**1.INTRODUCTION:**

**PROJECT TITLE:**

**SHOPSMART-DIGITAL GROCERY STORE EXPERIENCE**

**TEAM MEMBERS:**

1.SYAM SURYA KUMAR.N - FRONTEND DEVELOPER

2.SYAM KRISHNA.N - BACKEND DEVELOPER

3.SATVIKA NAGA DEVI.N - DATABASE MANAGER

4.NANDINI.CH - TESTER

**2.PROJECT OVERVIEW:**

**PURPOSE OF THE PROJECT:**

The purpose of this project is to create a digital platform that makes grocery shopping easier, faster, and more convenient. It aims to simulate a real-world online grocery store where users can browse products, add them to a cart, and place orders from the comfort of their homes.

### ****GOALS OF THE PROJECT****:

1. **Simplify the grocery shopping experience** by providing a user-friendly web application.
2. **Enable online product selection and purchase** with features like search, filter, and cart.
3. **Develop a secure system** for customer login, order placement, and payment process.
4. **Provide an admin interface** to manage products, orders, and user data effectively.
5. **Ensure a responsive and fast-loading website** using modern full stack technologies.

### ****KEY FEATURES AND FUNCTIONALITIES****

#### USER SIDE:

1. **User Registration and Login**
   * Secure sign-up and login system for customers.
2. **Product Browsing**
   * View all available grocery items with details like name, price, and image.
3. **Search and Filter**
   * Quickly search for products and filter by category or price.
4. **Add to Cart**
   * Users can add products to a virtual shopping cart.
5. **Cart Management**
   * Increase/decrease quantity, remove items, view total price.
6. **Checkout and Order Placement**
   * Place orders with address details and confirm the purchase.
7. **Order History**
   * View past orders and their status.

#### ADMIN SIDE:

1. **Admin Login**
   * Secure access for store admin.
2. **Product Management**
   * Add new products, update product details, or delete items.
3. **Order Management**
   * View all orders placed by users and update their delivery status.

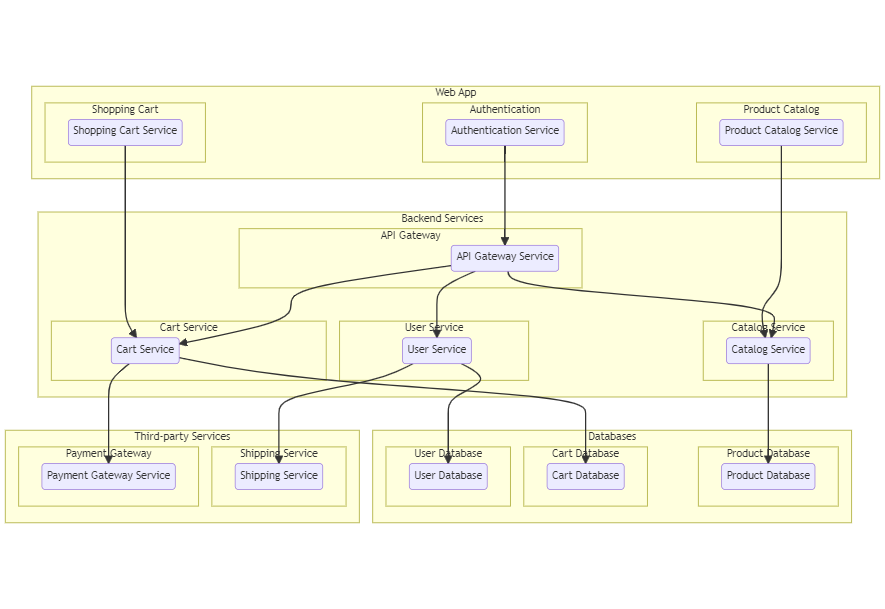
#### GENERAL FUNCTIONALITIES:

* **Responsive Design** – Works well on mobile, tablet, and desktop.
* **Real-time Data Handling** – Updated product/cart/order data using database connection.
* **User-Friendly Interface** – Clean and easy-to-navigate UI built using React.js.
* **Secure Backend** – Node.js and Express.js used to handle APIs and user data securely.
* **Database Integration** – MongoDB used to store product, user, and order details.

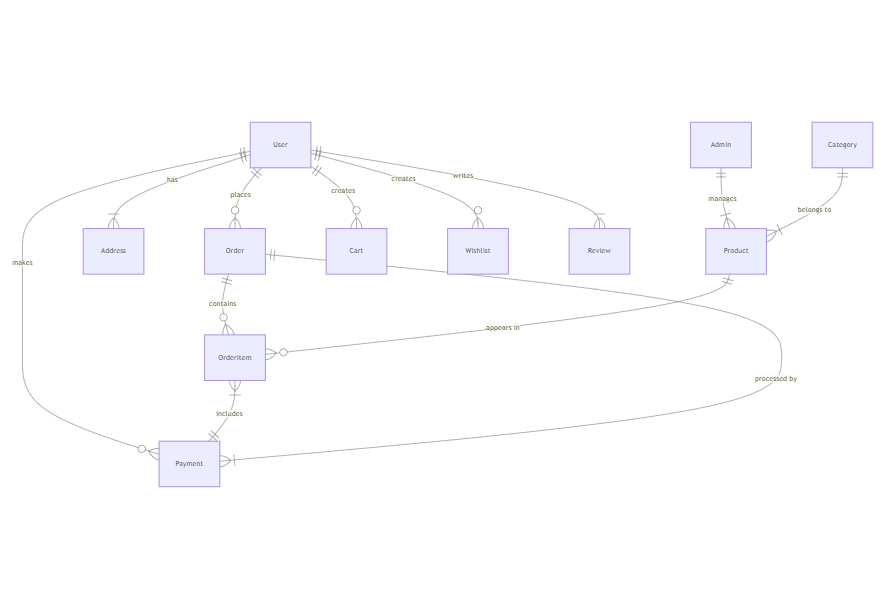
**3.ARCHITECTURE**

**Technical Architecture:**

The technical architecture of an flower and gift delivery app typically involves a client-server model, where the frontend represents the client and the backend serves as the server. The frontend is responsible for user interface, interaction, and presentation, while the backend handles data storage, business logic, and integration with external services like payment gateways and databases. Communication between the frontend and backend is typically facilitated through APIs, enabling seamless data exchange and functionality.

****

**ER DIAGRAM:**

:

### ****FRONTEND ARCHITECTURE USING REACT.JS****

The frontend of the **ShopSmart** application is built using **React.js**, a component-based JavaScript library for building interactive UIs. The architecture follows a **modular, reusable, and scalable structure**.

### ****ARCHITECTURE OVERVIEW:****

#### 1.COMPONENT BASED STRUCTURE:

* The entire UI is broken into small, reusable components.
* Each component manages its own logic and rendering.

##### **COMMON COMPONENTS:**

* Header.js – Navigation bar with links to Home, Cart, Login, etc.
* Footer.js – Bottom section with copyright.
* ProductCard.js – Displays individual product details.
* CartItem.js – Shows each item in the cart.
* FormInput.js – Custom input component for forms.

#### 2.PAGES/VIEWS:

These are full-page components that represent different screens in the app.

* HomePage.js – Shows all products.
* LoginPage.js – User login screen.
* RegisterPage.js – New user registration.
* CartPage.js – Displays cart items.
* CheckoutPage.js – Order confirmation.
* AdminPage.js – Admin dashboard for managing products and orders.

#### 3.STATE MANAGEMENT:

* **useState** – For managing local component states like cart count or input fields.
* **useEffect** – For handling side effects like fetching data from API.
* Optionally, **Context API** or **Redux** can be used for global state management (if app is large).

**4.ROUTING (NAVIGATION):**

* **React Router DOM** is used for navigation between pages.

jsx

CopyEdit

<Route path="/" element={<HomePage />} />

<Route path="/cart" element={<CartPage />} />

<Route path="/login" element={<LoginPage />} />

#### 5.API COMMUNICATION:

* Uses **Axios** or **Fetch API** to call backend APIs (Node.js + Express).
* Example: GET /api/products to fetch all grocery items.

#### 6.STYLING:

* CSS Modules or plain CSS/SCSS is used for styling.
* Optionally, libraries like **Bootstrap** or **Tailwind CSS** may be used for responsive design.

### ****FRONTEND WORKFLOW:****

1. User opens the app → React loads the HomePage.
2. React fetches product data via API and displays it using ProductCard.
3. User adds items to cart → React updates cart state.
4. User proceeds to checkout → Data is sent to backend using POST API.
5. Admin views and updates orders via the AdminPage.

### ****BACKEND ARCHITECTURE USING NODE.JS AND EXPRESS.JS****

The backend of ShopSmart is built using **Node.js** (JavaScript runtime) and **Express.js** (web framework). It handles data processing, API creation, and communication with the database.

**ARCHITECTURE OVERVIEW:**

**1.EXPRESS SERVER SETUP:**

* The server is created using **Express.js**.
* It listens to client requests (like login, view products, place orders).

js

CopyEdit

const express = require('express');

const app = express();

app.listen(3000, () => console.log('Server running'));

**2.ROUTING LAYER:**

* All URLs (API endpoints) are defined using Express routes.
* Examples:
  + GET /api/products → Get all products
  + POST /api/login → Login user
  + POST /api/orders → Place an order
* Organized in different route files:
  + userRoutes.js
  + productRoutes.js
  + orderRoutes.js

**3.CONTROLLER LAYER:**

* Each route connects to a **controller function**.
* Controllers contain the actual logic (e.g., check login details, fetch product list).

js

CopyEdit

const loginUser = (req, res) => {

// validate and respond

};

**4.MIDDLEWARE:**

* Functions that run **before** or **after** the main route logic.
* Examples:
  + body-parser – to parse incoming JSON
  + authMiddleware – to check if user is logged in

js

CopyEdit

app.use(express.json());

**5.DATABASE LAYER ( MONGODB ):**

* Uses **Mongoose** to connect Node.js with **MongoDB**.
* Models are created for:
  + User (for login/signup)
  + Product (name, price, quantity)
  + Order (items, user, address)

**6.API COMMUNICATION:**

* The backend exposes REST APIs used by the React frontend.
* Data is exchanged in **JSON format**.

### ****BACKEND WORKFLOW:****

1. React frontend sends a request (e.g., /api/products).
2. Express.js receives it and routes it to the correct controller.
3. Controller interacts with MongoDB to get or update data.
4. Response is sent back to the frontend.

**DATABASE: MONGODB**

MongoDB is a **NoSQL database** that stores data in **collections** of **documents** (JSON-like format). In ShopSmart, it stores all the information related to users, products, and orders.

We use **Mongoose**, an ODM (Object Data Modeling) library for MongoDB, to define schemas and interact with the database easily in Node.js.

**KEY COLLECTIONS ( SCHEMAS ):**

### 1. ****USER SCHEMA****

Stores customer and admin details.

js

CopyEdit

const UserSchema = new mongoose.Schema({

name: String,

email: { type: String, unique: true },

password: String,

isAdmin: { type: Boolean, default: false }

});

**PURPOSE**:

* Handles login/signup
* Identifies admin or customer

### 2. ****PRODUCT SCHEMA****

Stores details of each grocery item.

js

CopyEdit

const ProductSchema = new mongoose.Schema({

name: String,

price: Number,

image: String,

description: String,

category: String,

stock: Number

});

**PURPOSE**:

* Display product listings
* Allow admin to add/edit/delete products

### 3. ****ORDER SCHEMA****

Stores customer orders and delivery details.

js

CopyEdit

const OrderSchema = new mongoose.Schema({

user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },

items: [

{

product: { type: mongoose.Schema.Types.ObjectId, ref: 'Product' },

quantity: Number

}

],

address: String,

status: { type: String, default: "Pending" },

date: { type: Date, default: Date.now }

});

**PURPOSE**:

* Tracks which user ordered what
* Stores order status and delivery address

**HOW INTERACTIONS WORK WITH (MONGOSE)**

### ****FROM FRONTEND TO BACKEND****

1. User clicks "Add to Cart" or "Place Order".
2. React app sends a **POST** request to the Node.js backend.

### ****BACKEND ACTIONS****

1. **Express Route** receives the request.
2. **Controller** processes the request.
3. **Mongoose** interacts with MongoDB:
   * User.findOne({ email }) → Fetch user
   * Product.find({}) → Get all products
   * Order.create(orderData) → Save order

### ****FROM BACKEND TO FRONTEND****

1. MongoDB returns data to the controller.
2. Controller sends a **JSON response** to the React frontend.
3. UI updates based on the returned data.

### EXAMPLE: PLACING AN ORDER

* **Frontend** → sends cart and address info.
* **Backend** → creates a new Order document.
* **MongoDB** → stores the order, linked to the user and products**.**

1. **SETUP INSTRUCTIONS**

**SOFTWARE DEPENDANCIES:**

### ****FRONTEND (REACT.JS)****

| **Dependency** | **Purpose** |
| --- | --- |
| react | Core library for building UI |
| react-dom | Renders React components to the DOM |
| react-router-dom | Handles routing between pages |
| axios | Makes HTTP requests to backend API |
| bootstrap or tailwindcss (optional) | For responsive UI design and styling |
| react-icons (optional) | Adds icons to UI easily |

### ****BACKEND (NODE.JS + EXPRESS.JS)****

| **Dependency** | **Purpose** |
| --- | --- |
| express | Web framework to create routes and handle requests |
| mongoose | Connects and interacts with MongoDB |
| dotenv | Loads environment variables from .env file |
| cors | Allows cross-origin requests between frontend and backend |
| body-parser (or included in express) | Parses incoming request data |
| bcryptjs | Encrypts user passwords |
| jsonwebtoken (JWT) | Generates tokens for secure authentication |

**DATABASE**

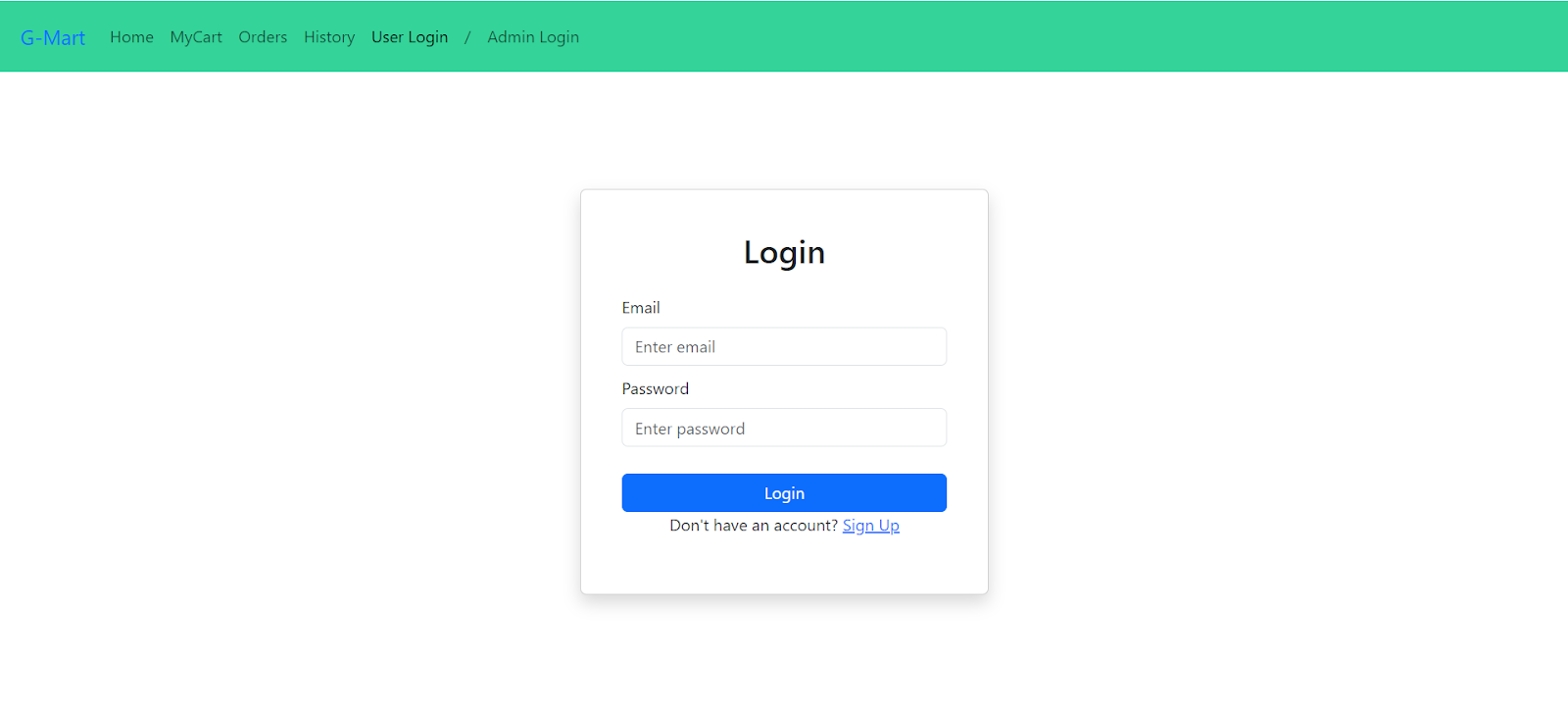
| **Dependency** | **Purpose** |
| --- | --- |
| mongodb | Database to store users, products, and orders |
| mongoose | ODM tool to define schema and interact with MongoDB |

### ****DEV TOOLS****

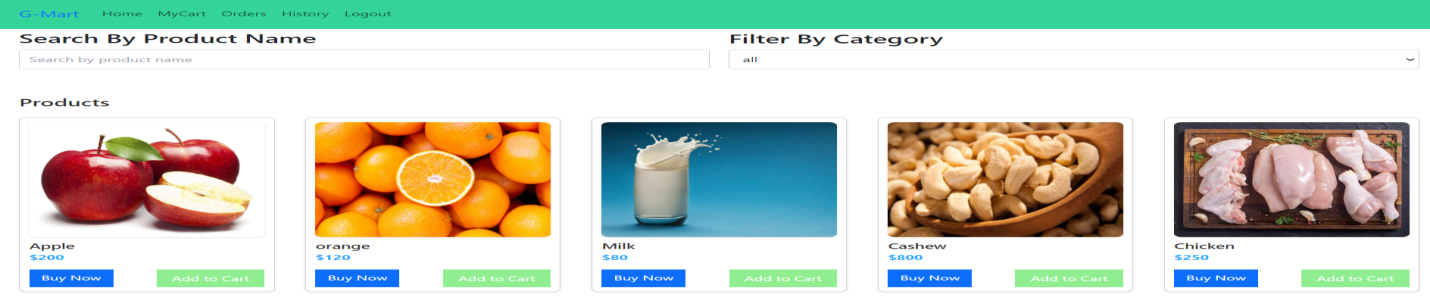
| **Dependency** | **Purpose** |
| --- | --- |
| nodemon | Auto-restarts server on file changes during development |
| concurrently | Runs both frontend and backend with one command (optional) |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| STEP-BY-STEP PROJECT SETUP GUIDE****1. PREREQUISITES**** Before starting, make sure you have:   * **Node.js** and **npm** installed * **Git** installed * **MongoDB** running locally or have a MongoDB Atlas account   **2. CLONE THE PROJECT**  Open terminal or VS Code terminal and run:  bash  CopyEdit  git clone https://github.com/your-username/shopsmart.git  Replace the URL with your actual GitHub repository link. ****3. NAVIGATE TO PROJECT FOLDER**** bash  CopyEdit  cd shopsmart ****4. SET UP FRONTEND**** bash  CopyEdit  cd frontend  npm install  This installs all frontend dependencies (like React, Axios, etc.) ****5. SET UP BACKEND**** bash  CopyEdit  cd ../backend  npm install  This installs backend dependencies (like Express, Mongoose, etc.) ****6. CREATE ENVIRONMENT VARIABLE FILE**** In the **backend folder**, create a file named .env:  bash  CopyEdit  touch .env  Paste the following content into .env:  env  CopyEdit  PORT=5000  MONGO\_URI=your\_mongodb\_connection\_string  JWT\_SECRET=your\_jwt\_secret\_key  ✨ Replace your\_mongodb\_connection\_string with your real MongoDB URI (from MongoDB Atlas or local)  ✨ Choose any string for JWT\_SECRET, e.g., mysecret123 ****7. RUN THE BACKEND SERVER**** Make sure you’re in the backend folder:  bash  CopyEdit  npm run start  or if you're using **nodemon**:  bash  CopyEdit  npm run dev ****8. RUN THE FRONTEND APP**** Open a **new terminal**, go to frontend folder:  bash  CopyEdit  cd frontend  npm start ****9. VIEW THE APP****  * **Frontend** will run at: http://localhost:3000 * **Backend API** will run at: http://localhost:5000  ****10. TEST THE SETUP****  * Visit the site * Try browsing products, registering, logging in * Use tools like **Postman** to test backend APIs   **5.FOLDER STRUCTURE:**  **CLIENT:** ****STRUCTURE OF REACT FRONTEND (FOLDER-WISE)****frontend/ This is the root folder for the React frontend. public/  * Contains static files like:   + index.html: The main HTML file loaded in the browser.   + favicon.ico: Tab icon.  src/ This is the **main development folder** – all your React code lives here. index.js  * Starting point of the app. * Renders the main <App /> component into the DOM.  App.js  * Main component that contains the **router** and **layout**. * It acts like the “central brain” of the frontend.  Suggested Folder Structure Inside src/components/  * Reusable UI blocks (used in multiple pages).   + Header.js – Top navigation bar   + Footer.js – Bottom of the page   + ProductCard.js – For showing a single product   + CartItem.js – For items in the cart  pages/  * Full-page components for different screens.   + HomePage.js – Displays all products   + LoginPage.js – Login form   + RegisterPage.js – Registration form   + CartPage.js – Shopping cart   + CheckoutPage.js – Place order   + AdminPage.js – Admin dashboard  services/ (optional but useful)  * API functions to interact with the backend using axios.   + productService.js – Fetch/add products   + userService.js – Login/register   + orderService.js – Place/view orders  context/ (if using Context API)  * For managing global states like authentication or cart.  assets/ (optional)  * Images, logos, icons used in the UI.  styles/ (optional)  * CSS or SCSS files to style your components.  REACT ROUTER EXAMPLE IN APP.JS jsx  CopyEdit  import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';  import HomePage from './pages/HomePage';  import CartPage from './pages/CartPage';  import LoginPage from './pages/LoginPage';  function App() {  return (  <Router>  <Header />  <Routes>  <Route path="/" element={<HomePage />} />  <Route path="/cart" element={<CartPage />} />  <Route path="/login" element={<LoginPage />} />  </Routes>  <Footer />  </Router>  );  } SUMMARY  | **Folder/File** | **Purpose** | | --- | --- | | public/ | Static files (HTML, favicon) | | src/ | Main React app code | | components/ | Reusable UI components | | pages/ | Full-page views | | services/ | API communication logic | | context/ | Global state (like auth/cart) | | styles/ | CSS for styling | | index.js | Entry point | | App.js | Routing and layout | |  |  |   **SERVER:** ****ORGANIZATION OF NODE.JS BACKEND**** The backend is built using **Node.js** with the **Express.js framework**, and it follows a clean, modular structure that separates responsibilities like routing, business logic, and database models. ****TYPICAL FOLDER STRUCTURE:**** bash  CopyEdit  backend/  │  ├── server.js # Main entry point of the server  ├── .env # Environment variables  ├── config/  │ └── db.js # MongoDB connection setup  │  ├── models/  │ ├── User.js # User schema  │ ├── Product.js # Product schema  │ └── Order.js # Order schema  │  ├── routes/  │ ├── userRoutes.js # Routes related to user actions (login, register)  │ ├── productRoutes.js # Routes for product listing and management  │ └── orderRoutes.js # Routes for placing and managing orders  │  ├── controllers/  │ ├── userController.js # Logic for user login/register  │ ├── productController.js # Logic for product operations  │ └── orderController.js # Logic for order processing  │  ├── middleware/  │ ├── authMiddleware.js # Checks if user is logged in/admin  │  ├── package.json # Lists all backend dependencies  └── README.md # Documentation ****EXPLANATION OF EACH PART:****server.js  * Main file that starts the Express server. * Connects to MongoDB. * Uses routes, middleware, etc.   js  CopyEdit  const express = require('express');  const connectDB = require('./config/db');  connectDB();  const app = express();  app.use('/api/users', require('./routes/userRoutes')); config/db.js  * Sets up and exports the MongoDB connection using **Mongoose**.   js  CopyEdit  const mongoose = require('mongoose');  const connectDB = async () => {  await mongoose.connect(process.env.MONGO\_URI);  };  module.exports = connectDB; models/  * Defines **MongoDB schemas** using Mongoose. * Example: Product.js, User.js, Order.js.  routes/  * Defines **API endpoints** like:   + GET /api/products   + POST /api/login   + POST /api/orders * These are mapped to controller functions.  controllers/  * Contains the **business logic** for each route. * Keeps route files clean.   js  CopyEdit  const registerUser = async (req, res) => {  // Validate and save new user  }; middleware/  * Middleware functions that run before controllers. * Example: check if a user is authenticated using JWT.  .env  * Stores **sensitive data** like:   env  CopyEdit  MONGO\_URI=mongodb+srv://...  JWT\_SECRET=mysecret  PORT=5000 ****BACKEND FLOW EXAMPLE:****  1. React calls POST /api/login 2. Route in userRoutes.js is triggered 3. userController.js handles logic 4. User data is fetched using User.js model 5. JWT token is returned as response  ****WHY THIS ORGANIZATION IS GOOD?****  * **Separation of Concerns**: Logic, routing, and DB are clearly separated. * **Easier Debugging and Scaling**. * **Cleaner Code and Reusability**.   **6.RUNNING THE APPLICATION:** ****1. START THE BACKEND SERVER (NODE.JS + EXPRESS)****STEP-BY-STEP: bash  CopyEdit  # Go to the backend folder  cd backend  # Install all backend dependencies  npm install  # Start the server  npm run start  If you're using **Nodemon** for auto-reload in development:  bash  CopyEdit  npm run dev ****2. START THE FRONTEND SERVER (REACT.JS)****STEP-BY-STEP: bash  CopyEdit  # Open a new terminal  # Go to the frontend folder  cd frontend  # Install all frontend dependencies  npm install  # Start the React app  npm start ****RESULT:****  * **Frontend** runs at: http://localhost:3000 * **Backend API** runs at: http://localhost:5000 (or the port defined in your .env)   **7.API DOCUMENTATION:** ****BACKEND API ENDPOINTS DOCUMENTATION**** Base URL: http://localhost:5000/api ****USER ROUTES (****/api/users****)****  | **Method** | **Endpoint** | **Description** | **Access** | | --- | --- | --- | --- | | POST | /register | Register a new user | Public | | POST | /login | Login user and return JWT | Public | | GET | /profile | Get user profile details | Private |  ****PRODUCT ROUTES (****/api/products****)****  | **Method** | **Endpoint** | **Description** | **Access** | | --- | --- | --- | --- | | GET | / | Get all products | Public | | GET | /:id | Get single product by ID | Public | | POST | / | Add a new product | Admin Only | | PUT | /:id | Update an existing product | Admin Only | | DELETE | /:id | Delete a product | Admin Only |  ****ORDER ROUTES (****/api/orders****)****  | **Method** | **Endpoint** | **Description** | **Access** | | --- | --- | --- | --- | | POST | / | Place a new order | Private | | GET | /myorders | Get all orders of the logged-in user | Private | | GET | / | Get all orders (admin only) | Admin Only | | PUT | /:id | Update order status (e.g., delivered) | Admin Only |  ****PROTECTED ROUTES**** Routes like /profile, /orders, /products (POST/PUT/DELETE) require a valid **JWT token** in the Authorization header:  makefile  CopyEdit  Authorization: Bearer <your\_token> ****EXAMPLE API REQUEST:**** **POST /api/users/login**  json  CopyEdit  Request Body:  {  "email": "user@example.com",  "password": "123456"  }  **RESPONSE:**  json  CopyEdit  {  "token": "xyz123...",  "user": {  "id": "abc123",  "name": "User",  "email": "user@example.com"  }  } ****COMPLETE API ENDPOINT DOCUMENTATION**** **Base URL:** http://localhost:5000/api ****USER ROUTES**** — /api/users****REGISTER USER****  * **Method:** POST * **Endpoint:** /api/users/register * **Request Body:**   json  CopyEdit  {  "name": "Lakshmi",  "email": "lakshmi@example.com",  "password": "123456"  }   * **RESPONSE:**   json  CopyEdit  {  "token": "jwt\_token\_here",  "user": {  "id": "123abc",  "name": "Lakshmi",  "email": "lakshmi@example.com"  }  } ****LOGIN USER****  * **Method:** POST * **Endpoint:** /api/users/login * **Request Body:**   json  CopyEdit  {  "email": "lakshmi@example.com",  "password": "123456"  }   * **RESPONSE:**   json  CopyEdit  {  "token": "jwt\_token\_here",  "user": {  "id": "123abc",  "name": "Lakshmi",  "email": "lakshmi@example.com"  }  } ****GET USER PROFILE****  * **Method:** GET * **Endpoint:** /api/users/profile * **Headers:** Authorization: Bearer <jwt\_token> * **Response:**   json  CopyEdit  {  "id": "123abc",  "name": "Lakshmi",  "email": "lakshmi@example.com"  } ****PRODUCT ROUTES**** — /api/products****GET ALL PRODUCTS****  * **Method:** GET * **Endpoint:** /api/products * **Response:**   json  CopyEdit  [  {  "\_id": "p1",  "name": "Rice",  "price": 40,  "category": "Grains",  "stock": 100  },  {  "\_id": "p2",  "name": "Milk",  "price": 25,  "category": "Dairy",  "stock": 50  }  ] ****GET PRODUCT BY ID****  * **Method:** GET * **Endpoint:** /api/products/:id * **Example:** /api/products/p1 * **Response:**   json  CopyEdit  {  "\_id": "p1",  "name": "Rice",  "price": 40,  "category": "Grains",  "stock": 100  } ****ADD NEW PRODUCT****  * **Method:** POST * **Endpoint:** /api/products * **Headers:** Authorization: Bearer <admin\_token> * **Request Body:**   json  CopyEdit  {  "name": "Oil",  "price": 90,  "category": "Grocery",  "stock": 200  }   * **RESPONSE:**   json  CopyEdit  {  "\_id": "newProductId",  "name": "Oil",  "price": 90,  "category": "Grocery",  "stock": 200  } ****UPDATE PRODUCT****  * **Method:** PUT * **Endpoint:** /api/products/:id * **Headers:** Authorization: Bearer <admin\_token> * **Request Body:**   json  CopyEdit  {  "price": 95,  "stock": 180  }   * **RESPONSE:**   json  CopyEdit  {  "message": "Product updated successfully"  } ****DELETE PRODUCT****  * **Method:** DELETE * **Endpoint:** /api/products/:id * **Headers:** Authorization: Bearer <admin\_token> * **Response:**   json  CopyEdit  {  "message": "Product deleted successfully"  } ****ORDER ROUTES**** — /api/orders****PLACE ORDER****  * **Method:** POST * **Endpoint:** /api/orders * **Headers:** Authorization: Bearer <user\_token> * **Request Body:**   json  CopyEdit  {  "items": [  {  "product": "p1",  "quantity": 2  }  ],  "address": "123 Street, City"  }   * **RESPONSE:**   json  CopyEdit  {  "\_id": "orderId123",  "status": "Pending",  "user": "userId123"  } ****GET USER'S ORDERS****  * **Method:** GET * **Endpoint:** /api/orders/myorders * **Headers:** Authorization: Bearer <user\_token> * **Response:**   json  CopyEdit  [  {  "\_id": "orderId123",  "items": [{ "product": "p1", "quantity": 2 }],  "status": "Pending"  }  ] ****GET ALL ORDERS (ADMIN)****  * **Method:** GET * **Endpoint:** /api/orders * **Headers:** Authorization: Bearer <admin\_token> * **Response:**   json  CopyEdit  [  {  "\_id": "orderId123",  "user": "Lakshmi",  "status": "Pending",  "address": "123 Street"  }  ] ****UPDATE ORDER STATUS****  * **Method:** PUT * **Endpoint:** /api/orders/:id * **Headers:** Authorization: Bearer <admin\_token> * **Request Body:**   json  CopyEdit  {  "status": "Delivered"  }   * **RESPONSE:**   json  CopyEdit  {  "message": "Order status updated"  }  **NOTE ON AUTHORIZATION**   * **JWT token** must be included in headers for protected routes:   makefile  CopyEdit  Authorization: Bearer your\_token\_here  **8.AUTHENTICATION** ****AUTHENTICATION & AUTHORIZATION IN SHOPSMART********WHAT IS AUTHENTICATION?**** Authentication means **verifying who the user is** (e.g., login using email and password). ****WHAT IS AUTHORIZATION?**** Authorization means **checking what the user is allowed to do** (e.g., only admin can add or delete products). ****HOW IT WORKS IN SHOPSMART****1. ****USER REGISTRATION****  * New users register by providing:   + Name, Email, Password * The password is **encrypted using bcrypt** before saving in the database.  2. ****USER LOGIN****  * User logs in using email and password. * If valid, the backend:   + **Verifies the password**   + **Generates a JWT (JSON Web Token)**  ****JWT TOKEN (JSON WEB TOKEN)****  * A secure token string (e.g., abc123xyz...) that contains user info. * Sent back to the frontend after login.   json  CopyEdit  {  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",  "user": {  "id": "123abc",  "name": "Lakshmi",  "email": "lakshmi@example.com"  }  } 3. ****FRONTEND STORES TOKEN****  * React frontend stores the token in:   + **Local storage** or **session storage** * This token is included in the **Authorization header** of every protected request.  4. ****BACKEND VALIDATES TOKEN****  * Backend uses a **middleware function** to:   + Check if token is valid   + Decode token to get user info   + Allow or deny access based on role (user/admin)   EXAMPLE MIDDLEWARE (AUTHMIDDLEWARE.JS):  js  CopyEdit  const jwt = require("jsonwebtoken");  const authMiddleware = (req, res, next) => {  const token = req.headers.authorization?.split(" ")[1];  if (!token) return res.status(401).json({ message: "Not authorized" });  const decoded = jwt.verify(token, process.env.JWT\_SECRET);  req.user = decoded;  next();  }; 5. ****ROLE-BASED AUTHORIZATION****  * After validating the token, backend checks:   + Is the user an admin?     - Yes → Can access admin routes (e.g., add products)     - No → Block admin-only actions  ****SUMMARY TABLE****  | **Step** | **Description** | | --- | --- | | 1 | User registers (password hashed with bcrypt) | | 2 | User logs in and gets a JWT token | | 3 | Token is stored in frontend (localStorage) | | 4 | Token sent in request headers (Authorization: Bearer <token>) | | 5 | Backend verifies token and user role | |  | Allows or denies access to protected routes |   **9.USER INTERFACE**  LANDING PAGE:  landingpage.png |  |

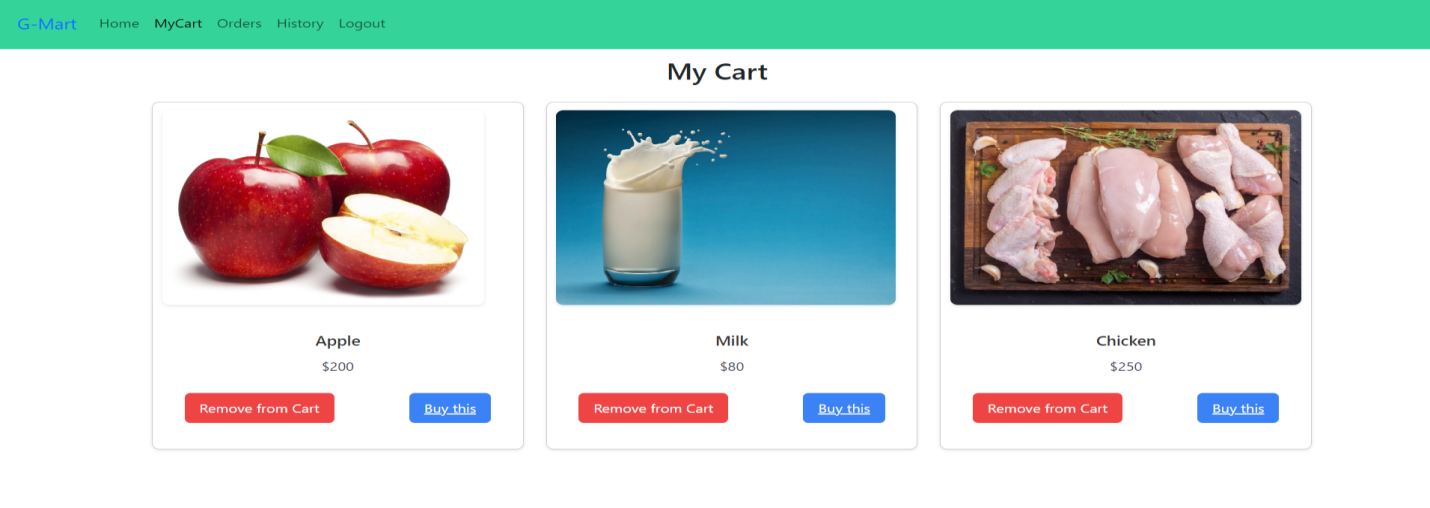
**LOGIN PAGE:**



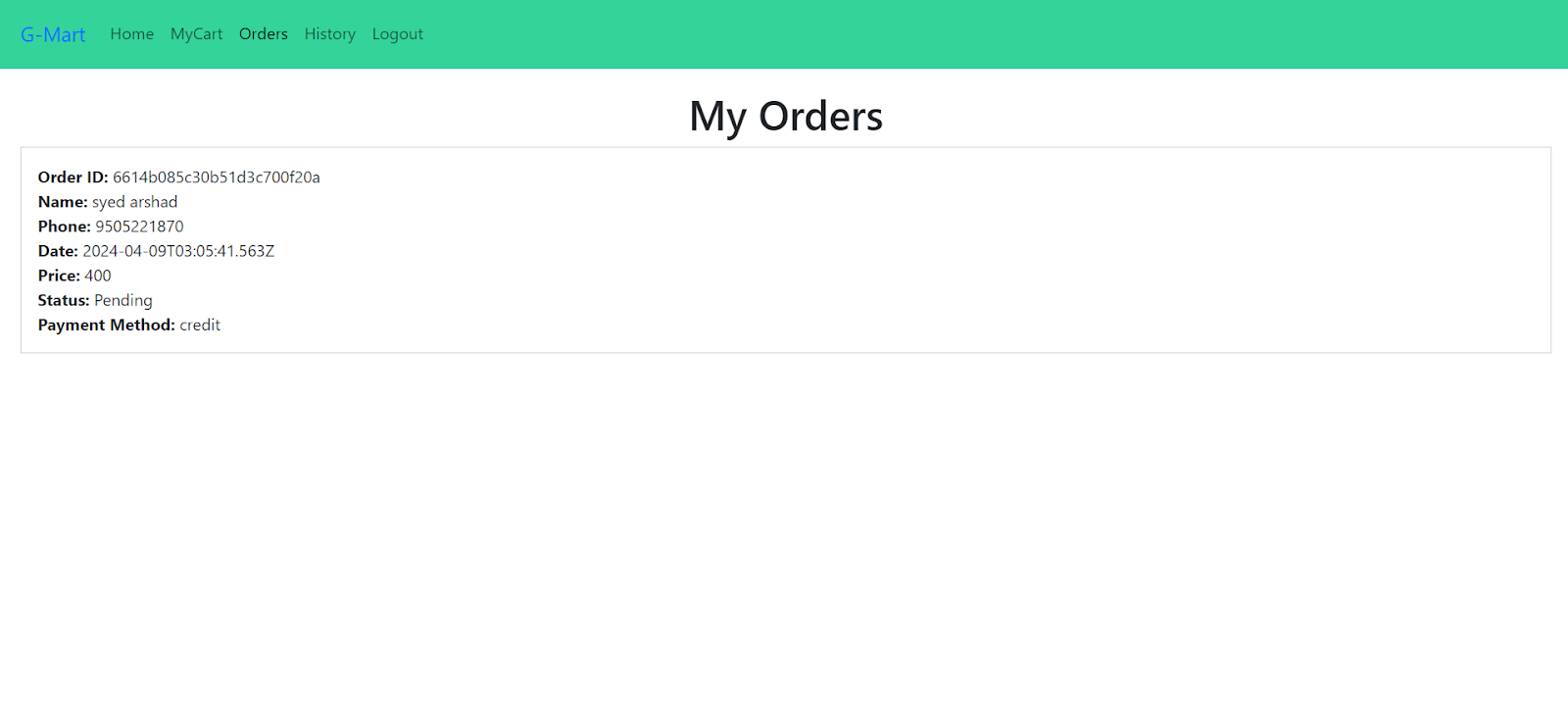
**ITEMS PAGE**

****

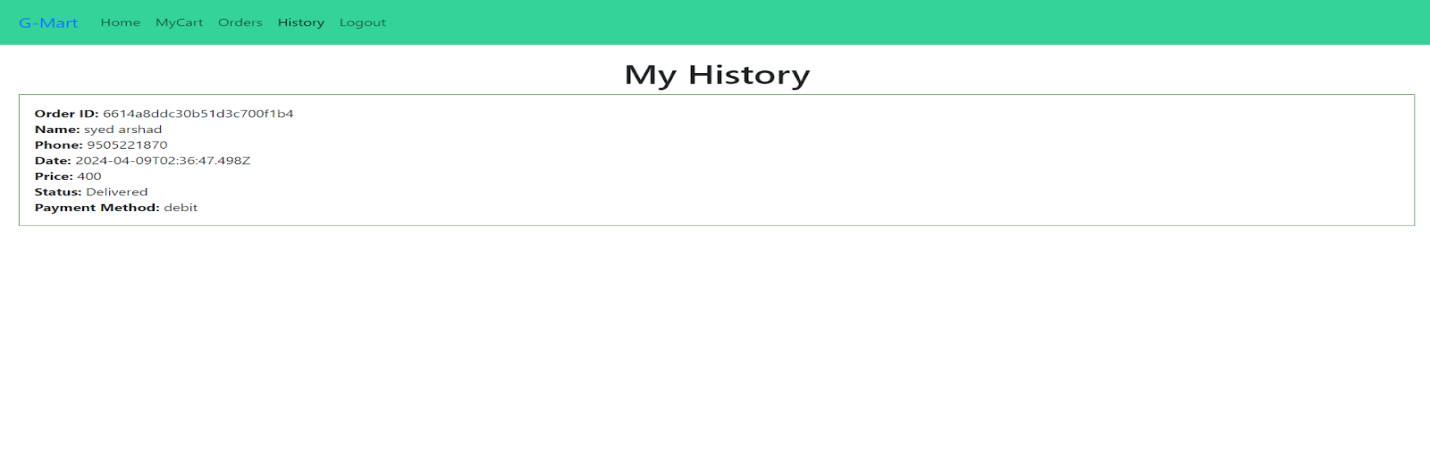
**MY CART:**

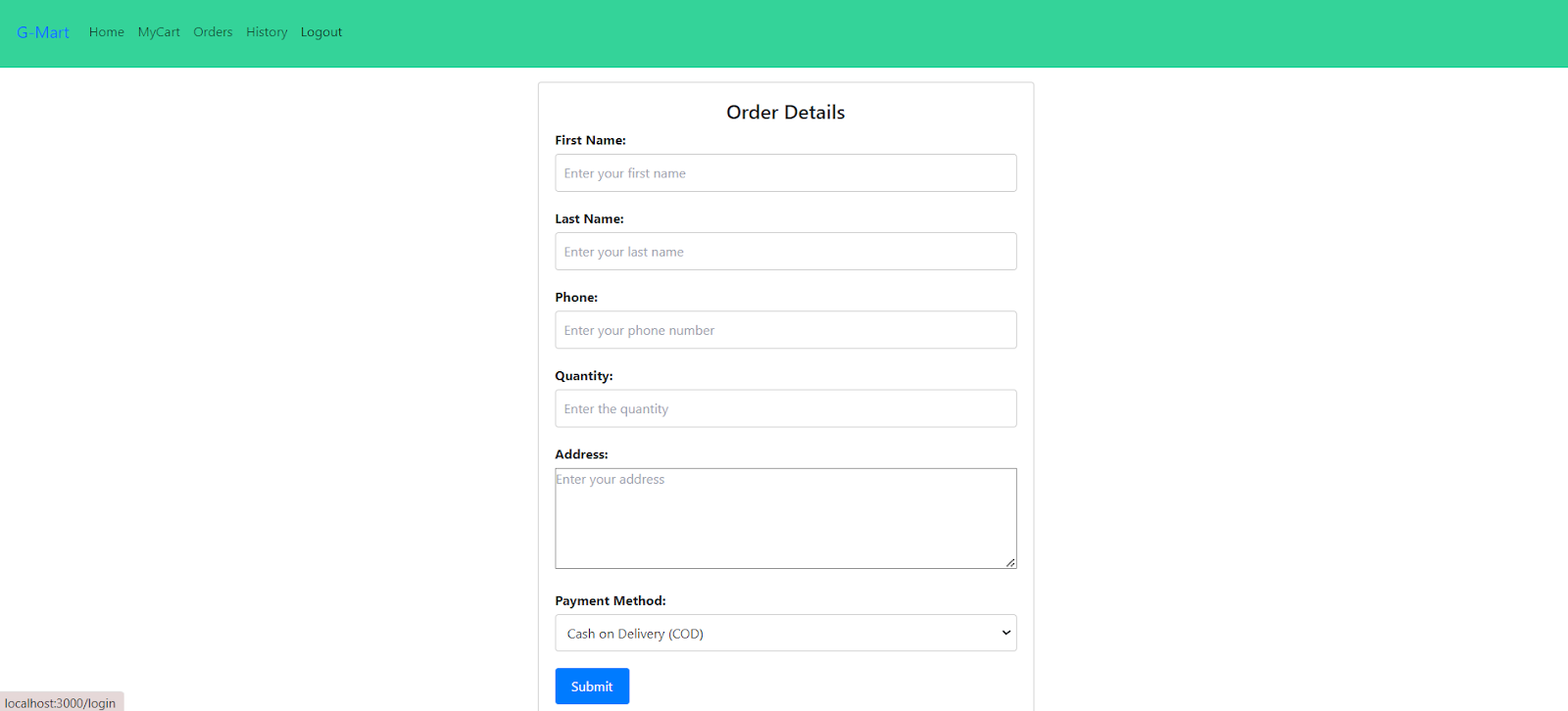
****

MY ORDERS:

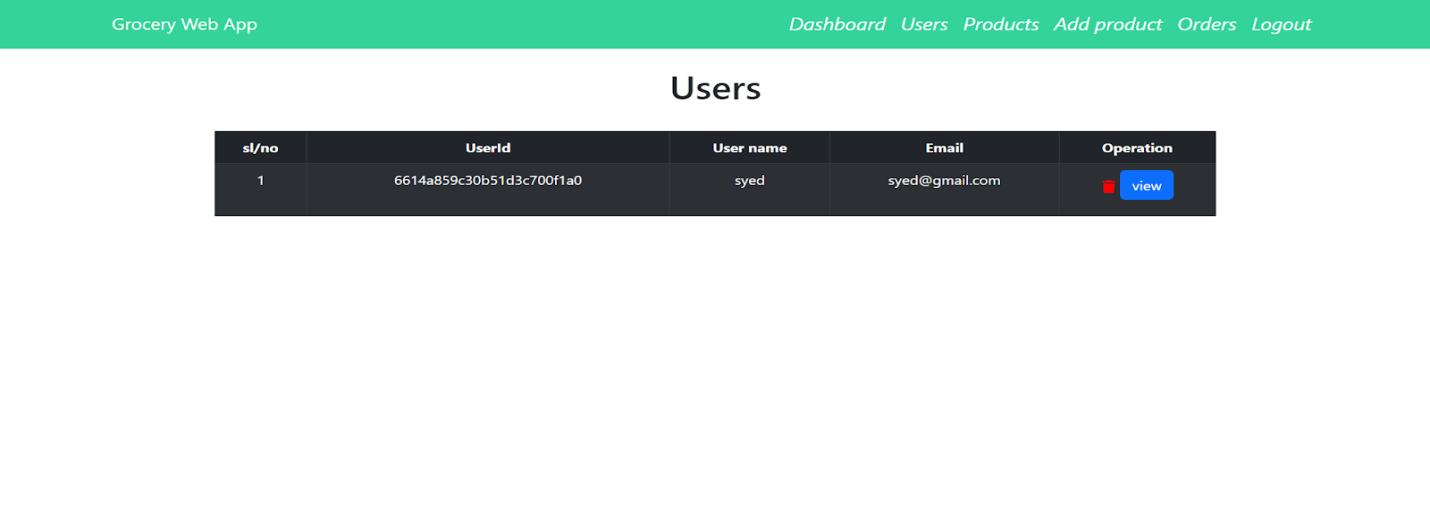


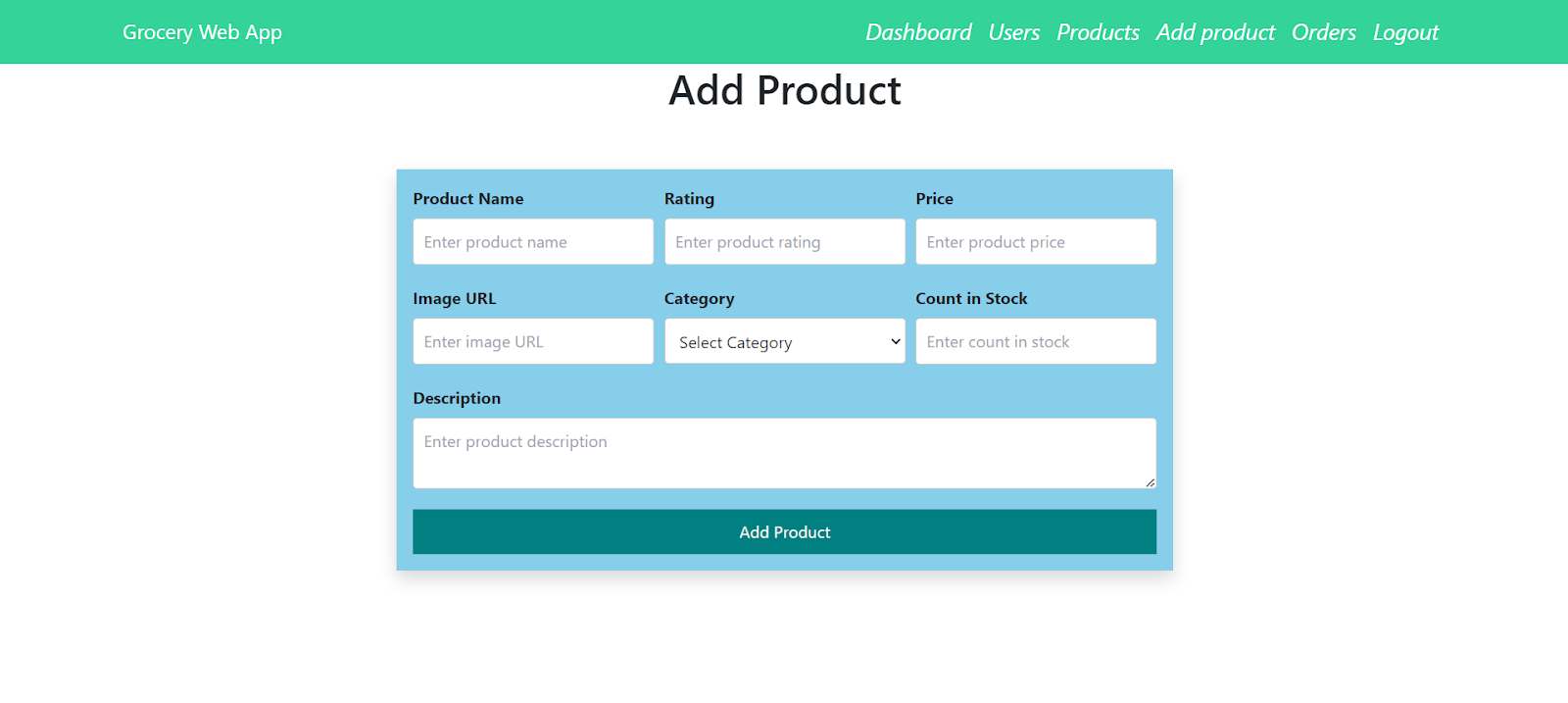
MY HISTORY:

PLACE ORDER:

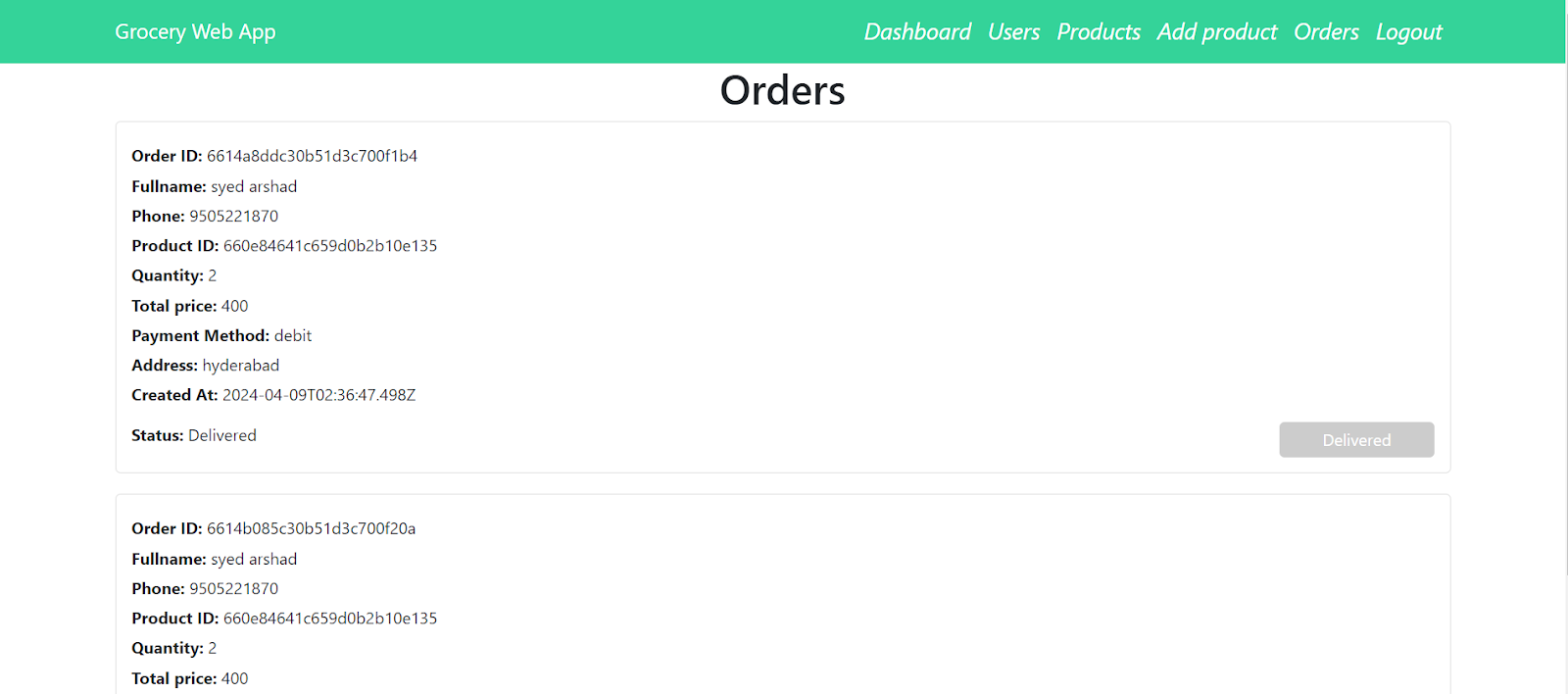


USERS PAGE:

ADD PRODUCT PAGE:



ADMIN ORDERS PAGE:



**10.TESTING**

## ****TESTING STRATEGY****

The goal of testing in ShopSmart is to make sure that both **frontend** and **backend** work correctly and smoothly, without bugs.

We followed **manual testing** for this project with the help of **common tools** and **step-by-step checking** of each feature.

### ****1. MANUAL TESTING****

We tested all major functionalities manually by:

* Entering different inputs
* Performing user actions like:
  + Login
  + Add to cart
  + Place order
  + Admin product management
* Checking responses and UI behavior

### ****2. TOOLS USED FOR TESTING****

| **Tool** | **Purpose** |
| --- | --- |
| **Postman** | To test backend APIs (like login, fetch products, place orders) |
| **Browser (Chrome)** | To test the frontend UI and how it interacts with backend |
| **Console/Inspect Tool** | To view errors, warnings, and network requests |
| **MongoDB Compass** | To verify data saved in the database after operations |

### ****3. WHAT WE TESTED****

#### ****USER FEATURES:****

* Register and Login
* Token generation and storage
* Product browsing
* Add to cart, checkout
* Order history

#### ****ADMIN FEATURES:****

* Add/Edit/Delete products
* View all orders
* Update order status

#### ****API TESTING WITH POSTMAN:****

* Checked all endpoints like:
  + POST /api/users/login
  + GET /api/products
  + POST /api/orders
* Validated inputs and outputs

### ****BUG HANDLING****

If we found issues during testing:

1. We checked the browser console or Postman error.
2. Debugged the backend or frontend code.
3. Retested the fixed part.

## ****SUMMARY****

| **Type of Testing** | **Tools Used** | **What Was Tested** |
| --- | --- | --- |
| Manual Testing | Chrome, VS Code | UI flow and input handling |
| API Testing | Postman | Backend endpoints and responses |
| Database Checking | MongoDB Compass | Whether data is correctly stored |
| Debugging | Console/Inspect | Error handling and flow control |

**11.KNOWN ISSUES**

## ****KNOWN BUGS & ISSUES****

### 1. ****CART NOT UPDATING ON PAGE REFRESH****

* **Issue:** Items added to the cart disappear after refreshing the page.
* **Cause:** Cart data is stored in local state only, not saved in localStorage or backend.
* **Fix Suggestion:** Save cart data to localStorage or database.

### 2. ****NO PASSWORD STRENGTH VALIDATION****

* **Issue:** Users can register with weak passwords (e.g., "123").
* **Cause:** No password rules implemented.
* **Fix Suggestion:** Add frontend and backend password validation rules.

### 3. ****PRODUCT IMAGE UPLOAD NOT FUNCTIONAL****

* **Issue:** Admin cannot upload real product images.
* **Cause:** Image upload feature is not implemented or only placeholder is used.
* **Fix Suggestion:** Integrate image upload with cloud services (like Cloudinary or Firebase).

### 4. ****LAYOUT BREAKS ON SMALL SCREENS****

* **Issue:** Some UI elements (like buttons or cards) overflow on small mobile devices.
* **Cause:** Missing responsive CSS for small breakpoints.
* **Fix Suggestion:** Use media queries or responsive classes (e.g., Bootstrap/Tailwind).

### 5. ****NO LOADING INDICATOR****

* **Issue:** User sees nothing or blank screen when data is loading.
* **Cause:** No spinner or loading message is shown.
* **Fix Suggestion:** Add a simple loader (e.g., "Loading...") during API calls.

### 6. ****ORDER HISTORY MISSING PAGINATION****

* **Issue:** Users with many orders see all of them at once, causing lag.
* **Cause:** No pagination or limit set in API.
* **Fix Suggestion:** Implement pagination on backend and frontend.

### 7. ****ERROR MESSAGES ARE GENERIC****

* **Issue:** Users see vague error messages like "Something went wrong".
* **Cause:** Errors are not properly handled or shown.
* **Fix Suggestion:** Add clear error messages (e.g., “Email already exists”, “Invalid password”).

## ****NOTE FOR DEVELOPERS****

These issues are **common in early-stage projects** and can be fixed as you continue development or during the next sprint.

**12.FUTURE ENHANCEMENTS:**

## ****FUTURE FEATURES & IMPROVEMENTS****

### 1. ****ONLINE PAYMENT INTEGRATION****

* **What:** Allow users to pay via UPI, cards, or net banking.
* **Why:** Adds real e-commerce experience.

### 2. ****LIVE ORDER TRACKING****

* **What:** Show real-time order status (e.g., Packed, Shipped, Delivered).
* **Why:** Helps users stay informed and builds trust.

### 3. ****PROGRESSIVE WEB APP (PWA)****

* **What:** Make the app installable like a mobile app.
* **Why:** Improves mobile experience and offline support.

### 4. ****PRODUCT RATINGS AND REVIEWS****

* **What:** Let users rate and review products.
* **Why:** Increases product reliability and user engagement.

### 5. ****ADMIN ANALYTICS DASHBOARD****

* **What:** Show sales reports, most ordered items, user stats.
* **Why:** Helps admin make better business decisions.

### 6. ****EMAIL & SMS NOTIFICATIONS****

* **What:** Notify users when order is placed, shipped, or delivered.
* **Why:** Keeps users updated without logging in.

### 7. ****ADVANCED SEARCH AND FILTERS****

* **What:** Add filters by brand, price range, availability, etc.
* **Why:** Makes product discovery easier for users.

### 8. ****LIVE CHAT SUPPORT****

* **What:** Chat system for users to ask questions.
* **Why:** Improves customer service experience.

### 9. ****SAVE FOR LATER / WISHLIST****

* **What:** Users can save items to buy later.
* **Why:** Helps users plan future purchases.

### 10. ****MULTI-LANGUAGE SUPPORT****

* **What:** Provide the app in multiple languages.
* **Why:** Makes the app accessible to a wider audience.