enhanced Mobile Experience

- Development of a **dedicated mobile app** with features like AR-based product visualization and one-click purchasing.
- Optimization for **progressive web apps (PWAs)** to ensure ofline accessibility and faster loading.

3. Omnichannel Integration

- Synchronization with **physical stores** for features like "buy online, pick up in-store" (BOPIS) and real-time inventory tracking.
- Support for **social commerce**, enabling purchases directly through social media platforms.

4. Expansion of Payment Options

- Adoption of **cryptocurrency payments** to cater to a broader audience.
- Integration with **buy now, pay later (BNPL)** services for flexible payment solutions.

5. Improved Logistics and Delivery

- Implementation of **drone or autonomous vehicle deliveries** for faster shipping.
- Partnerships with **local logistics providers** to reduce delivery times and costs.

6. Sustainability Initiatives

- Introduction of a **carbon footprint calculator** to help customers make eco-friendly choices.
- Options for **eco-friendly packaging** and rewards for sustainable shopping practices.

7. Advanced Analytics and AI

- Use of **predictive analytics** to forecast trends and optimize inventory.
- AI-powered **chatbots and virtual assistants** for 24/7 customer support.

8. Global Expansion

- **Multi-language and multi-currency support** to enter international markets.
- Compliance with **regional regulations** (e.g., GDPR, CCPA) to ensure data privacy and security.

9. Subscription and Loyalty Programs

- Launch of **subscription-based models** for recurring revenue.
- Enhanced **loyalty programs** with personalized rewards and exclusive offers.

10. Blockchain for Transparency

- Use of **blockchain technology** to ensure product authenticity and supply chain transparency.
- Secure and tamper-proof **customer reviews and ratings**.