Full Stack Development with MERN

Project Documentation Format

1. INTRODUCTION

1.1. Project Title

ShopEZ: One-Stop Shop for Online Purchases

1.2. Team Members

Bidisha Biswas (Team Lead) – Backend Developer
Bhargavee Singh – Frontend Developer
Diya Raj – Database & Deployment
Namrata Bhutani – Tester

1.3. Team ID: SWTID1742751842

---

2. PROJECT OVERVIEW

2.1. Purpose

The purpose of this project is to design and develop a fully functional e-commerce website that provides users with a seamless online shopping experience. The platform aims to bridge the gap between consumers and sellers by offering a convenient, secure, and user-friendly interface for browsing, selecting, and purchasing products. It is built using the MERN (MongoDB, Express.js, React, Node.js) stack to ensure scalability, real-time performance, and maintainability.

This project is intended to streamline the entire shopping process—from product discovery and cart management to order placement and payment—while ensuring robust user authentication, responsive design, and efficient backend management. The goal is to replicate and enhance the essential features of leading e-commerce platforms, making it adaptable for various product categories and business models.

2.2. Features

1. **User Authentication & Authorization**
   - Secure registration and login system
   - JWT-based authentication
   - Role-based access control (Admin/User)
2. **Product Catalog**
   - Dynamic listing of products with images, descriptions, and prices

    - Filtering and sorting by categories, brands, and price range
    - Product detail page with specifications and reviews
3. **Shopping Cart**
    - Add/remove items to/from cart
    - Quantity management
    - Real-time cart updates and total price calculation
4. **Search Functionality**
    - Keyword-based product search
    - Autocomplete suggestions for faster navigation
5. **Order Management**
    - Checkout flow with order summary
    - Address and payment information collection
    - Order tracking and history
6. **Admin Dashboard**
    - Product management (add, edit, delete products)
    - User management and analytics
    - Order management with delivery status updates
7. **Responsive UI**
    - Fully responsive design for desktops, tablets, and mobile devices
    - Clean and intuitive user interface using modern CSS and design libraries
8. **Secure Payment Integration** *(Optional/Planned Feature)*
    - Integration with payment gateways (e.g., Razorpay, Stripe, etc.)
    - Secure handling of payment transactions
9. **Wishlist & Favorites** *(Optional)*
    - Users can save favorite products for later
10. **Notifications**

- Success and error alerts for user actions
- Optional email notifications for orders

---

3. ARCHITECTURE

3.1. Frontend Architecture (React.js)

The frontend is designed as a **Single Page Application (SPA)** using React.js to ensure seamless user interaction and a dynamic experience. It adheres to a component-based architecture where each UI element is modular and reusable.

Key Features:

- **Component Hierarchy:** The application is broken into reusable components like Header, Footer, ProductCard, LoginForm, Dashboard, etc., following a clean and maintainable structure.

- **Client-Side Routing:** Implemented using react-router-dom, enabling smooth navigation between routes such as /login, /register, /products, /cart, and /admin.
- **State Management:** Redux Toolkit is used to manage global application state. AsyncThunk handles asynchronous operations such as API calls and state updates.
- **Authentication Handling:** JWT tokens (Access and Refresh) are stored securely (e.g., in memory or localStorage) and attached to requests using axios interceptors.
- **Responsive UI:** Designed using Tailwind CSS to ensure responsiveness and consistency across various screen sizes and devices.
- **Error & Notification Handling:** Integrated toast notifications to provide feedback on successful or failed user actions.

3.2. Backend Architecture (Node.js and Express.js)

The backend is developed using Node.js and Express.js, structured in a RESTful API format. It is responsible for user authentication, CRUD operations, order processing, and email notifications.

Key Components:

- **Modular Routing:** Organized routes for users, products, categories, orders, and admin functionalities.
- **Controllers and Middleware:**
  - Controllers handle request logic and validation.
  - Middleware ensures route protection (authentication and authorization), input validation, and error handling.
- **Authentication System:**
  - JWT tokens are used for secure, stateless authentication.
  - Passwords are hashed using bcryptjs before storage.
- **Security Practices:**
  - Middleware protects against unauthorized access.
  - Environment variables are managed securely using .env files.
- **Email Integration:** Nodemailer is used to send password reset emails and order confirmation messages.
- **Deployment-Ready:** Backend is structured to support production deployment, with appropriate logging, error responses, and configuration support.

3.3. Database Architecture (MongoDB)

MongoDB is used as the primary database for ShopEZ, providing a scalable and flexible NoSQL structure. Mongoose ODM (Object Data Modeling) library is used to define schemas and handle data operations.

Primary Collections:

1. **Users**
   - Fields: name, email, hashedPassword, role, address, resetToken, resetTokenExpiry

- Contains authentication and authorization data.

2. **Products**
   - Fields: title, description, price, image, category, stock, colors, createdAt
   - Holds all product-related data and variations.
3. **Orders**
   - Fields: userId, items, totalAmount, shippingAddress, paymentStatus, orderStatus, createdAt
   - Manages order records placed by users.
4. **Categories**
   - Fields: name, description
   - Used for product classification.
5. **Wishlists**
   - Fields: userId, products[]
   - Tracks user-selected products for future purchases.

Database Operations:

- Create, Read, Update, and Delete (CRUD) functionalities for each collection.
- Aggregation pipelines are used for filtering and searching data efficiently.
- Relations (e.g., referencing userId in Orders) are maintained using MongoDB references.
- Indexing is applied on frequently queried fields such as email, product name, and category.

---

4. SETUP INSTRUCTIONS

4.1. Prerequisites

4.1.1. Software Requirements

Tool	Version (Recommended)	Description
Node.js	v16.x or later	JavaScript runtime to run backend and React
npm	Comes with Node.js	Package manager for dependencies
MongoDB	v6.x or Atlas (Cloud)	NoSQL database for storing project data
Git	Latest	To clone the Git repository
VS Code	Latest	Code editor (optional but recommended)

Verification of installations:

node -v

npm -v
mongo --version
git –version

4.2. Installation Steps

To run the ShopEZ e-commerce web application on your local machine, follow these step-by-step instructions. The project uses the **MERN stack** — consisting of **MongoDB**, **Express.js**, **React.js**, and **Node.js** — and includes both a user-facing store and an admin panel for backend management.

Step 1: Clone the Git Repository

- Begin by cloning the project repository from GitHub using the following command:

  git clone <rhttps://github.com/bidisha-biswas0610/SMARTBRIDGE_MERN_Project_ECommerceWebsite_ShopEZ.git>
- Once the repository is cloned, navigate into the main project directory:

  cd E-Commerce Website

Step 2: Backend Setup (Node.js + Express.js)

The backend is responsible for handling API requests, user authentication, order processing, and more.

- Navigate to the backend folder:

  cd Backend
- Install all required backend dependencies:

  npm install
- Create a *.env* file in the *Backend* directory to store environment variables securely:

  touch .env
- Add the following configurations to your *.env* file:

  PORT=5000
  MONGO_URI=mongodb_connection_string
  JWT_SECRET=jwt_secret
  EMAIL_USER=email@example.com
  EMAIL_PASS=email_password
- Start the backend server:

  npm run server

By default, the backend server will start at *http://localhost:5000*.

Step 3: Frontend Setup (React.js + Tailwind CSS)

The frontend includes the customer shopping experience and the admin dashboard, all built using React.

- Navigate to the frontend folder:
  cd ../Frontend
- Install frontend dependencies:
  npm install
- Create a *.env* file in the *Frontend* directory:
  touch .env
- Add the API base URL to the *.env* file:
  VITE_API_URL=http://localhost:5000
- Start the React development server:
  npm run dev

The frontend should now be accessible on http://localhost:5173.

Step 4: MongoDB Database Setup

ShopEZ uses MongoDB to store data such as product listings, user information, orders, and wishlist items.

You have two options:

- **Option 1 (Local):** Install MongoDB on your machine and use:
  MONGO_URI=mongodb://127.0.0.1:27017/shopez
- **Option 2 (Cloud):** Use MongoDB Atlas and provide your connection URI in the Backend *.env* file.

Make sure the database is accessible and collections like users, products, orders, and categories are created through backend API calls or seed data.

---

5. FOLDER STRUCTURE

5.1. Frontend (Client):

```
/Frontend
  /node_modules
  /public
  /src
     /app
     /Features
     /Pages
     /assets
     /utils
     App.jsx
     App.css
```

constants.js
index.css
main.jsx

Folder/File	Description
/node_modules	Contains all the frontend dependencies installed via *npm*. Not manually edited.
/public	Static files like *favicon*, *manifest*, and *index.html* reside here. They are accessible directly via the browser.
/src	Core application source code resides here.
/app	Configuration for the Redux store setup and slice integration.
/Features	Houses Redux slices and logic for state management.
/Pages	Individual page components (e.g., Home, Login, Register, ProductList, Checkout). These are routed components.
/assets	Stores all static files such as images, logos, and icons.
/utils	Utility/helper functions used across the app (e.g., date formatter, validators, etc.).
App.jsx	Root component that wraps the whole application, includes layout and routing logic.
App.css	Main styling file for general app-level styling.
constants.js	Contains constant values used throughout the frontend (e.g., URLs, roles, enums).
index.css	Global styling or Tailwind CSS base styles.
main.jsx	Entry point of the React app; renders the *App* component into the DOM.

5.2. Backend (Server)

/Backend
/controllers
/db
/node_modules
/router
/models
/middleware
/service
index.js
brand.json
category.json
constants.js
products.json
package-lock.json
package.json

Folder/File	Description
/controllers	Contains controller files responsible for handling the logic behind each route (e.g., productController, authController).
/db	Database connection setup (e.g., MongoDB configuration and initialization).
/node_modules	Contains all backend dependencies installed via *npm*.
/router	Defines all API routes (e.g., authRoutes.js, productRoutes.js).
/models	Mongoose schema definitions for collections like User, Product, Order, etc.
/middleware	Contains middleware for authentication, error handling, etc.
/service	Business logic or helper services (e.g., mailService)
index.js	Main server entry point; initializes Express app and connects to the database.
brand.json, category.json, products.json	Static JSON datasets used for testing or initial seeding of product data.
constants.js	Application-wide constants like status codes, roles, environment settings.
package-lock.json	Lockfile to ensure consistent dependency versions.
package.json	Lists dependencies, scripts, and metadata about the backend project.

---

6. RUNNING THE APPLICATION

6.1. Backend (Server)

To run the backend of the ShopEZ application, follow these steps:

Step 1: Navigate to the Backend Directory

- Open the terminal and move into the backend folder using the command:
  cd Backend

Step 2: Install Dependencies

- Run the following command to install the necessary Node.js packages listed in *package.json*:
  npm install

Step 3: Environment Configuration

- Create a .env file in the root of the Backend directory and define all necessary environment variables, such as:
  - PORT: The port number on which the backend server should run.
  - MONGO_URI: The connection string to the MongoDB database.

- ACCESS_TOKEN_SECRET and REFRESH_TOKEN_SECRET: Used for JWT authentication.
- EMAIL_USER and EMAIL_PASS: For sending transactional emails.
- BASE_URL: Frontend base URL.

Step 4: Start the Server

- Once the environment is configured, run the following command to start the backend server:
  npm start

6.2. Frontend (Client)

To run the frontend of the application, follow these steps:

Step 1: Navigate to the Frontend Directory:

- In a new terminal window, move into the frontend folder:
  cd Frontend

Step 2: Install Dependencies:

- Install the required frontend packages using:
  npm install

Step 3: Run the Frontend Development Server

- Start the frontend using the development command (typically for Vite or React):
  npm run dev

This will start the frontend server, usually accessible at http://localhost:5173.

Step 4: Access the Application:

- Open a browser and navigate to the frontend URL. From there, we can:
  - Register/Login as a user.
  - Browse products and view product details.
  - Add products to the cart and place orders.
  - Access admin functionalities if logged in as an admin.

---

7. API DOCUMENTATION

7.1. User Routes

7.1.1. Register a New User

- **Method:** POST
- **URL:** /api/users/register
- **Headers:** Content-Type: application/json
- **Body Parameters:**

```
{
  "name": "Shubham Mehta",
  "email": "shubhammehta212@gmail.com",
  "password": "1234567890"
}
```

- **Success Response:**

```
{
  "message": "User registered successfully",
  "user": { "id": "abc123", "name": "Shubham Mehta", "email":
"shubhammehta212@gmail.com" },
  "token": "JWT_TOKEN"
}
```

- **Error Response**

```
{
  "error": "Email already in use"
}
```

7.1.2. User Login

- **Method:** POST
- **URL:** /api/users/login
- **Headers:** Content-Type: application/json
- **Body Parameters:**

```
{
  "email": "shubhammehta212@gmail.com",
  "password": "1234567890"
}
```

- **Success Response:**

```
{
  "token": "JWT_TOKEN",
  "user": { "id": "abc123", "name": "Shubham Mehta" }
}
```

- **Error Response:**

```
{
  "error": "Invalid email or password"
}
```

7.2. Product Routes

7.2.1. Get All Products

- **Method:** GET
- **URL:** /api/products
- **Headers:** None
- **Success Response:**

```
[
  { "id": "p1", "name": "iPhone 16", "price": 771, "stock": 10 },
  ...
]
```

7.2.2. Get Product by ID

- **Method:** GET
- **URL:** /api/products/:id
- **Success Response:**
  ```
  {
    "id": "p1",
    "name": "iPhone 16",
    "description": "Latest Apple iPhone",
    "price": 771
  }
  ```
- **Error Response:**
  ```
  {
    "error": "Product not found"
  }
  ```

7.3. Cart Routes

7.3.1. Add to Cart

- **Method:** POST
- **URL:** /api/cart/add
- **Headers:** Authorization: Bearer JWT_TOKEN
- **Body Parameters:**
  ```
  {
    "productId": "p1",
    "quantity": 2
  }
  ```
- **Success Response:**
  ```
  {
    "message": "Item added to cart"
  }
  ```

7.3.2. View Cart

- **Method:** GET
- **URL:** /api/cart
- **Headers:** Authorization: Bearer JWT_TOKEN
- **Success Response:**
  ```
  {
    "cartItems": [
      { "productId": "p1", "name": "iPhone 16", "quantity": 2 }
    ]
  }
  ```

7.4. Order Routes

7.4.1. Place Order

- **Method:** POST
- **URL:** /api/orders
- **Headers:** Authorization: Bearer JWT_TOKEN
- **Body Parameters:**
  ```
  {
    "items": [{ "productId": "p1", "quantity": 2 }],
    "shippingAddress": "123 Street, City",
  ```

```
  "paymentMethod": "Cash"
}
```

- **Success Response:**

```
{
  "message": "Order placed successfully",
  "orderId": "o123"
}
```

7.4.2. Get User Orders

- **Method:** GET
- **URL:** /api/orders/myorders
- **Headers:** Authorization: Bearer JWT_TOKEN
- **Success Response:**

```
[
  { "orderId": "o123", "status": "Processing" },
  ...
]
```

7.5. Authentication Routes

7.5.1. Register User

- **Method:** POST
- **URL:** /api/auth/register
- **Body Parameters:**

```
{
  "name": "Shubham Mehta",
  "email": "shubhammehta212@gmail.com",
  "password": "1234567890"
}
```

- **Success Response:**

```
{
  "message": "User registered successfully",
  "token": "JWT_TOKEN"
}
```

7.5.2. Login User

- **Method:** POST
- **URL:** /api/auth/login
- **Body Parameters:**

```
{
  "email": "shubhammehta212@gmail.com",
  "password": "1234567890"
}
```

- **Success Response:**

```
{
  "message": "Login successful",
  "token": "JWT_TOKEN"
}
```

7.6. Wishlist Routes

7.6.1. Add to Wishlist

- **Method:** POST
- **URL:** /api/wishlist
- **Headers:** Authorization: Bearer JWT_TOKEN
- **Body Parameters:**
  ```
  {
    "productId": "p1"
  }
  ```
- **Success Response:**
  ```
  {
    "message": "Product added to wishlist"
  }
  ```

7.6.2. Get Wishlist

- **Method:** GET
- **URL:** /api/wishlist
- **Headers:** Authorization: Bearer JWT_TOKEN
- **Success Response:**
  ```
  [
    { "productId": "p1", "productName": "iPhone 16" },
    ...
  ]
  ```

7.6.3. Remove from Wishlist

- **Method:** DELETE
- **URL:** /api/wishlist/:productId
- **Headers:** Authorization: Bearer JWT_TOKEN
- **Success Response:**
  ```
  {
    "message": "Product removed from wishlist"
  }
  ```

7.7. Admin Dashboard Routes

7.7.1. Get All Users

- **Method:** GET
- **URL:** /api/admin/users
- **Headers:** Authorization: Bearer JWT_TOKEN (Admin Only)
- **Success Response:**
  ```
  [
    { "userId": "abc123", "name": "Shubham Mehta", "email":
  "shubhammehta212@gmail.com" },
    ...
  ]
  ```

7.7.2. Get All Orders

- **Method:** GET
- **URL:** /api/admin/orders
- **Headers:** Authorization: Bearer JWT_TOKEN (Admin Only)
- **Success Response:**
  ```
  [
    { "orderId": "o123", "status": "fulfilled" },
  ```

```
  ...
]
```

7.7.3. Update Order Status

- **Method:** PUT
- **URL:** /api/admin/orders/:orderId
- **Headers:** Authorization: Bearer JWT_TOKEN (Admin Only)
- **Body Parameters:**
```
{
  "status": "fulfilled"
}
```
- **Success Response:**
```
{
  "message": "Order status updated"
}
```

7.8. Brand Routes

7.8.1. Get All Brands

- **Method:** GET
- **URL:** /api/brands
- **Success Response:**
```
[
  { "brandId": "b1", "brandName": "Apple" },
  ...
]
```

7.8.2. Add New Brand (Admin)

- **Method:** POST
- **URL:** /api/brands
- **Headers:** Authorization: Bearer JWT_TOKEN (Admin Only)
- **Body Parameters:**
```
{
  "brandName": "Sony"
}
```
- **Success Response:**
```
{
  "message": "Brand added successfully"
}
```

7.9. Category Routes

7.9.1. Get All Categories

- **Method:** GET
- **URL:** /api/categories
- **Success Response:**
```
[
  { "categoryId": "c1", "categoryName": "Mobiles" },
  ...
]
```

7.9.2. Add New Category (Admin)

- **Method:** POST

- **URL:** /api/categories
- **Headers:** Authorization: Bearer JWT_TOKEN (Admin Only)
- **Body Parameters:**
```
{
  "categoryName": "Accessories"
}
```
- **Success Response:**
```
{
  "message": "Category added successfully"
}
```

---

8. AUTHENTICATION

8.1. JWT-Based Authentication

- **Authentication** in ShopEZ is handled using **JWT (JSON Web Tokens)** stored in cookies.
- After login or registration, an accessToken, refreshToken, and loggedInUserInfo are set as cookies on the client.
- The accessToken is used for authorizing API requests.
- The refreshToken is used to **renew** expired access tokens without requiring re-login.

8.1.1. How it Works

- verifyJWT middleware checks if the accessToken, refreshToken, and loggedInUserInfo cookies are present.
- It verifies the accessToken using ACCESS_TOKEN_SECRET. If expired or invalid, it calls the renewTokens() function.
- The renewTokens() method:
    - Verifies the refreshToken using REFRESH_TOKEN_SECRET.
    - If valid, it generates **new access and refresh tokens**, stores them, and continues the request.
- The user ID and role are extracted and attached to the request via req.body.user.

8.2. Role-Based Access Control (RBAC)

- **Role support** includes:
    - user: Can access user-level endpoints (cart, wishlist, orders, profile).
    - admin: Can access admin-level features (product, brand, category, user management).
- verifyAdminJWT is a dedicated middleware that:
    - Checks if the logged-in user's role is admin.
    - If not, it blocks the request with a 401 Unauthorized error.

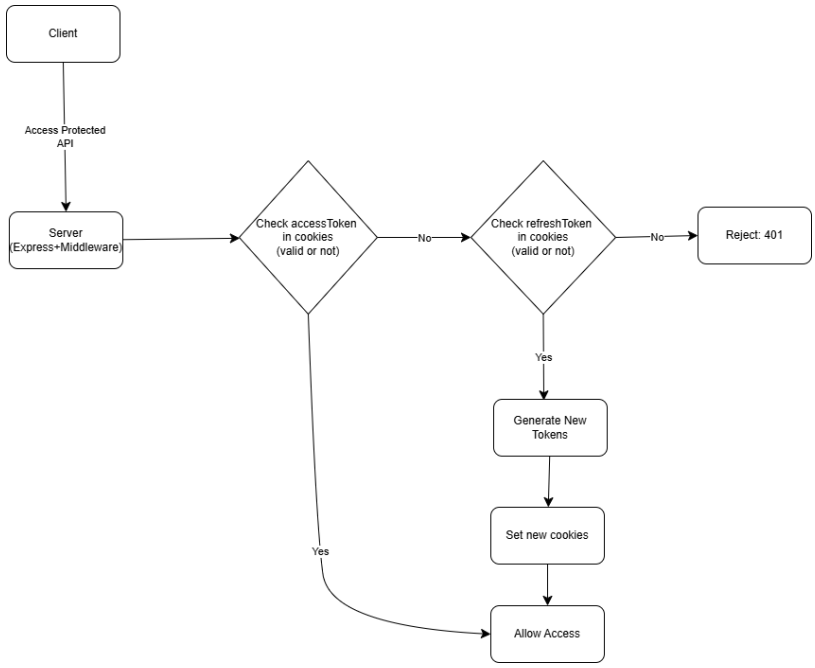
```
if(role !== "admin") {

  return next(new ApiError(401, "Unauthorized"));
```

}

8.3. Token Expiration & Refresh Handling

- **Token Expiry Settings:**
    - accessToken: 10 minutes (maxAge: 600000)
    - refreshToken: 20 minutes (maxAge: 1200000)
- When the accessToken expires:
    - The renewTokens() function is triggered automatically.
    - It checks the refreshToken and user ID.
    - New tokens are generated and stored in cookies without affecting the user experience.
- This ensures seamless and secure sessions for users.

8.4. Token Flow (Middleware)



8.5. Middleware Files

File	Purpose
auth.middleware.js	Handles token verification, renewal, and RBAC for both users and admins.
pagination.middleware.js	Adds pagination logic to any route fetching long lists (e.g., products, orders).

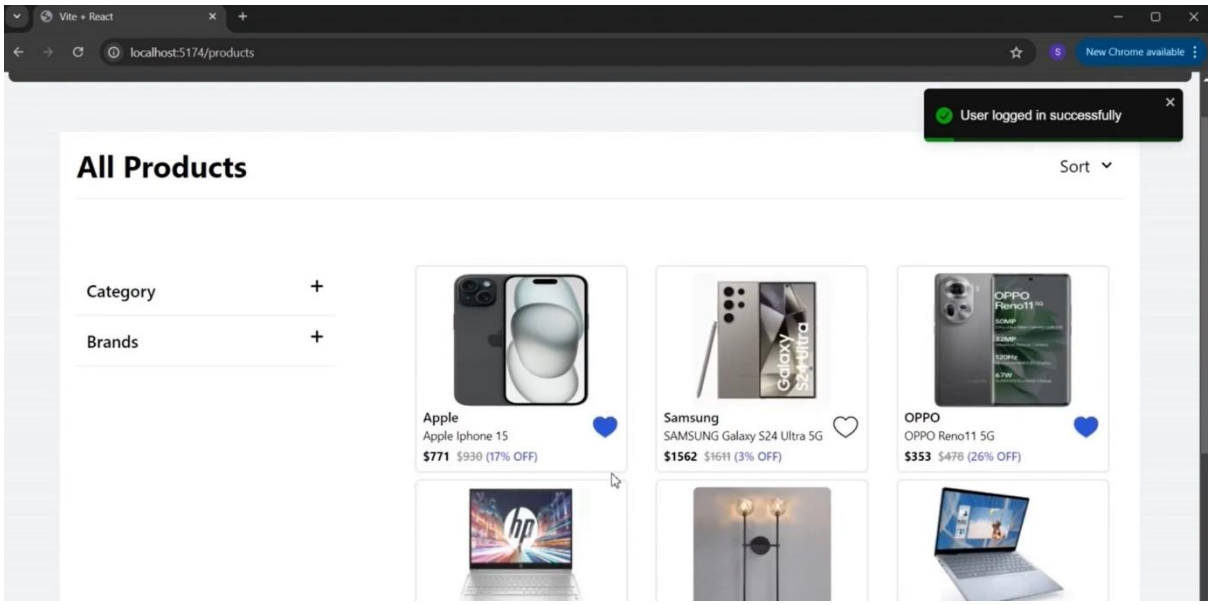
8.6. Example Protected Endpoints

Endpoint	Role	Middleware
GET /api/user/profile	user	verifyJWT
POST /api/orders	user	verifyJWT
POST /api/admin/product	admin	verifyAdminJWT

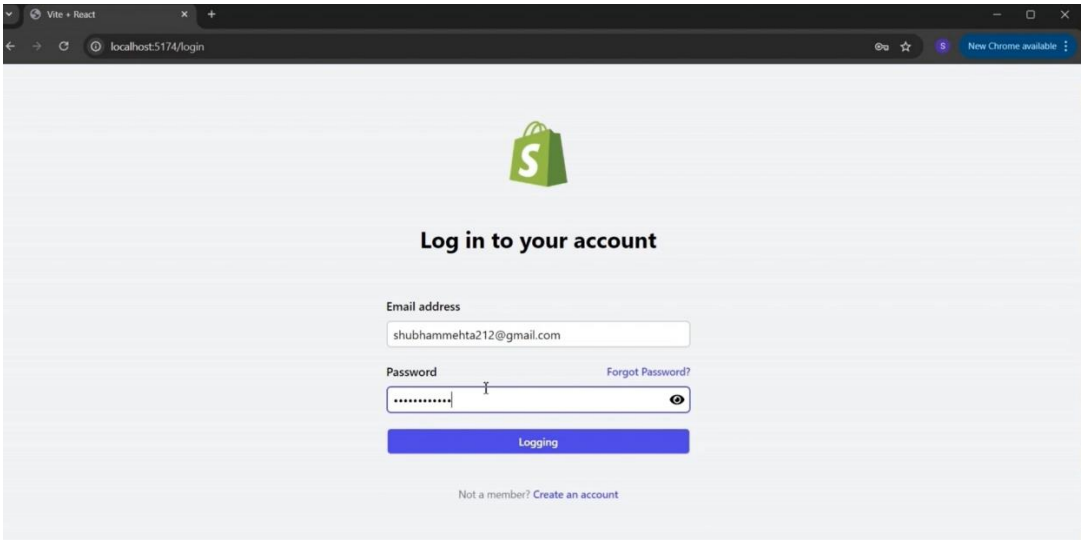
GET /api/admin/users	admin	verifyAdminJWT

9. USER INTERFACE

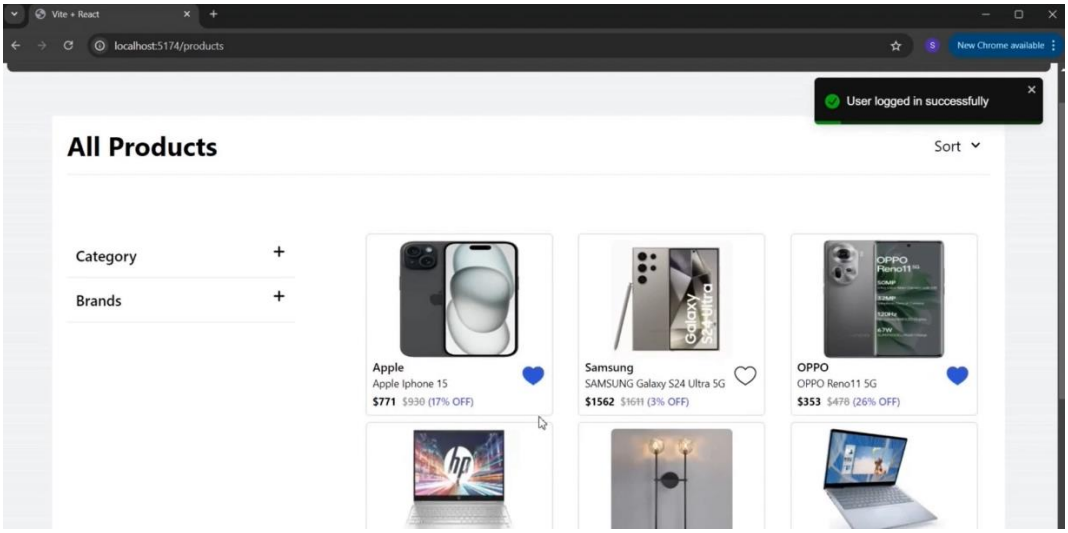
9.1. Landing Page



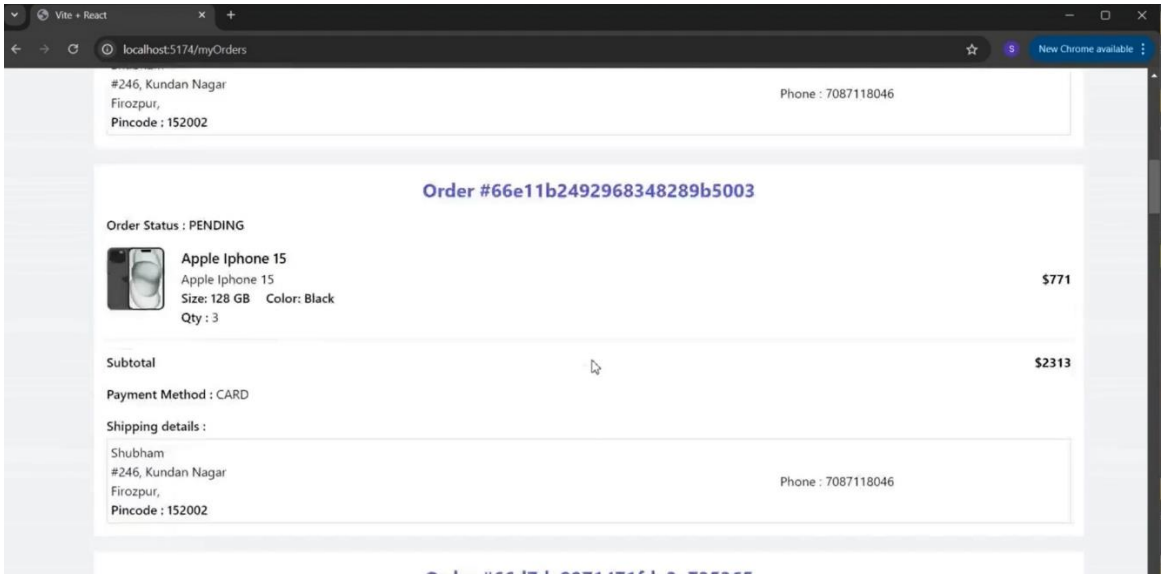
9.2. Login



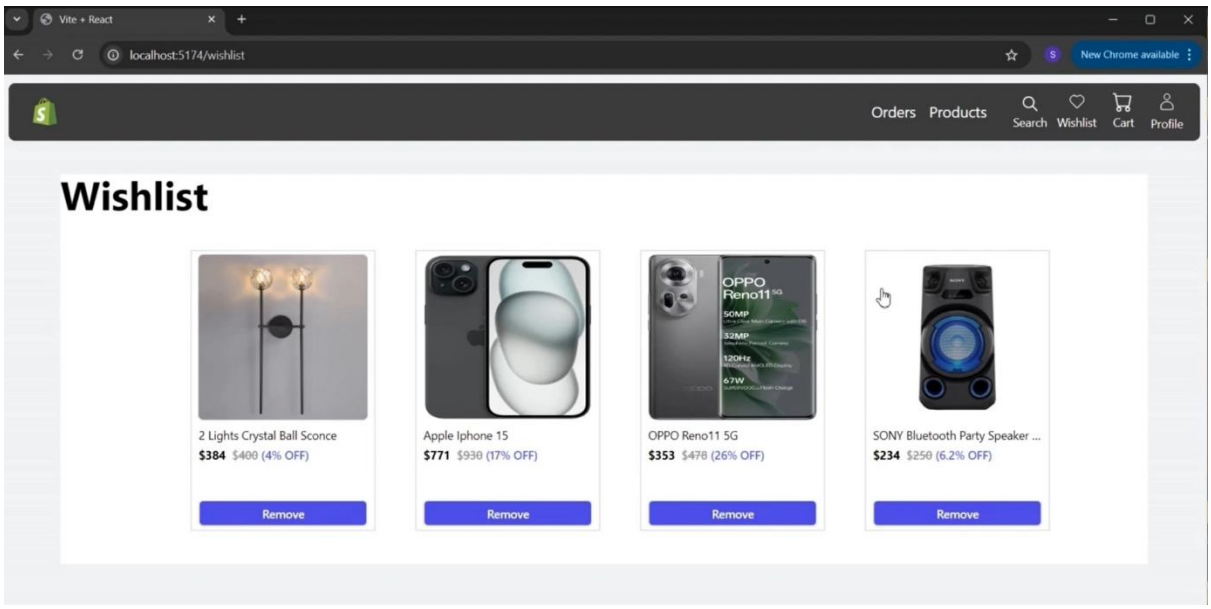
9.3. User Dashboard



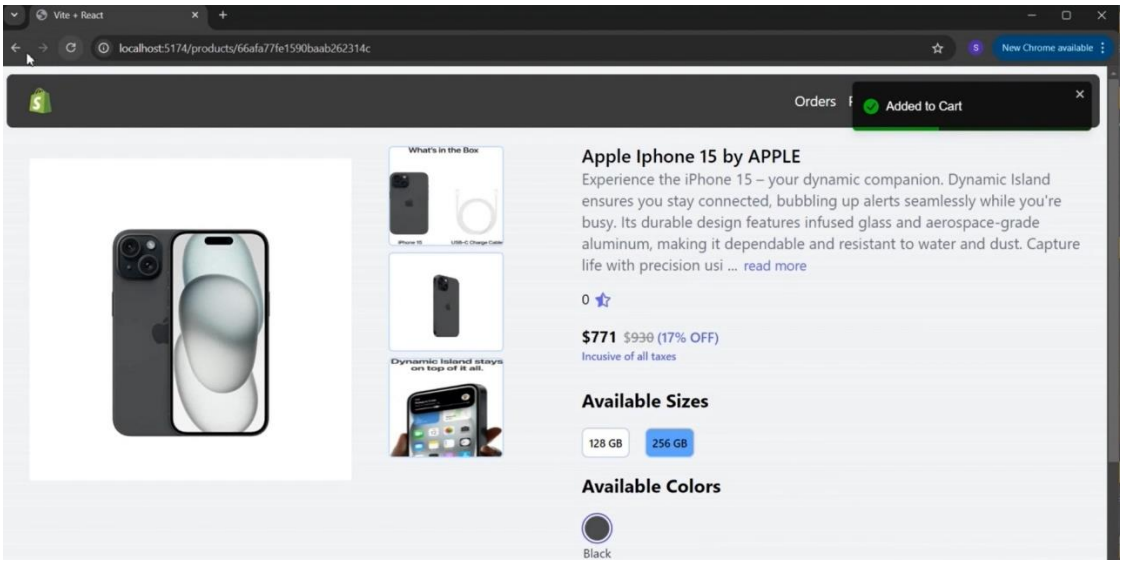
9.4. Order History



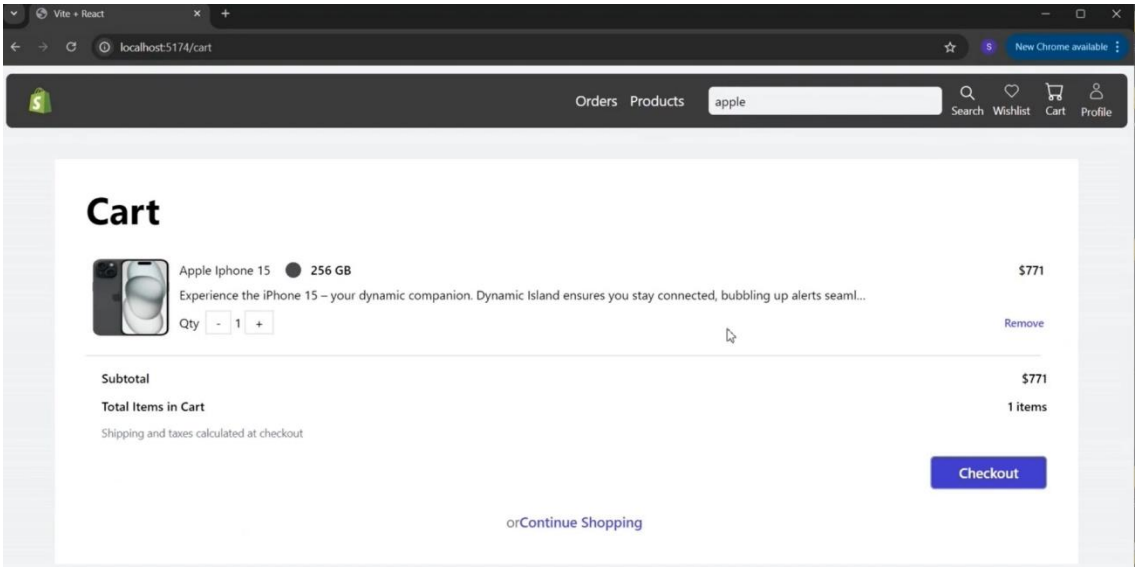
9.5. Wishlist



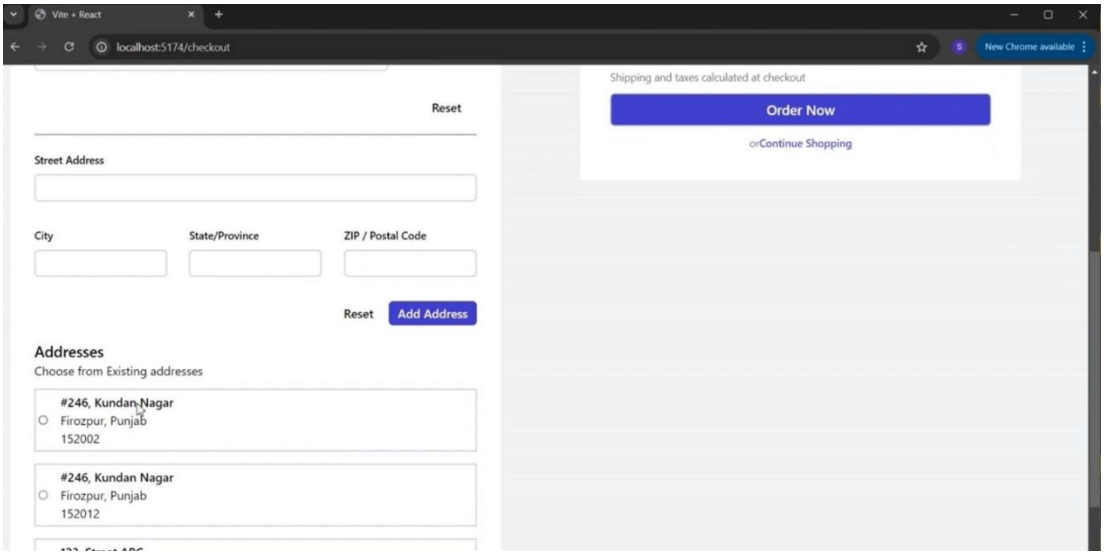
9.6. Individual Products Page

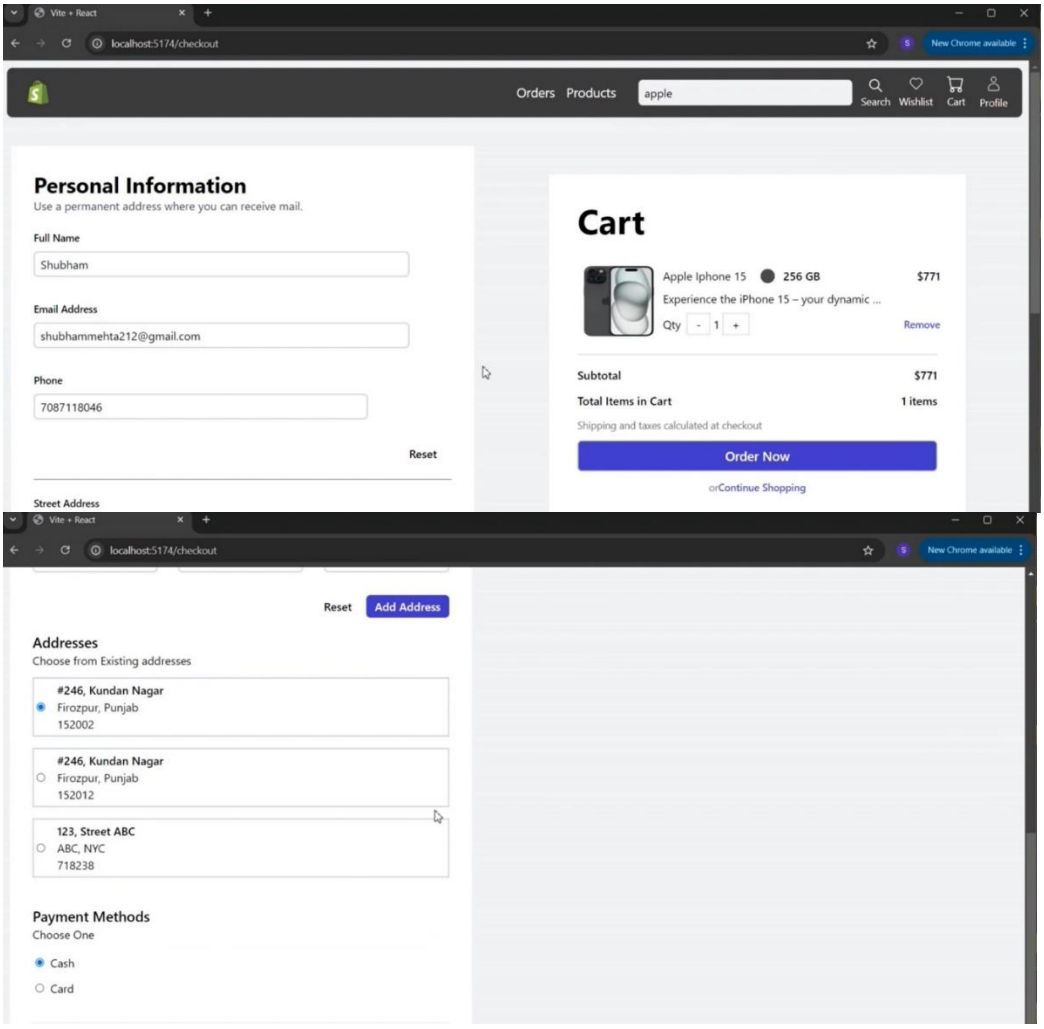


9.7. Cart

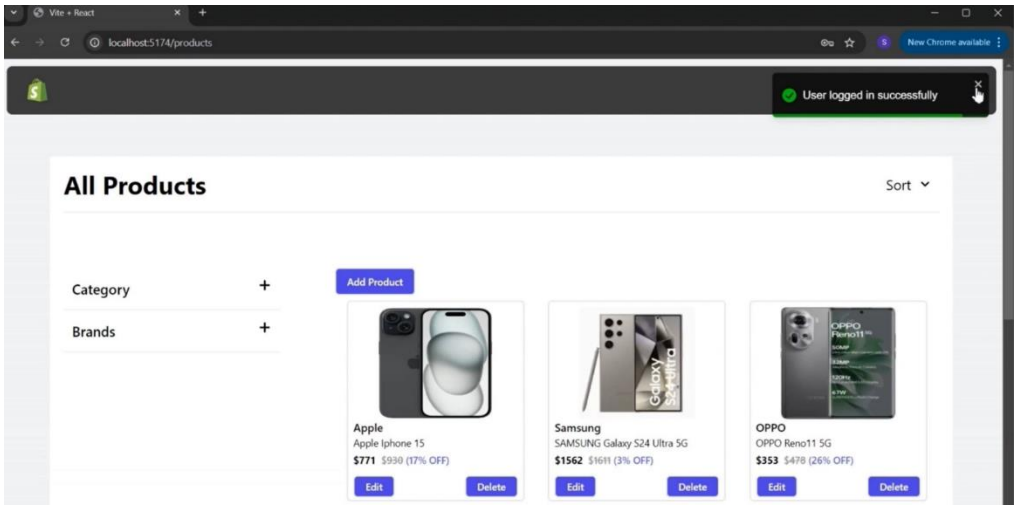


9.8. Checkout

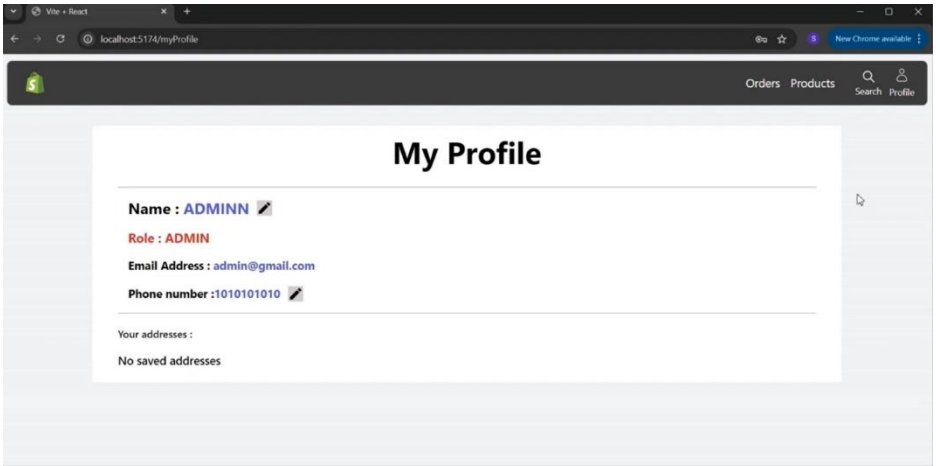




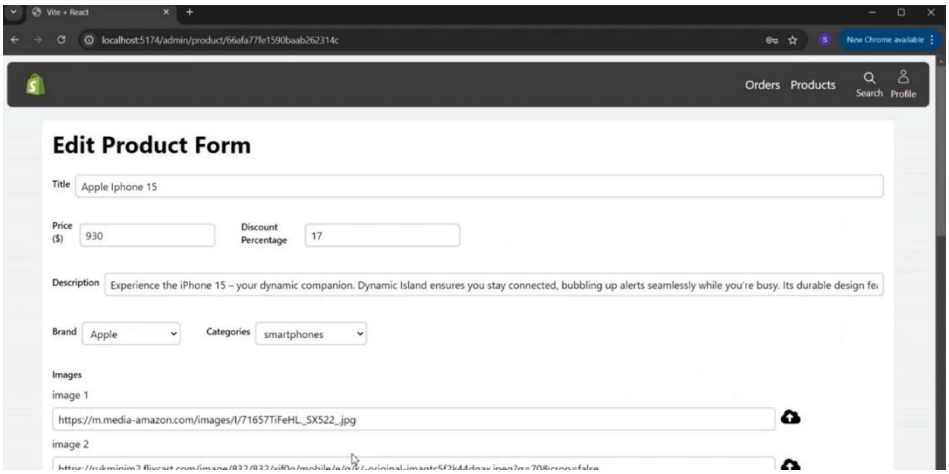
9.9. Admin Dashboard

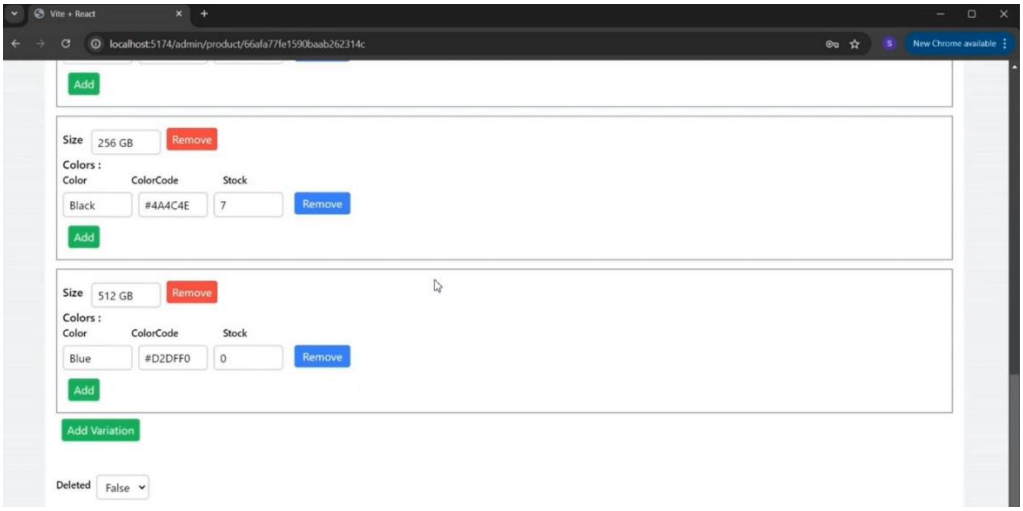


9.10. Admin Profile

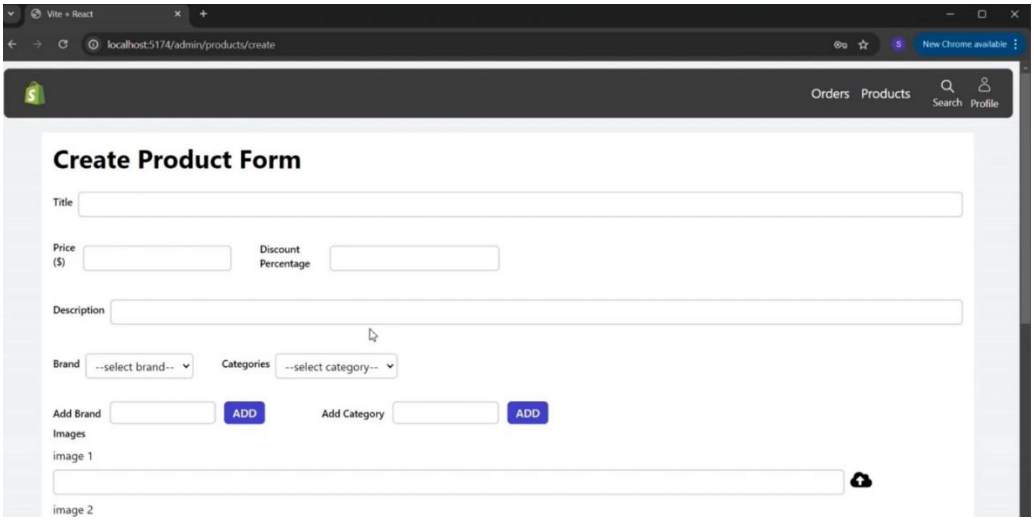


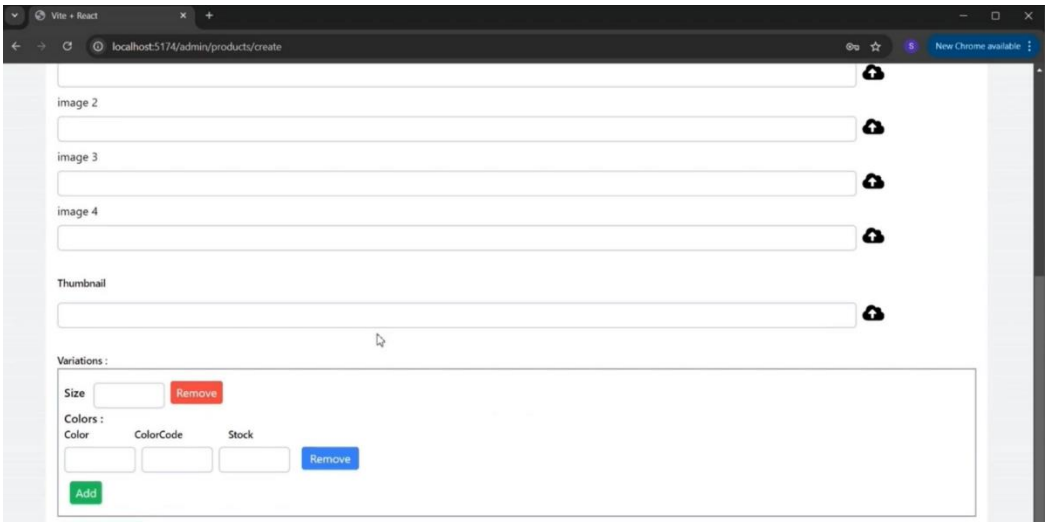
9.11. Edit Product Form



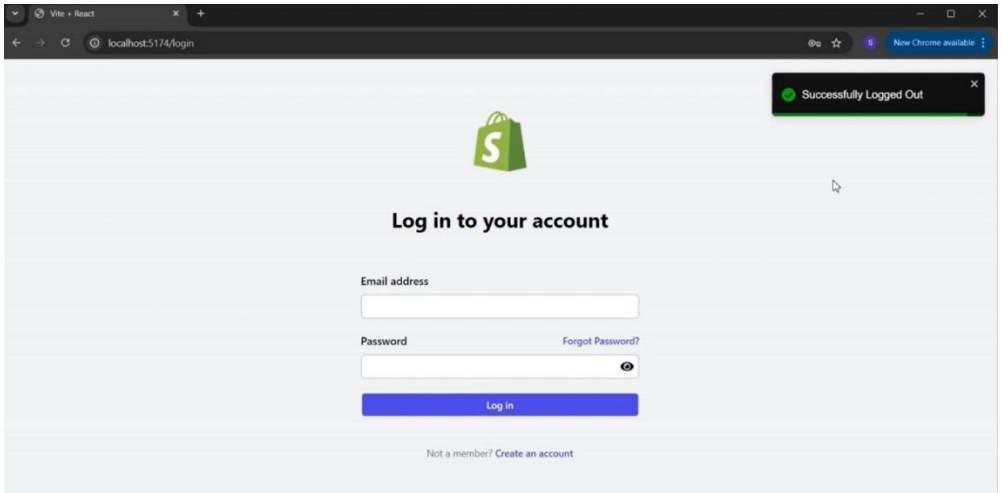


9.12. Create Product Form


Vite + React
localhost:5174/admin/products/create
New Chrome available
Orders
Products
Search
Profile
Create Product Form
Title
Price ($)
Discount Percentage
Description
Brand
--select brand--
Categories
--select category--
Add Brand
ADD
Add Category
ADD
Images
image 1
image 2


Vite + React
localhost:5174/admin/products/create
New Chrome available
image 2
image 3
image 4
Thumbnail
Variations :
Size
Remove
Colors :
Color
ColorCode
Stock
Remove
Add

9.13. Logout


Vite + React
localhost:5174/login
New Chrome available
Successfully Logged Out
Log in to your account
Email address
Password
Forgot Password?
Log in
Not a member? Create an account

10. TESTING

10.1. FUNCTIONAL AND PERFORMANCE TESTING

10.1.1. Performance Testing (GenAI Functional & Performance Testing)

10.1.2. Test Scenarios & Results

Test Case ID	Scenario (What to test)	Test Steps (How to test)	Expected Result	Actual Result	Pass/Fail
FT-01	Text Input Validation (e.g., topic, job title)	Enter valid and invalid text in input fields	Valid inputs accepted, errors for invalid inputs	Valid inputs generated content; invalid entries showed proper error messages	Pass
FT-02	Number Input Validation (e.g., word count, size, rooms)	Enter numbers within and outside the valid range	Accepts valid values, shows error for out-of-range	Accepts numbers in range; error shown for out-of-range values	Pass
FT-03	Content Generation (e.g., blog, resume, design idea)	Provide complete inputs and click "Generate"	Correct content is generated based on input	Accurate content generated matching the input context	Pass
FT-04	API Connection Check	Check if API key is correct and model responds	API responds successfully	Connection stable; API responded on every call	Pass
PT-01	Response Time Test	Use a timer to check content generation time	Should be under 3 seconds	Average response time: 2.4 seconds	Pass
PT-02	API Speed Test	Send multiple API calls at the same time	API should not slow down	API handled 5+ concurrent calls without delay or failure	Pass

PT-03	Image Upload Load Test (Admin-Product Images)	Upload multiple PDFs and check processing	Images should upload quickly without delay or errors. The product should be created successfully, and images should display correctly in product listings.	Admin uploaded multiple high-res images without delay or crash. Product was created successfully, and images displayed correctly...	Pass

---

11. DEMO

To offer a clear understanding of the platform's functionality and user experience, we have prepared a demo video showcasing the end-to-end flow of the **ShopEZ** application. The video highlights both **user-side operations**—like browsing products, adding items to the cart, logging in, placing orders—as well as **admin-side capabilities**, including managing products, updating order statuses, and overseeing user data. This visual walkthrough serves as a comprehensive guide for stakeholders, evaluators, or contributors to grasp how the application operates in real-time. It also demonstrates the responsiveness and interactive nature of the UI across different pages.

Click here to watch the demo video

---

12. KNOWN ISSUES

Despite thorough development and testing, a few known issues and limitations currently exist in the ShopEZ platform:

12.1. Token Expiry Edge Case

- If the refresh token expires during user activity, re-authentication might be required manually.
- Auto-logout mechanism is planned but not yet implemented.

12.2. Cart Not Synced Across Devices

- Cart items are stored locally; logging in from a different device may not retain previous cart contents.

12.3. Limited Inventory Updates

- Inventory does not automatically update when an order is placed; relies on manual or delayed updates.

12.4. Responsiveness Issues on Older Devices

- Some UI components may break on smaller or older mobile devices due to limited media query handling.

12.5. Slow Loading on Initial Render

- The homepage can have a slight delay during the first load due to large JSON file parsing (products, brands).

12.6. No Real-Time Order Status

- Order status updates are not pushed in real-time. Users must refresh the page to view updates.

12.7. Password Reset Not Fully Functional

- The "Forgot Password" feature is under development and currently non-functional.

---

13. FUTURE ENHANCEMENTS

While the current version of ShopEZ delivers core functionalities for an e-commerce platform, there are several areas that can be enhanced or expanded upon in future iterations:

13.1. Enhanced Payment Integration

- Integrate real-time payment gateways like Razorpay, Stripe, or PayPal for secure and smooth transactions.
- Enable support for UPI, Net Banking, and Wallets.

13.2. Mobile Application Development

- Develop a dedicated mobile app using React Native or Flutter to improve accessibility and reach.
- Provide push notifications for order updates and offers.

13.3. Smart Recommendation System

- Use machine learning to recommend products based on user behavior, search history, and purchase trends.

13.4. Advanced Search & Filters

- Implement Elasticsearch or Algolia for better product discovery through full-text search, voice search, and advanced filters.

13.5. Inventory & Order Management for Sellers

- Allow vendors or sellers to manage their own products, track inventory, and view orders from a separate dashboard.

13.6. Admin Analytics Dashboard

- Add data visualizations for sales trends, customer activity, and inventory using tools like Chart.js or Recharts.

13.7. Multi-language & Currency Support

- Enable users to view the site in their preferred language and currency to support a global audience.

13.8. Enhanced Security Features

- Add 2FA (Two-Factor Authentication), CAPTCHA on login, and real-time threat monitoring.

13.9. Invoice Generation & Order Tracking

- Automatically generate PDF invoices and allow users to track their orders in real-time with shipment status.

13.10. Accessibility & Performance Improvements

- Improve site accessibility (WCAG compliance) and optimize performance with lazy loading, caching, and CDN.
