

## Project Title

# ShopSmart: Your Digital Grocery Store Experience

## Team Members:

Here List team members and their roles:

- 1. G. SRIHARI (Full Stack Developer):** Combines both frontend and backend responsibilities, ensuring smooth communication between the two. This role also handles bug fixing, feature integration, and overall system performance.
- 2. CH. JAHNAVI (Frontend Developer):** Responsible for designing the user interface using **React.js**. This role focuses on ensuring a responsive, user-friendly design, as well as integrating the frontend with backend APIs.
- 3. G. NAVEEN KUMAR (Backend Developer):** Develops the backend server using **Node.js** and **Express.js**, ensuring the creation of secure, scalable RESTful APIs, as well as handling authentication, data processing, and business logic.
- 4. I. CHARAN AJAY (Database Administration):** Manages the **MongoDB** database, focusing on schema design, data integrity, and database optimization to ensure efficient data storage and retrieval.

## 1. Introduction

**Our** basic grocery-web app! Our app is designed to provide a seamless online shopping experience for customers, making it convenient for them to explore and purchase a wide range of products. Whether you are a tech enthusiast, a fashionista, or a homemaker looking for everyday essentials, our app has something for everyone. With user-friendly navigation and intuitive design, our grocery-webapp app allows customers to browse through various categories, view product details, add items to their cart, and securely complete the checkout process. We prioritize user satisfaction and aim to provide a smooth and hassle-free shopping experience. For sellers and administrators, our app offers robust backend functionalities. Sellers can easily manage their product listings, inventory, and orders, while administrators can efficiently handle customer inquiries, process payments, and monitor overall app performance. With a focus on security and privacy, our grocery-webapp app ensures that customer data is protected, transactions are secure, and personal information remains confidential. We strive to build trust with our customers and provide a safe platform for online shopping. We are excited to have you on board and look forward to providing you with a delightful shopping experience. Happy shopping with our grocery-webapp!

### Project Idea:

ShopSmart is an innovative e-commerce application designed to revolutionize the grocery shopping experience. The platform offers a seamless, user-friendly interface where customers can browse a wide variety of groceries, add items to their cart, and complete their purchase in just a few clicks. By providing real-time product availability, personalized shopping experiences, and secure payment options, ShopSmart aims to bridge the gap between traditional grocery stores and modern digital conveniences. It not only benefits end-users but also provides grocery store owners with an efficient way to manage inventories and orders.

### Motivation:

With the increasing reliance on online services, traditional grocery shopping has transformed. Consumers seek convenience, personalized experiences, and reliable delivery options. ShopSmart is designed to address this by offering a seamless digital grocery shopping experience. It enables users to browse a wide range of products, add items to a virtual cart, and make secure payments, all from the comfort of their homes.

## 2. Project Overview

### Purpose:

The purpose of ShopSmart is to offer a comprehensive platform for digital grocery shopping. The application allows users to easily browse products, view details, add items to a cart, and place orders. It simplifies inventory management for store administrators and enhances the shopping experience for customers with a user-friendly interface.

### Goals and objectives

1. **Convenience:** Provide an easy-to-use platform that allows customers to shop for groceries online, saving them time and effort.
2. **Real-Time Updates:** Ensure real-time product availability, so customers know what is in stock before placing an order.
3. **Secure Transactions:** Integrate secure payment gateways for safe and quick transactions.
4. **Efficient Inventory Management:** Offer store owners an administrative tool to manage stock levels, product listings, and customer orders seamlessly.

5. **Scalability:** Build the platform with scalability in mind to handle increased user traffic and expand the product range without performance issues.
6. **Customer Satisfaction:** Focus on providing a high-quality user experience through personalized shopping suggestions and features like order tracking.

### **Key Features and Functionalities:**

Key features of the proposed system include:

#### **1. Product Browsing and Search:**

- Users can easily search and filter products by category, name, brand, or price range.
- Real-time product search suggestions for a smoother shopping experience.

#### **2. Smart Shopping Cart:**

- Users can add, update, or remove items from their shopping cart.
- The cart dynamically updates pricing and total costs as items are added or removed.

#### **3. Product Details Page:**

- Each product has a detailed page displaying product images, descriptions, pricing, and availability.
- Option to view related or recommended products based on user behavior.

#### **4. Secure User Authentication:**

- User registration and login with encrypted passwords.
- JWT-based authentication ensures secure access to protected routes.
- Persistent login sessions until the user logs out.

#### **5. Order Management:**

- Users can place orders, review order summaries, and track the status of their orders.
- Order history allows users to view and manage previous orders, including re-ordering items.

#### **6. Payment Integration:**

- Integrated with Stripe for secure online payments.
- Supports various payment methods including credit/debit cards.
- Real-time order confirmation after payment completion.

#### **7. Admin Dashboard:**

- Admins can manage product listings, inventory levels, and pricing.
- Ability to view and manage user orders, including processing, shipping, and cancellations.

#### **8. User Profile Management:**

- Users can update their profile information, including delivery addresses and contact details.
- Order history and personalized recommendations based on past purchases.

## 9. Responsive and Mobile-Friendly Design:

- Fully responsive design that works seamlessly across mobile, tablet, and desktop devices.
- Optimized for performance with minimal loading times and smooth navigation.

## 10. Notifications and Alerts:

- Real-time notifications for order confirmations, shipping updates, and stock alerts

## Challenges Addressed:

- **Time-Consuming Shopping:** Eliminates the need for physical store visits.
- **Lack of Personalization:** Offers a personalized experience by saving preferences and past orders.
- **Order Tracking:** Users can track the status of their orders in real-time.

## 3. Architecture

### 1. Frontend Architecture:

The frontend architecture of ShopSmart: Your Digital Grocery Store Experience is built with React.js, providing a modular, component-based structure that ensures high scalability and maintainability. Here's a detailed description of how the frontend is organized:

#### ➤ Component-Based Structure:

- **React Components:** The entire UI is divided into reusable components, each responsible for rendering a specific part of the user interface (e.g., Productcard, Cart, Header, Footer).
  - **Functional Components:** Most of the components in ShopSmart are functional, utilizing React Hooks to manage state and side effects.
  - **Reusable UI Components:** Components like buttons, input fields, and product cards are built to be reused across different pages, ensuring code efficiency and consistency in design.
- **Component Hierarchy:**
  - **Top-Level Components** (e.g., App.js) handle routing and major layout components.
  - **Page Components** (e.g., Home.js, ProductPage.js) represent the different sections of the application like the home page, product page, cart page, etc.
  - **Child Components** (e.g., ProductCard.js, CartItem.js) represent individual elements that are embedded within pages.
- **State Management with Redux:**

- **Global State:** State management is handled using **Redux**, allowing global states like user information, cart data, and product lists to be shared across the application.
- **Actions and Reducers:**
  - **Actions:** Redux actions are dispatched to update the global state, such as adding items to the cart, updating user details, or fetching product data.
  - **Reducers:** Reducers are pure functions that handle changes in the global state based on the dispatched actions. For example, the `cartReducer` manages all state changes related to cart operations.
- **Redux Toolkit:** To streamline Redux operations, **Redux Toolkit** is used for defining slices and managing complex state logic more efficiently.
- **React Router for Navigation:**
  - **Client-Side Routing:** **React Router** is implemented to handle navigation between different pages (e.g., Home, Products, Cart, Checkout) without refreshing the entire page.
  - **Protected Routes:** Some routes, such as the **User Profile** or **Checkout** pages, are protected and accessible only to authenticated users. These routes are implemented using custom React Router components that check the authentication state.
- **React Hooks:**
  - **useState and useEffect:** The project makes extensive use of React Hooks like `useState` for managing local component states and `useEffect` for handling side effects such as API calls to fetch data.
  - **Custom Hooks:** Some reusable logic, like form validation or fetching data from the API, is abstracted into custom hooks, making the codebase more modular and reusable.
- **API Integration:**
  - **Asynchronous Data Fetching:** The frontend communicates with the backend through RESTful APIs. **Fetch API** is used to make asynchronous requests to the backend for fetching products, managing the cart, and submitting orders.
  - **Error Handling:** The application includes centralized error handling for failed API calls, ensuring smooth user experience by displaying error messages or fallback UI components.
- **Styling and Design:**
  - **CSS and Tailwind:** **Tailwind** is used for the overall grid system and responsive layout design. Custom **CSS** modules are applied for styling specific components to maintain a consistent look and feel.
  - **Responsive Design:** The application is designed to be fully responsive, ensuring a seamless user experience across different devices (desktops, tablets, smartphones). Media queries are used to adjust layout and font sizes based on screen dimensions.
- **Form Handling and Validation:**
  - **Controlled Components:** Form inputs are managed as controlled components in React, ensuring that the form data is linked directly to the component state.

- **Performance Optimization:**

- **Lazy Loading:** **React.lazy()** is used to load components only when they are required (e.g., loading heavy components like the product page only when a user navigates to it), improving the initial load time.
- **Code Splitting:** Code splitting is implemented with **Webpack** to load only the essential JavaScript bundles needed for each page, improving overall application performance.

- **Third-Party Libraries:**

- **Stripe Integration:** **Stripe.js** is integrated into the frontend for handling secure payments during the checkout process. It includes the payment form and validation.
- **React Icons:** **React Icons** is used for including scalable icons across the application, enhancing visual appeal.

- **Testing with Jest and React Testing Library:**

- The frontend components are tested using **Jest** and **React Testing Library** to ensure they render correctly, function as expected, and respond properly to user interactions.
- **Snapshot Testing:** Snapshot tests are written to capture the component's initial render and ensure the UI does not change unexpectedly.

## 2. Backend Architecture:

The backend architecture of **ShopSmart: Your Digital Grocery Store Experience** is designed to be scalable, maintainable, and secure. It uses **Node.js** as the runtime environment and **Express.js** as the web framework. Here's a detailed overview of the backend architecture:

- **Overall Architecture**

- **Server-Side Framework:** The backend is built using **Express.js**, which simplifies the development of server-side applications by providing a robust set of features for building web applications and APIs.
- **Middleware Architecture:** Express.js employs a middleware architecture, allowing for modular and reusable code that handles requests and responses through a series of middleware functions.

- **API Routes for Shopsmart**

- **User Routes:**

1. **POST /signup:** Register a new user.
2. **POST /login:** User login.
3. **GET /allusers:** Fetch all user.
4. **GET /user:** Fetch user details.

- **Product Routes:**

1. **GET /product:** Retrieve all products.
2. **POST /uploadProduct:** Add a new product (admin only).

- **Order Routes:**

1. **POST /orders:** Place a new order.
2. **GET /orders/userId:** Fetch all orders for a specific user.
3. **GET /orders:** Fetch all orders.
4. **GET /my-orders:** Fetch the order details for customer.
5. **PUT /orders/:id:** Update order status (admin only).

- **Environment Configuration**

- **dotenv Package:** The application uses the dotenv package to load environment variables from a .env file, enabling configuration for different environments (development, testing, production). This includes database connection strings, JWT secret keys, and API keys for third-party services.

### 3. Database Architecture:

The ShopSmart application uses MongoDB as its database to store user, product, and order data. Below is a detailed overview of the database schema, including the collections, their structure, and how they interact within the application.

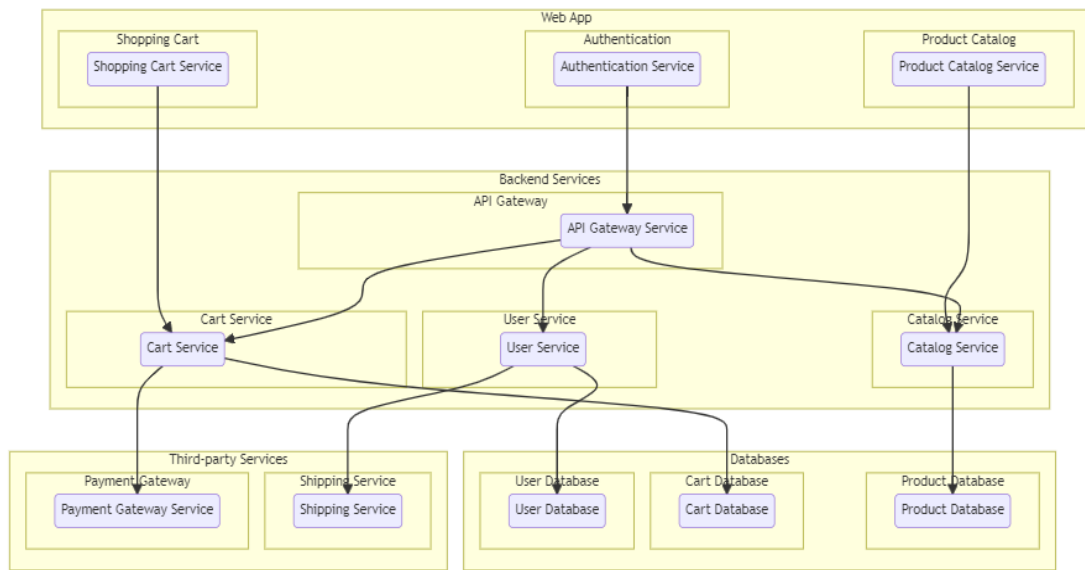
#### 1. Collections Overview:

The application consists of the following main collections:

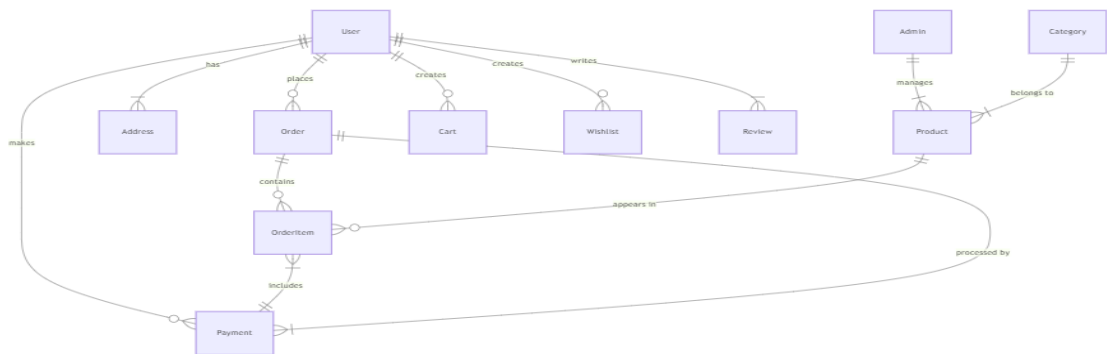
- **Users**
- **Products**
- **Orders**
- **Contacts**
- **Addresses**
- **Feedbacks**

#### 2. Data Relationships:

- **User-Order Relationship:** Each user document can reference multiple orders in the `orderList` field. The `Order` model contains a `userId` field to link each order to the corresponding user.
- **Order-Product Relationship:** Each order contains an array of product references, allowing easy access to product details for each item ordered.



**Fig: Technical Architecture**



**Fig: ER Diagram**



## 4. Setup Instructions

To develop a full-stack grocery web app using React.js, Node.js, and MongoDB, several **prerequisites** are necessary to **install**. Below are the key steps and tools required:

### 1. Node.js and npm

Node.js is essential for running JavaScript on the server side, and npm (Node Package Manager) is used for managing project dependencies.

- **Download Node.js:** [Download Node.js](#)
- **Installation Instructions:** [Install Node.js via Package Manager](#)

### 2. MongoDB

MongoDB is the NoSQL database used to store data such as user profiles, products, and orders. You can either install MongoDB locally or use a cloud-based MongoDB service like MongoDB Atlas.

- **Download MongoDB:** [Download MongoDB Community Edition](#)
- **Installation Instructions:** [MongoDB Installation Guide](#)

### 3. Express.js

Express.js is a web framework for Node.js that simplifies server-side development by providing tools for routing, middleware, and API development.

- **Install Express.js:** Open your terminal or command prompt and run the following command:  
`npm install express`

### 4. React.js

React.js is the JavaScript library used to build the frontend user interface. React enables the development of dynamic, component-based applications that allow for fast and responsive user experiences.

#### Steps to Set Up React:

##### 1. Create a New React Project:

- Install the Create React App tool, which sets up a new project with all required configurations:  
`npx create-react-app client`

##### 2. Navigate to the Project Directory:

`cd client`

##### 3. Start the React Development Server:

- Launch the development server by running:  
`npm start`
- Open your browser and go to `http://localhost:3000` to view your running React app.
- **React Documentation:** [Official React Docs](#)

## 5. Tailwind CSS for Styling

Tailwind CSS is a utility-first CSS framework that allows you to style components without writing custom CSS classes. It speeds up development by providing pre-designed components and styles.

### Steps to Set Up Tailwind CSS in React:

#### 1. Install Tailwind CSS:

`npm install -D tailwindcss`

#### 2. Initialize Tailwind CSS Configuration:

`npx tailwindcss init`

#### 3. Configure `tailwind.config.js`:

- Update the content array to specify which files Tailwind should scan for class names:

```
module.exports = {  
  content: [  
    "./src/**/*.{js,jsx,ts,tsx}",  
  ],  
  theme: {  
    extend: {},  
  },  
  plugins: [],  
}
```

#### 4. Add Tailwind to Your CSS:

- In the `src/index.css` file, include the following Tailwind imports:

```
@tailwind base;  
@tailwind components;  
@tailwind utilities;
```

#### 5. Run the React App:

- Tailwind CSS is now set up, and you can use its utility classes within your React components.
- Tailwind CSS Documentation:** [Tailwind CSS Docs](#)

## 6. HTML, CSS, and JavaScript

You need a solid understanding of HTML and CSS for structuring and styling the user interface, and JavaScript to handle client-side logic and interactivity in React.

## 7. Database Connectivity with Mongoose

Use Mongoose, an Object-Document Mapping (ODM) library for MongoDB, to connect your Node.js server with the database and perform CRUD (Create, Read, Update, Delete) operations.

- Install Mongoose:** `npm install mongoose`
- Mongoose Documentation:** <https://mongoosejs.com/docs/api/document.html>

## 8. Front-End Development with React.js

Use React.js to create the user interface (UI) of the grocery app. React components enable dynamic updates and interactivity in real-time.

- **Product Listings:** Display a list of available grocery items with images, descriptions, and prices.
- **Shopping Cart:** Allow users to add products to a shopping cart and modify item quantities.
- **User Forms:** Provide login, registration, and checkout forms.
- **Admin Dashboard:** Create a user-friendly interface for managing products, orders, and customers.

## 9. Version Control with Git

Use Git for version control, enabling collaboration, and tracking changes throughout the development process. Platforms like GitHub or Bitbucket can host your repository.

- **Download Git:** [Download Git](#)

## 10. Development Environment

Choose a code editor or Integrated Development Environment (IDE) for writing and managing code. Popular options include:

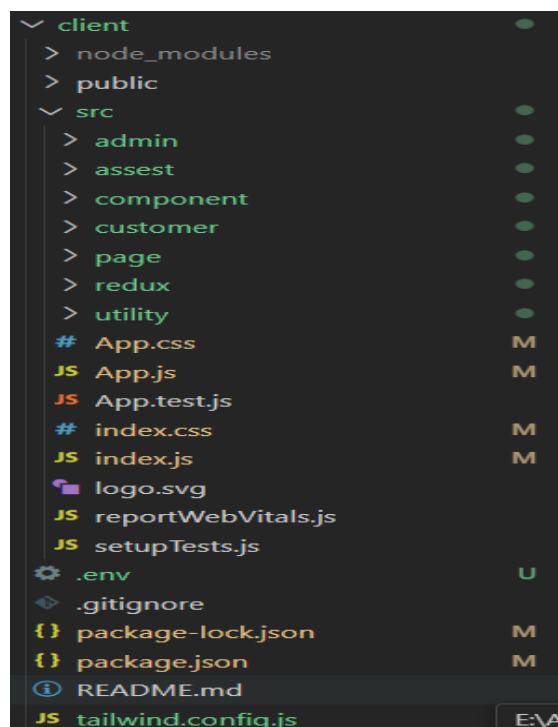
- **Visual Studio Code:** [Download VS Code](#)
- **Sublime Text:** [Download Sublime Text](#)
- **WebStorm:** [Download WebStorm](#)

## 5. Folder Structure

The core structure of the React frontend project typically looks like this:

/client

```
|— /node_modules
|— /public
|— /src
|   |— /admin
|   |— /assets
|   |— /components
|   |— /customer
|   |— /pages
|   |— /redux
|   |— /utility
|— App.css
```



```

├── App.test.js
├── App.js
├── index.js
├── index.css
├── package-lock.js
├── package.json
├── tailwind.config.js
└── .env

```

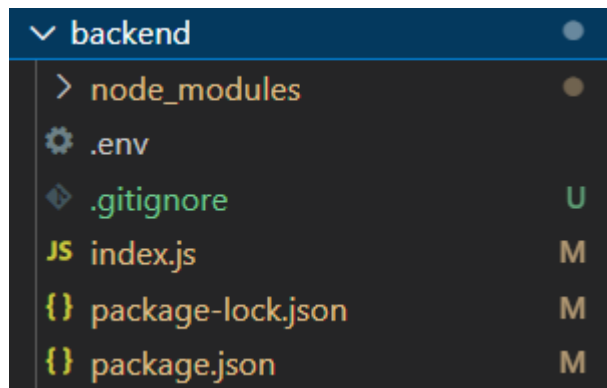
**The core structure of the React Backend project typically looks like this:**

**/backend**

```

├── node_modules
├── index.js
├── .env
├── package.json
├── package-lock.json
└── README.md

```



## 6. Running the Application

Running the Full-Stack Application:

### 1. Once both the frontend and backend are set up:

1. Start the Backend:
  - Navigate to the backend directory and run: **npm run dev**

```

PS E:\Applicatios\PROJECT524> cd .\backend
PS E:\Applicatios\PROJECT524\backend> npm run dev

> backend@1.0.0 dev
> node index.js

mongodb+srv://admin:admin524@project.1sisn.mongodb.net/ShopSmart?retrywr
ites=true&w=majority&appName=project
Server is running at port : 8000
connectd to the database

```

## 2. Start the Frontend:

- Navigate to the client directory and run: **npm start**
- Run `npm start` in the React project directory (client) and open `http://localhost:3000` in your browser.

```
Compiled successfully!

You can now view clinet in the browser.

Local:            http://localhost:3000
On Your Network:  http://192.168.56.1:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

## 3. Database Operations:

- Ensure MongoDB is running locally or via MongoDB Atlas for remote access.

### 2. Development Environment:

Choose a code editor or Integrated Development Environment (IDE) that suits your preferences, such as Visual Studio Code.

- Visual Studio Code: Download from <https://code.visualstudio.com/download>

## 7. API Documentation

This section documents all the API endpoints exposed by the backend of ShopSmart. Each endpoint is detailed with the HTTP request method, parameters, and example request/response formats. These endpoints manage users, products, and orders for the application.

### 1. User API Endpoints

#### User Registration

- **Endpoint:** `/signup`
- **Method:** `POST`
- **Description:** Registers a new user.

- **Request Body and Response**

```

▼
{
  _id: ObjectId('671093b58cf8c9a91e8e50cf'),
  firstName: "Srihari",
  lastName: "Gudipati",
  email: "srihari24@gmail.com",
  password: "Srihari@524",
  confirmPassword: "Srihari@524",
  image: "data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAQABAAD/2wCEAAICAgICAgICAgI...
  __v: 0
}

```

### User Login

- **Endpoint:** /login
- **Method:** POST
- **Description:** Authenticates the user and returns user data.
- **Request Body and Response**

```

{ email: 'srihari24@gmail.com', password: 'Srihari@524' }
{
  _id: new ObjectId('671093b58cf8c9a91e8e50cf'),
  firstName: 'Srihari',
  lastName: 'Gudipati',
  email: 'srihari24@gmail.com',
  image: 'data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAQABAAD/2wCEAAICAgI
CAgICAgIDAwIDAwQDAwMDBAYEBAQEBAJBQYFBQYFCQgJBWCHCQgOCwkJCw4QDQwNEBMRERM
YFxfghyoBAGICAgICAgICAgMDAgMDAwMEBgQEBAQEBAkFBgUFBgUJCAkHBwcJCA4LCQk
LDhANDAA0QEXERExgXGB8fKv/CABEIAgACAAMBIgACEQEDEQH/xAA3AAACAwEBAQEBAQAAAA

```

### User Logout

- **Endpoint:** /logout
- **Method:** POST
- **Description:** Logs out the user and clears user data from the session (localStorage).
- **Response:**

```

{
  "message": "User logged out successfully."
}

```

## 2. Integration with Redux Toolkit

Using your provided Redux slice, here's how the API interactions correspond with the state management:

### 1. User Registration:

- On successful registration, the user data returned from the /register endpoint can be dispatched to the loginRedux action to store it in the Redux state and localStorage.

### 2. User Login:

- Upon successful login, the data returned from the /login endpoint should also dispatch the loginRedux action, setting the user data in the Redux state.

### 3. Get User Profile:

- The data retrieved from the /profile endpoint can be used to pre-fill user information in your application, or it can be dispatched to update the Redux state using updateUser.

### 4. Update User Profile:

- After updating the user profile through the /profile endpoint, dispatch the updateUser action to update the Redux state and localStorage.


### 5. User Logout:

- When the user logs out, you would call the /logout endpoint and then dispatch the logoutRedux action to clear the user state and remove data from localStorage.

## 3. Product API Endpoints


### Get All Products

- **Endpoint:** /product
- **Method:** GET
- **Description:** Retrieves a list of all products.
- **Query Parameters (optional):**
  1. **category:** Filter products by category.
  2. **search:** Search products by name or description.
- **Response:**

```
 {  "_id": ObjectId('66fe3af8c9f48e930f988d70'),  "name": "apple",  "category": "fruits",  "image": "data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAQABAAQ/2wCEAAoHCBYWFRgWFhU...",  "price": "56",  "description": "Fresh apples",  "__v": 0}
```

### Get Product by category

- **Endpoint:** /product/:category
- **Method:** GET
- **Description:** Fetches details of a specific product by its category.
- **URL Parameters:**
  1. **category:** Get all the product which is belongs to search category.

```
 {  "_id": ObjectId('66fe3af8c9f48e930f988d70'),  "name": "apple",  "category": "fruits",  "image": "data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAQABAAQ/2wCEAAoHCBYWFRgWFhU...",  "price": "56",  "description": "Fresh apples",  "__v": 0}
```

### Create New Product (Admin Only)

- **Endpoint:** /uploadProduct
- **Method:** POST
- **Description:** Adds a new product to the inventory.
- **Request Body:**

```
{  
  "name": "Mango",  
  "category": "Freshly picked mangoes.",  
  "image": "image.png.....",  
  "price": 200,  
  "description": "summer fruit",  
}
```
- **Response:**

### Update Product (Admin Only)

- **Endpoint:** /products/:id
- **Method:** PUT
- **Description:** Updates an existing product by its ID.
- **URL Parameters:**
  - id: The unique identifier of the product.
- **Request Body (Optional):**

```
{  
  "price": 499,  
}
```
- **Response:**

```
{  
  "_id": "60c72ba04f1a4c3d88047efb",  
  "name": "Mango",  
  "description": "Freshly picked mangoes.",  
  "price": 499,  
  "category": "fruits",  
  "image": "mango.jpg"  
}
```

## 3. Order API Endpoints



## Create New Order

- **Endpoint:** /orders
- **Method:** POST
- **Description:** Places a new order with selected products and user information.
- **Request Body:**

```
{
  "userId": "60c72b2f4f1a4c3d88047ef8",
  "products": [
    {
      "productId": "60c72b9f4f1a4c3d88047ef9",
      "quantity": 2
    },
    {
      "productId": "60c72ba04f1a4c3d88047efa",
      "quantity": 5
    }
  ],
  "totalAmount": 12.95,
  "address": {
    "street": "123 Main St",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90001"
  }
}
```

- **Response:**

```
{
  "_id": "60c72bcf4f1a4c3d88047efc",
  "userId": "60c72b2f4f1a4c3d88047ef8",
  "products": [
    {
      "productId": "60c72b9f4f1a4c3d88047ef9",
```

```

        "quantity": 2
      },
      {
        "productId": "60c72ba04f1a4c3d88047efa",
        "quantity": 5
      }
    ],
    "totalAmount": 12.95,
    "status": "pending",
    "address": {
      "street": "123 Main St",
      "city": "Los Angeles",
      "state": "CA",
      "zip": "90001"
    }
  }
}

```

#### Get User Orders

- **Endpoint:** /orders/:userId
- **Method:** GET
- **Description:** Retrieves all orders made by a specific user.
- **URL Parameters:**
  - **userId:** The unique identifier of the user.
- **Response:**

```

[
  {
    "_id": "60c72bcf4f1a4c3d88047efc",
    "products": [
      {
        "productId": "60c72b9f4f1a4c3d88047ef9",
        "quantity": 2
      },
      {

```

```

        "productId": "60c72ba04f1a4c3d88047efa",
        "quantity": 5
      }
    ],
    "totalAmount": 12.95,
    "status": "pending",
    "address": {
      "street": "123 Main St",
      "city": "Los Angeles",
      "state": "CA",
      "zip": "90001"
    }
  }
]

```

### Update Order Status (Admin Only)

- **Endpoint:** /orders/:id
- **Method:** PATCH
- **Description:** Updates the status of a specific order.
- **URL Parameters:**
  - id: The unique identifier of the order.
- **Request Body:**

```

{
  "status": "shipped"
}

```

- **Response:**

```

{
  "_id": "60c72bcf4f1a4c3d88047efc",
  "status": "shipped"
}

```

**This are the all API in the Project:**

```
32 //model
33 const userModel = mongoose.model("user", userSchema);
34 //api
35 app.get("/", (req, res) => {
36   res.send("Server is running");
37 });
38
39 //signup api
40 app.post("/signup", async (req, res) => {
41   console.log(req.body);
42   const { email } = req.body;
43
44   try {
45     const result = await userModel.findOne({ email: req.body.email });
46
47     if (result) {
48       res.send({ message: "Email id is already registered", alert: false });
49     } else {
50       const data = new userModel(req.body);
51       await data.save(); // Save the new user data
52       res.send({ message: "Registration is successful", alert: true });
53     }
54   } catch (err) {
55     console.error(err); // Log the error for debugging
56     res.status(500).send({ message: "An error occurred" });
57   }
58 });
```

Fig: Signup Api

```
59 //login api
60 app.post("/login", async (req, res) => {
61   console.log(req.body);
62   const { email } = req.body;
63
64   try {
65     const result = await userModel.findOne({ email: req.body.email });
66
67     if (result) {
68       const dataSend = {
69         _id: result._id,
70         firstName: result.firstName,
71         lastName: result.lastName,
72         email: result.email,
73         image: result.image,
74       };
75       console.log(dataSend);
76       res.send({
77         message: "Login is Successfully",
78         alert: true,
79         data: dataSend,
80       });
81     } else {
82       res.send({ message: "This email is not available", alert: false });
83     }
84   } catch (err) {
85     console.error(err); // Log the error for debugging
86     res.status(500).send({ message: "An error occurred" });
87   }
88 });
```

Fig: Login Api

```

114 //save product in data
115 //api
116 app.post("/uploadProduct", async (req, res) => {
117   // console.log(req.body)
118   const data = await productModel(req.body);
119   const datasave = await data.save();
120   res.send({ message: "Upload successfully" });
121 });
122 //
123 app.get("/product", async (req, res) => {
124   try {
125     const data = await productModel.find({});
126     res.json(data); // More efficient
127   } catch (error) {
128     res.status(500).json({ message: "Error retrieving products" });
129   }
130 });

```

**Fig: UploadProduct and Product Api**

```

156 app.post("/contact", async (req, res) => {
157   try {
158     const { name, email, phone, message } = req.body;
159     const result = await userModel.findOne({ email: req.body.email });
160
161     if (result) {
162       const data = await Contact(req.body);
163       const datasave = await data.save();
164       res.status(201).json({ message: "Submitted successfully" });
165     }
166   } catch (error) {
167     console.error("Error submitting contact form:", error);
168     res.status(500).json({ message: "Server error. Please try again." });
169   }
170 });
171 app.get("/allcontacts", async (req, res) => {
172   try {
173     const complaints = await Contact.find(); // Fetch all contacts
174     res.send({
175       message: "Data fetched successfully",
176       alert: true,
177       data: complaints,
178     });
179   } catch (err) {
180     console.error(err);
181     res.status(500).send({ message: "An error occurred while fetching Data" });
182   }
183 });

```

**Fig: Contact and AllContacts Api**

```

187 const stripe = new Stripe(process.env.STRIPE_SECRET_KEY);
188
189 app.post("/create-checkout-session", async (req, res) => {
190   try {
191     const params = {
192       submit_type: "pay",
193       mode: "payment",
194       payment_method_types: ["card"],
195       billing_address_collection: "auto",
196       shipping_options: [{ shipping_rate: "shr_1Q6xI7RxZdHdWlQKxHBETndM" }],
197
198       line_items: req.body.map((item) => {
199         return {
200           price_data: {
201             currency: "inr",
202             product_data: {
203               name: item.name,
204               // images : [item.image]
205             },
206           unit_amount: item.price * 100,
207         },
208         adjustable_quantity: {
209           enabled: true,
210           minimum: 1,
211         },
212         quantity: item.qty,
213       });
214     },
215     success_url: `${process.env.FRONTEND_URL}/success`,
216     cancel_url: `${process.env.FRONTEND_URL}/cancel`.

```

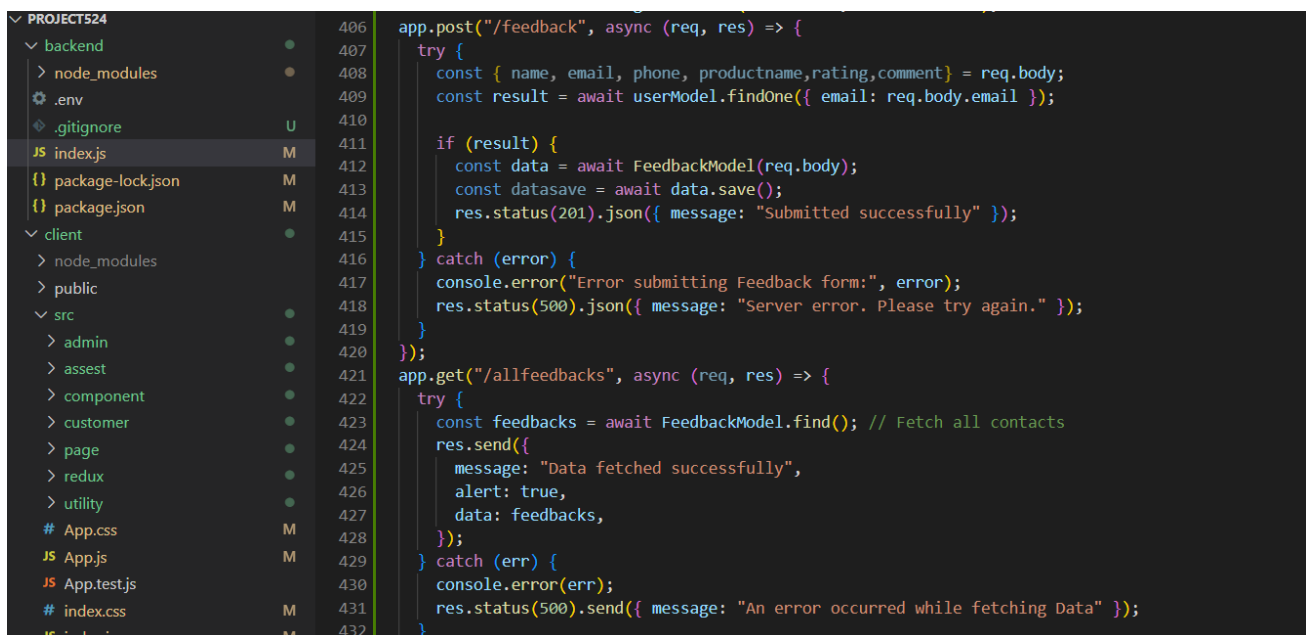
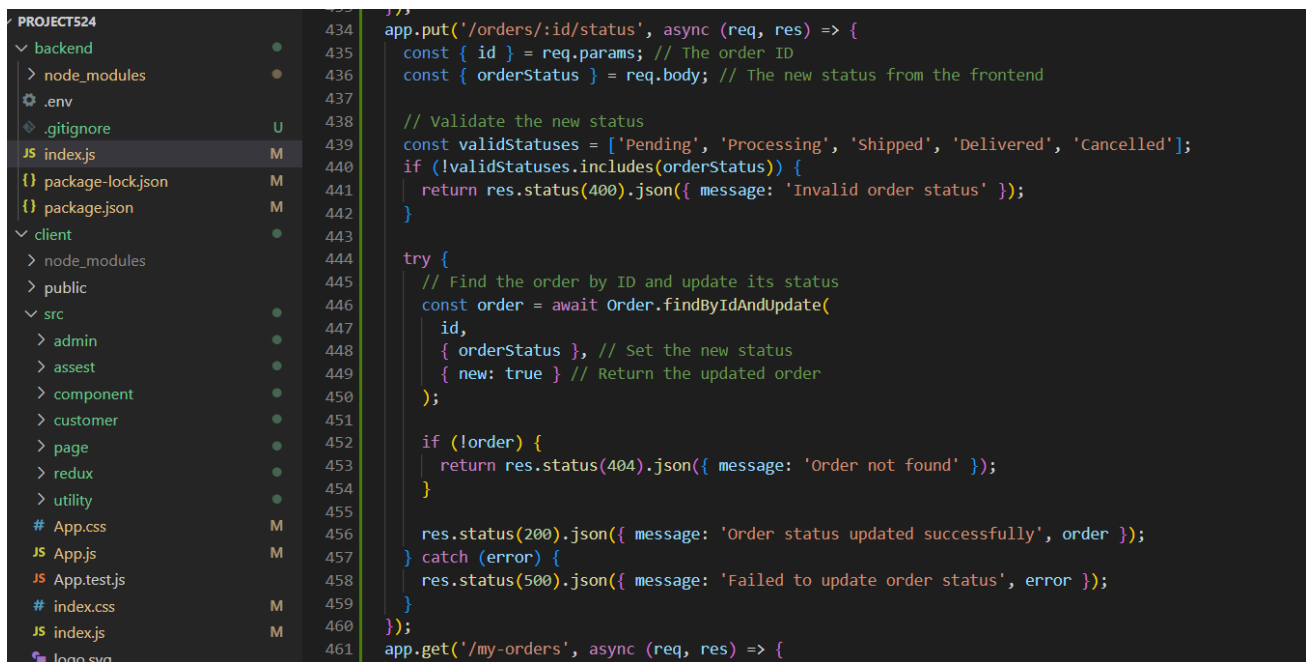
**Fig: Strip create-checkout-session Api**

```

341 const Order = mongoose.model('Order', orderSchema);
342 app.post('/order', async (req, res) => {
343   const { userId, cartItems, totalAmount } = req.body;
344
345   if (!userId || !cartItems || !totalAmount) {
346     return res.status(400).json({ message: 'User ID, cart items, and total amount are required.' });
347   }
348
349   try {
350     const newOrder = new Order({
351       userId,
352       cartItems,
353       totalAmount,
354       orderStatus: 'Pending' // Set the initial status if not set in the body
355     });
356
357     await newOrder.save();
358
359     res.status(201).json({ message: 'Order created successfully', orderId: newOrder._id });
360   } catch (error) {
361     res.status(500).json({ message: 'Error creating order', error });
362   }
363 }
364
365 app.get('/orders', async (req, res) => {
366   try {
367     const orders = await Order.find()
368       .populate('userId', 'firstName lastName email') // Populate user details

```

**Fig: Orders (Post the ordered details) and (retrieve all orders by Id)**



## 8. Authentication

In ShopSmart, authentication and authorization are managed using Redux Toolkit and localStorage. Unlike traditional systems that rely on JWT tokens or server-side sessions, the application leverages local state and client-side storage for managing user sessions. Below is an explanation of how this system works.

### 1. Overview of Authentication Process

The authentication process ensures that only registered users can log into the system, and once logged in, the user's session is maintained locally. Authorization defines which parts of the application different users can access (e.g., regular users versus admins).

Here's how **authentication** and **authorization** are handled:

- **User Registration:** When a user registers, their details (e.g., first name, last name, email, password) are stored in the database. Upon successful registration, user data is returned to the frontend and saved in the Redux store.
- **User Login:** After the user logs in, the server responds with user details, which are stored in the **Redux state** and **localStorage** for session persistence.
- **Session Management:** LocalStorage is used to store user details, allowing the session to persist across page reloads or browser closures.
- **User Logout:** Upon logout, the Redux store is cleared and the user's information is removed from **localStorage**, effectively ending the session.

### 2. Authentication Flow

#### Login Flow:

##### 1. Frontend:

- The user submits their email and password through a login form.
- The login request is sent to the backend API (/api/users/login).
- If login is successful, the server responds with the user's profile data.
- The profile data is then saved to both the Redux state and localStorage using the loginRedux action.

##### 2. Redux Toolkit:

- The loginRedux action is dispatched with the user data.
- The Redux slice saves the user's profile information (e.g., email, firstName, lastName, image) in the global state and stores the same data in localStorage for session persistence.

##### 3. Logout Flow:

- The user clicks the "Logout" button, which triggers the logoutRedux action.
- The user's session is cleared from **Redux state** and **localStorage**, ensuring that the user is logged out.

### 3. Session Management with localStorage:

Since localStorage is used to persist the user session, user data will remain even after refreshing the page or closing and reopening the browser (until explicitly logged out). Here's how session persistence is implemented:



- **On Login:** User data is saved to both the Redux store and localStorage:
- **On Page Load:** When the app initializes, it checks if any user data is stored in localStorage. If data is found, it is loaded into the Redux state to maintain the session.

```
// Retrieve initial state from localStorage if available, otherwise
const initialState = localStorage.getItem("user")
  ? JSON.parse(localStorage.getItem("user"))
  : {
    email: "", // Make sure it's set to a string
    firstName: "", // Ensure it's initialized
    image: "",
    lastName: "",
    _id: "",
  };

export const userSlice = createSlice({
  name: "user",
  initialState,
  reducers: {
    loginRedux: (state, action) => {
      const userData = action.payload.data;
      state._id = userData._id;
      state.firstName = userData.firstName;
      state.lastName = userData.lastName;
      state.email = userData.email;
      state.image = userData.image;
      localStorage.setItem("user", JSON.stringify(state));
    },
  },
});
```

**Fig: loginRedux Code**

```
logoutRedux: (state) => {
  state._id = "";
  state.firstName = "";
  state.lastName = "";
  state.email = "";
  state.image = "";
  localStorage.removeItem("user");
},
});
```

- **On Logout:** User data is removed from localStorage and the Redux state is reset:

**Fig: logoutRedux Code**

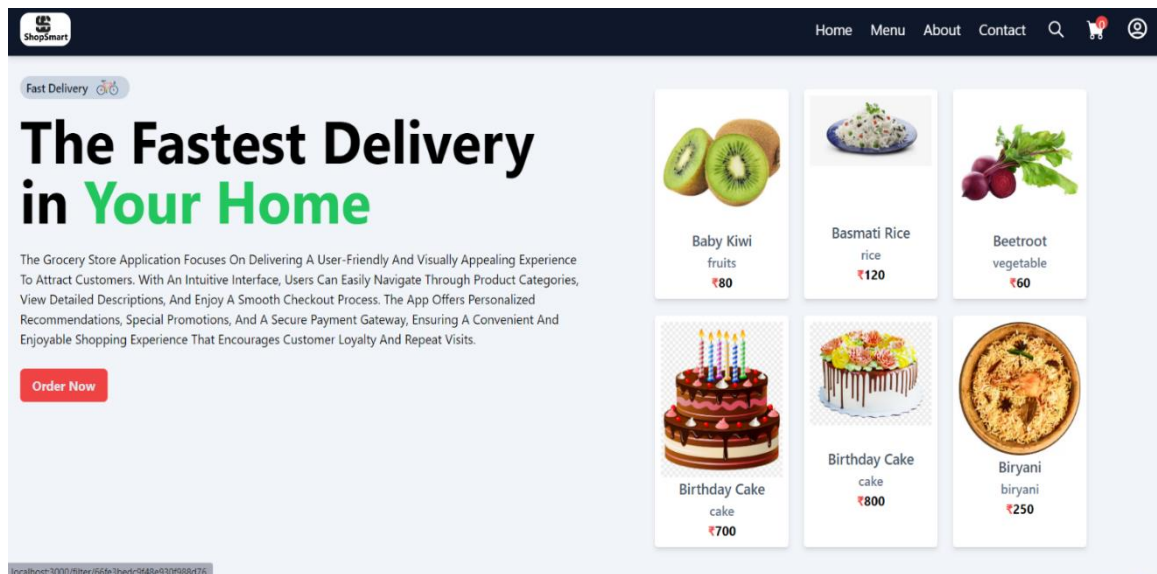
## 9. User Interface

### 1. Home Page

The home page serves as the landing page of the application, providing a welcoming interface and displaying featured products, categories, and any promotional banners.

#### Key UI Elements to Capture:

- Navigation bar (logo, search bar, categories, user login, menu, about, contact, addcart).



- Featured products section (highlighting special deals or new arrivals).

**Fig: Home page**

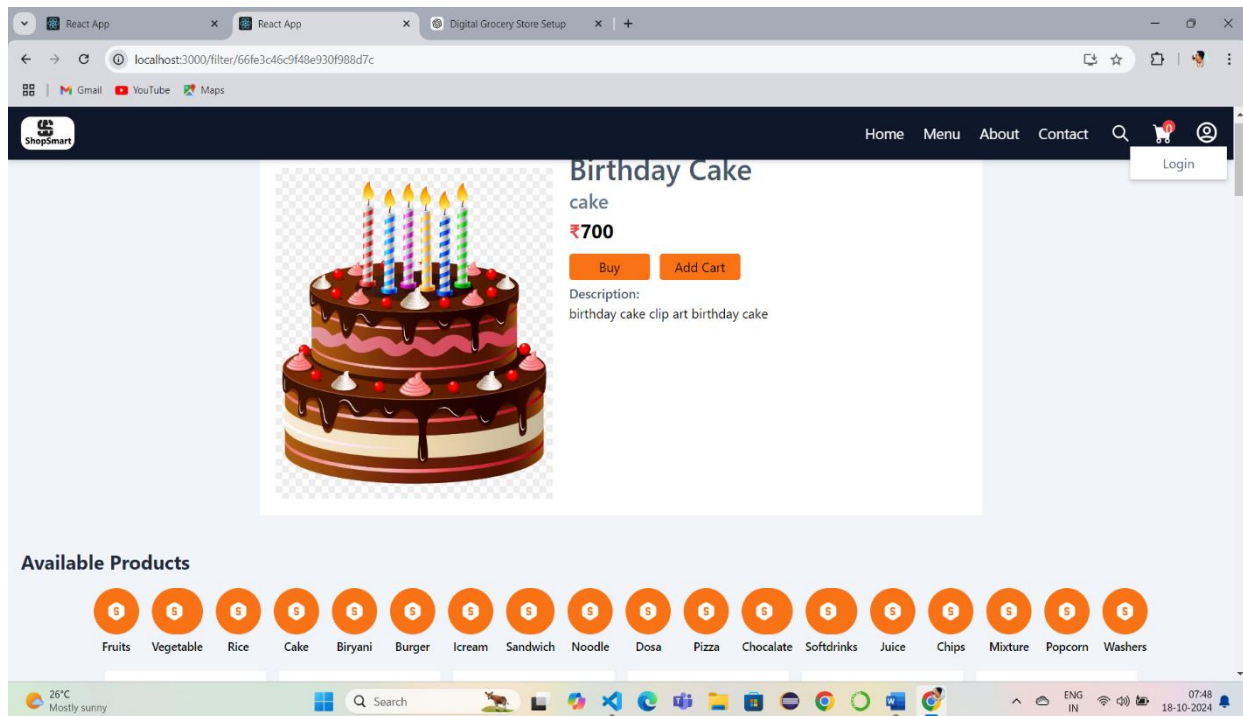
### 2. Product Listings Page

#### Description:

This page displays a list of products, allowing users to browse, filter, and search for items. It includes product cards showing images, names, prices, and stock availability.

#### Key UI Elements to Capture:

- Product search bar.
- Category filters (e.g., Fruits, Vegetables, Beverages).
- Product cards with product images, prices, and "Add to Cart" buttons.



**Fig: Product Listings Page**

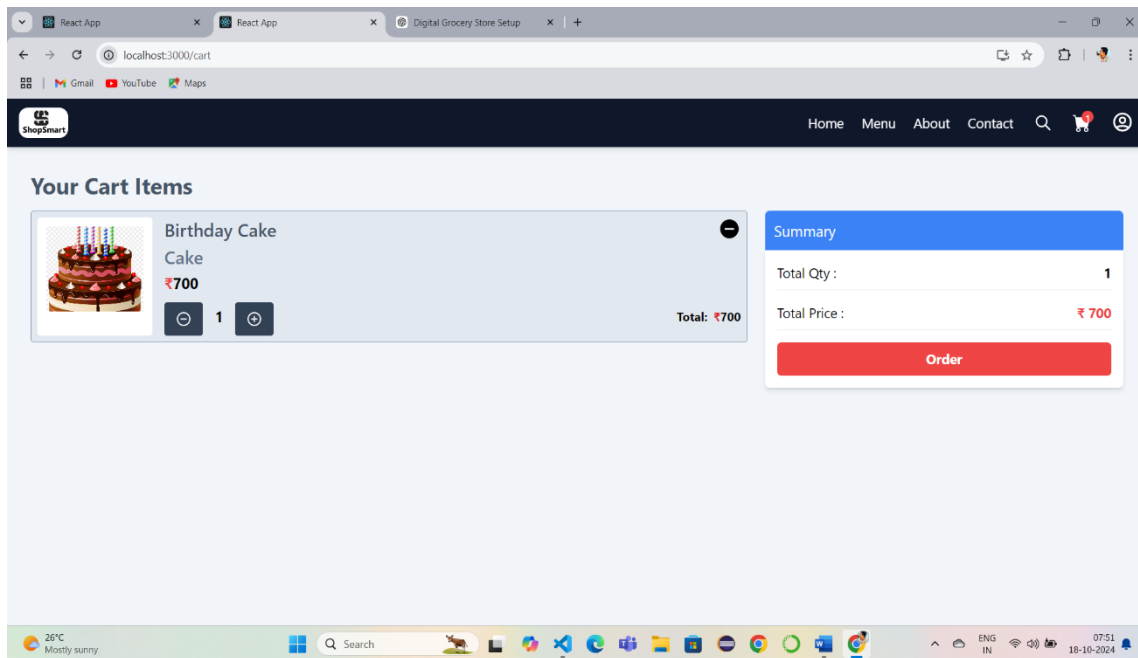
### 3. Shopping Cart Page

#### Description:

The shopping cart page shows a summary of the user's selected items. It allows users to update quantities, remove items, and proceed to checkout.

#### Key UI Elements to Capture:

- List of products in the cart with images, quantities, and prices.
- "Update Quantity" and "Remove" buttons.
- Cart subtotal, tax, and total amount.



**Fig: Shopping Cart Page**

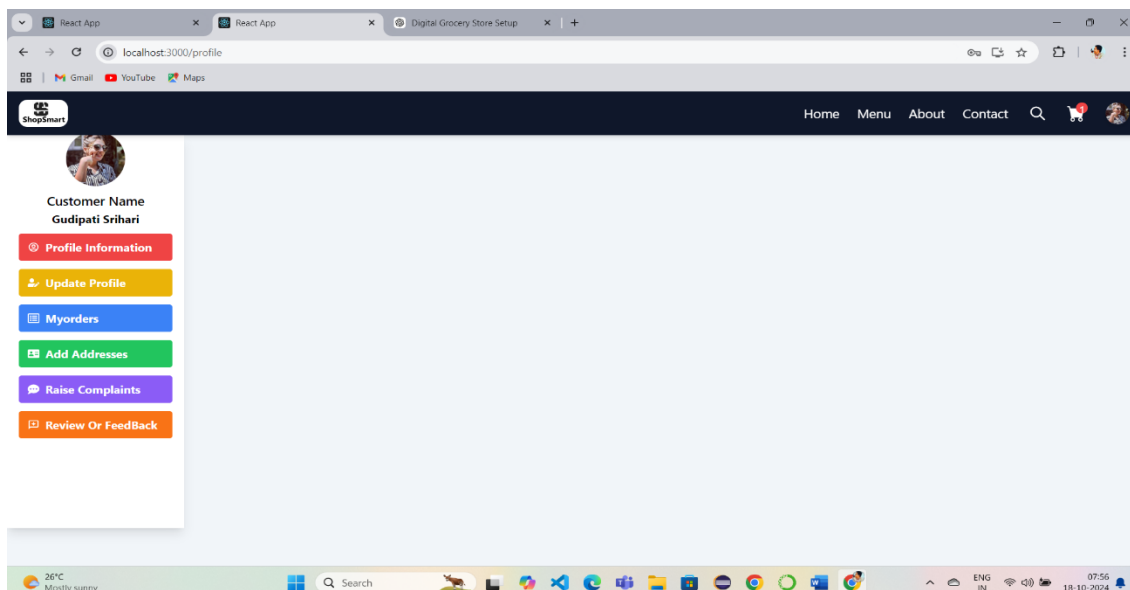
#### 4. User Profile Page

##### Description:

The user profile page allows users to view and update their personal information, such as their name, email, and profile picture. It also includes a section for viewing past orders.

##### Key UI Elements to Capture:

- Editable fields for first name, last name, email, and profile picture.
- List of past orders with order details (product, date, status).



**Fig: User Panel**

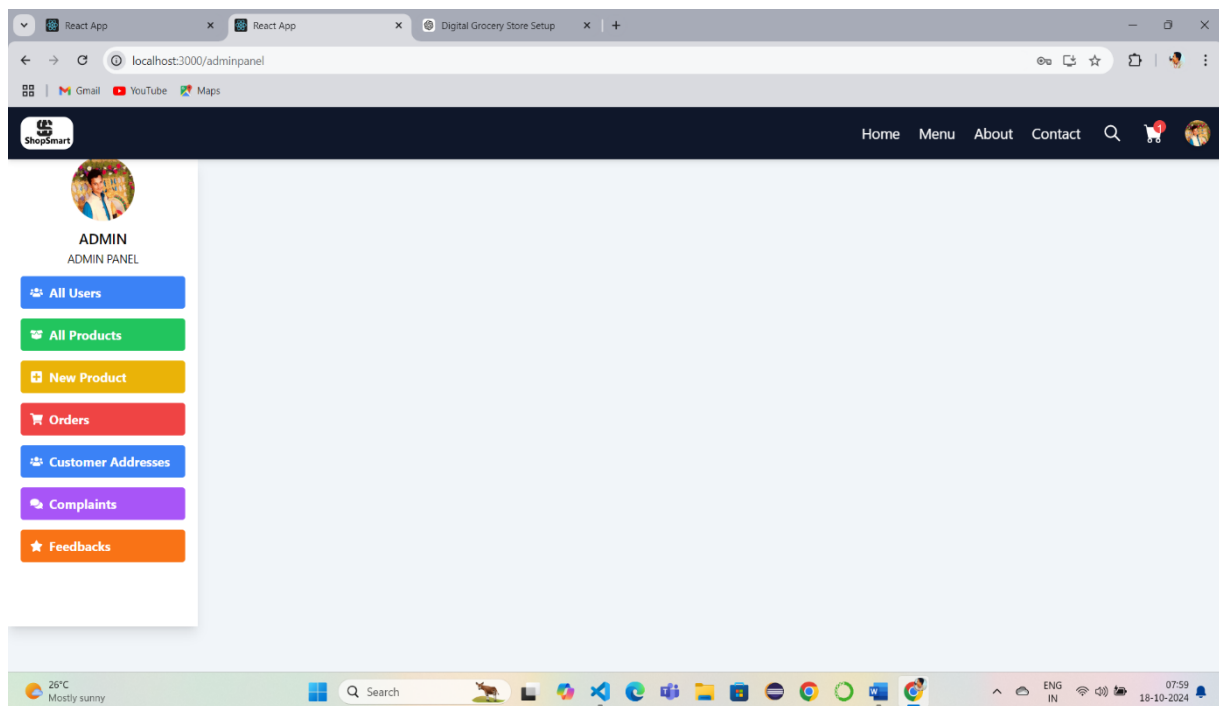
## 5. Admin Dashboard (If Applicable)

### Description:

For admin users, the dashboard provides functionality to manage products, view orders, and monitor user activities.

### Key UI Elements to Capture:

- Product management interface (add, update, delete products).
- Order management interface (view and update order statuses).



**Fig: Admin Dashboard**

## 10. Testing

Testing is an essential part of ensuring the reliability, functionality, and user experience of the ShopSmart grocery web application. The testing strategy used for ShopSmart aims to cover various aspects, including unit tests, integration tests, and manual testing of the user interface.

Below is a detailed description of the testing strategy and tools used in the project:

### 1. Testing Strategy:

The testing strategy for ShopSmart includes the following types of tests:

**I. Unit Testing:** Unit tests focus on testing individual components or functions in isolation. For example, a unit test would verify that a specific function returns the expected result given a certain input or that a React component renders correctly based on its props.

**II. Integration Testing:** Integration tests ensure that different parts of the application (e.g., frontend components and backend APIs) work together as expected. This includes testing how the UI interacts with the API to fetch data, submit forms, and handle errors.

**III. End-to-End Testing (E2E):** End-to-End tests validate the entire flow of the application from the user's perspective. E2E testing checks that critical user journeys (e.g., searching for products, adding to cart, completing checkout) work as expected.

**IV. Manual Testing:** Manual testing is performed by developers or QA engineers to check that the application works as expected across different devices and browsers. This includes testing the user interface, usability, and responsive design.

By utilizing a combination of **unit tests, integration tests, end-to-end tests, and manual testing**, ShopSmart ensures a high level of code quality, reliability, and user satisfaction. This testing strategy helps in catching bugs early in development, verifying that the application works as expected across different environments, and delivering a seamless shopping experience for users.

## 11. Screenshot or Demo

### Live Demo Link:

- [Shopsmart Demo Video](#) This the Demo video of the application.

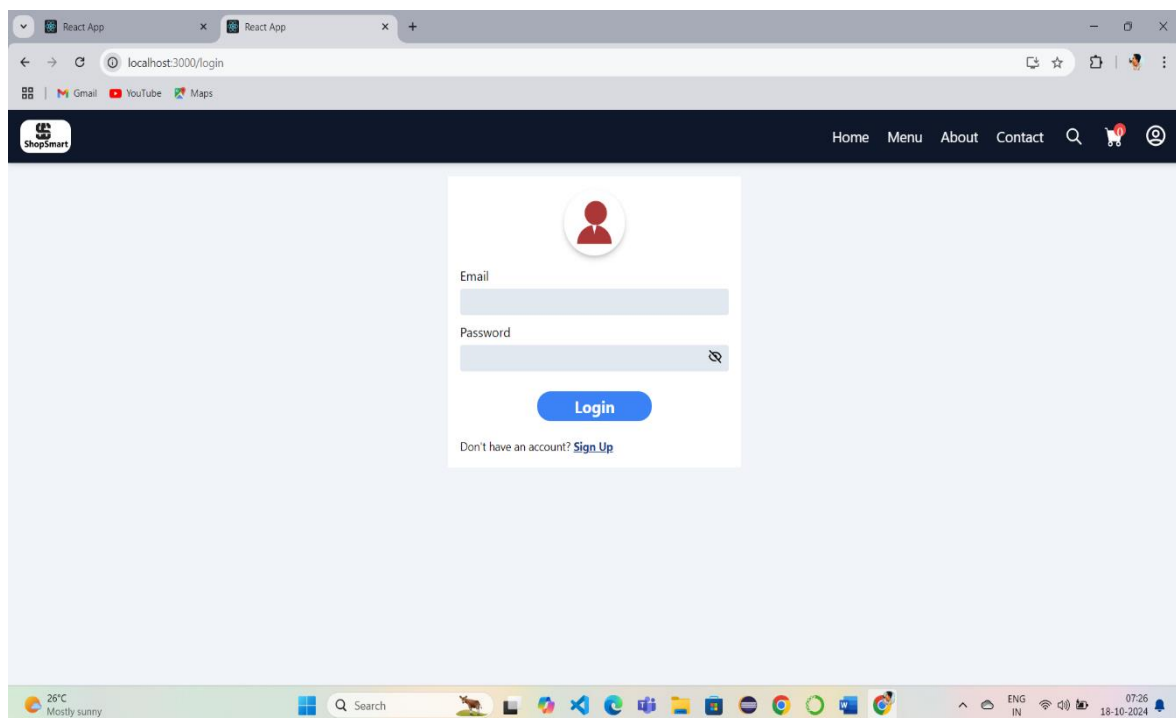
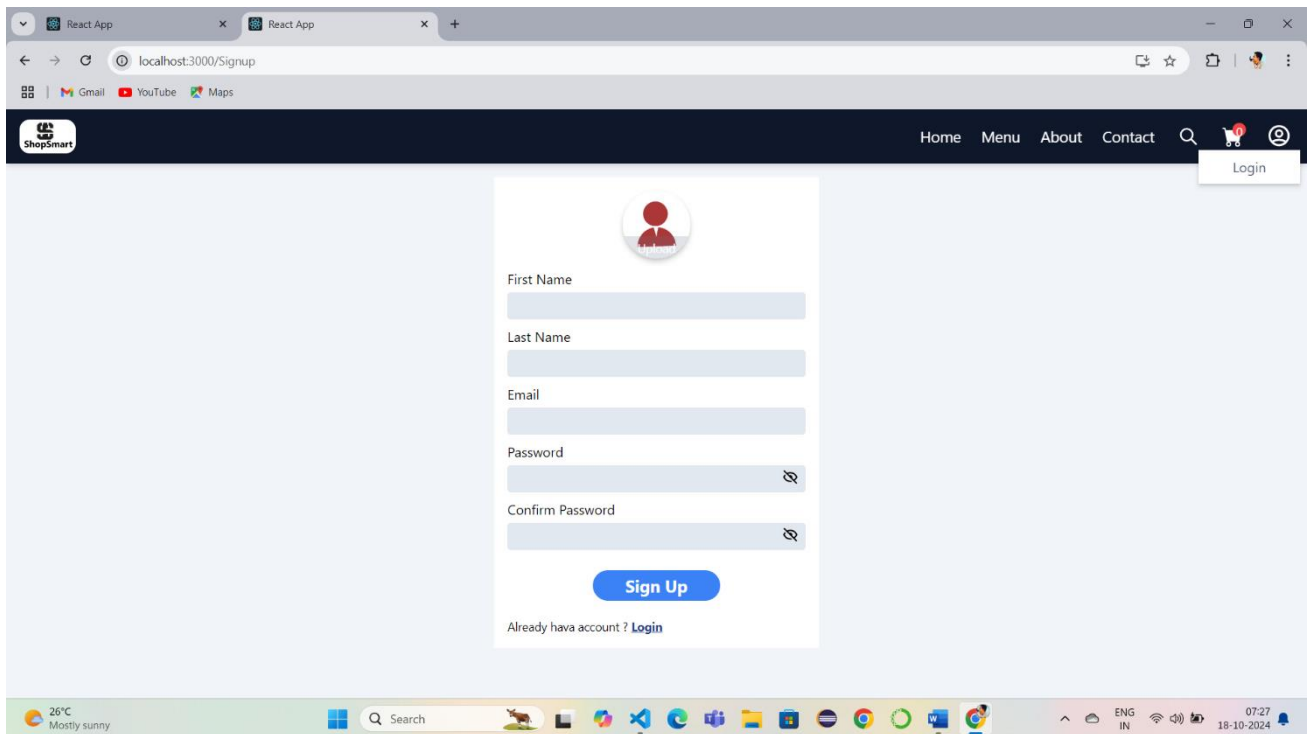


Fig: Login Page



**Fig: SignUp Page**

## 12. Known Issues

### 1. Slow Initial Load for Large Product Catalog

- **Issue:** The application experiences slow initial loading times when a large number of products are fetched from the backend.
- **Cause:** All product data is being fetched in a single API call without pagination, which causes the initial load to be slow, especially with a large product catalog.
- **Workaround:** Implement pagination or lazy loading to improve performance on large datasets. This is planned for a future update.

### 2. Product Management Not Responsive on Mobile:

- **Issue:** The admin product management dashboard is not fully responsive when accessed via mobile devices. Some buttons and fields are misaligned or overlap.
- **Cause:** The dashboard layout is designed primarily for desktop, and the mobile layout has not been fully optimized.
- **Workaround:** Advise admins to manage products on desktop or larger tablet screens. A mobile-friendly version of the admin dashboard is planned for a future release.

### 3. Session Timeout Without Notification

- **Issue:** The user session may expire after a period of inactivity without any warning, causing the user to be logged out unexpectedly.

- **Cause:** There is no session expiration notification or automatic refresh token system implemented, leading to session timeouts without user awareness.
- **Workaround:** Users need to log in again when their session expires. Implementing a session warning or token refresh system is planned.

#### 4. Inconsistent Data Sync Between LocalStorage and Redux

- **Issue:** Occasionally, user data stored in localStorage is not synced correctly with the Redux state, leading to discrepancies in user session management.
- **Cause:** The localStorage and Redux state initialization logic may not be handling asynchronous updates properly.
- **Workaround:** Ensure that the page is fully refreshed to reflect the correct state. Fixing the sync logic between localStorage and Redux is a high priority for future updates.

#### 5. Order History Not Fully Displaying on Profile Page

- **Issue:** For users with a large number of orders, the order history on the profile page does not load all past orders, displaying only a subset.
- **Cause:** This issue may be due to missing pagination or limits on the number of orders returned from the database.
- **Workaround:** Limit the number of orders displayed on the profile page to a fixed amount, with the option to load more. Proper pagination for the order history is planned for future updates.

## 13. Future Enhancements

The ShopSmart application is designed to offer a seamless digital grocery shopping experience. However, there are several potential future features and improvements that could enhance the user experience, improve performance, and introduce advanced functionality. Below is an outline of some ideas for future enhancements to the project.

### 1. AI-Powered Product Recommendations

#### Description:

Integrate AI and machine learning algorithms to offer personalized product recommendations based on a user's browsing and purchasing history.

- **Benefits:** Improves user engagement by suggesting products they may be interested in based on their preferences.
- **Implementation:**
  - Analyze user purchase patterns and suggest products using collaborative filtering or content-based filtering.
  - Integrate third-party recommendation engines or build a custom solution using tools like TensorFlow or PyTorch.



## 2. Voice Search Integration

### Description:

Enable voice search functionality so users can search for products using voice commands.

- **Benefits:** Enhances accessibility and offers a modern search experience, making the app more user-friendly for all audiences.
- **Implementation:**
  - Integrate with voice recognition APIs such as Google's Web Speech API or third-party services like AWS Alexa or Google Assistant.
  - Add a microphone button to the search bar to capture and process voice input.

## 3. Social Media Integration for Reviews and Sharing

### Description:

Allow users to share products or their shopping experience on social media platforms like Facebook, Instagram, and Twitter.

- **Benefits:** Boosts the app's visibility and encourages user engagement through social sharing.
- **Implementation:**
  - Add social media sharing buttons to product pages.
  - Integrate with Facebook's Open Graph API or Twitter's Card API for streamlined sharing.
  - Enable users to leave reviews via their social media accounts.

## 4. Mobile Application (iOS/Android)

### Description:

Develop native mobile applications for **iOS** and **Android** to provide a more optimized experience for mobile users, enhancing performance and responsiveness compared to a mobile web app.

- **Benefits:** Offers a more seamless experience for users on mobile devices, including access to push notifications and offline features.
- **Implementation:**
  - Use frameworks like **React Native** or **Flutter** to build cross-platform mobile apps.
  - Leverage push notifications for order updates, promotions, and reminders.

## 5. Advanced Order Tracking with Notifications

### Description:

Allow users to track the real-time status of their orders with detailed updates on shipping, delivery times, and location tracking.

- **Benefits:** Enhances the customer experience by providing visibility into the entire order process, reducing uncertainty and improving satisfaction.
- **Implementation:**
  - Integrate with third-party shipping APIs (e.g., FedEx, UPS, DHL) to get real-time shipping status.
  - Use WebSockets or long polling to provide real-time updates and notifications to the user.
  - Display a map with the live location of the delivery (similar to Uber or Amazon).

## 6. Loyalty and Rewards Program

### Description:

Introduce a loyalty program where users can earn points on purchases, redeemable for discounts, free products, or exclusive offers.

- **Benefits:** Encourages repeat purchases, improves customer retention, and builds brand loyalty.
- **Implementation:**
  - Track user purchases and assign reward points for each transaction.
  - Build a loyalty dashboard in the user profile showing point balance and available rewards.
  - Create an interface for applying points to purchases at checkout.

## 7. Subscription-Based Grocery Delivery Service

### Description:

Offer subscription-based services where users can set up recurring orders for essential grocery items, delivered on a schedule (e.g., weekly, monthly).

- **Benefits:** Provides a convenient solution for users who need regular deliveries of groceries, such as families or busy professionals.
- **Implementation:**
  - Allow users to create a "subscription" list of frequently purchased items.
  - Set up recurring payment functionality through integration with payment gateways (Stripe, PayPal).
  - Include subscription management features for users to adjust quantities or skip deliveries.

## 8. Improved Admin Dashboard with Analytics

**Description:**

Expand the admin dashboard to include detailed analytics on sales performance, user behavior, product popularity, and inventory management.

- **Benefits:** Provides store admins with insights to make data-driven decisions, optimize product listings, and manage inventory efficiently.
- **Implementation:**
  - Integrate with analytics tools such as Google Analytics or custom dashboard solutions like **Chart.js** or **D3.js**.
  - Display metrics such as total sales, average order value, most purchased products, and customer demographics.
  - Include automated inventory tracking and low-stock alerts.

## 9. Multi-Language and Multi-Currency Support

**Description:**

Expand the app's reach by offering multi-language and multi-currency support, allowing users from different regions to shop in their preferred language and currency.

- **Benefits:** Increases the app's appeal to a global audience, providing localization options that enhance user experience.
- **Implementation:**
  - Use internationalization libraries like **react-i18next** to support multiple languages.
  - Integrate currency conversion APIs to display product prices in various currencies.
  - Allow users to select their language and currency preferences from their profile settings.

## 10. Dark Mode and Theme Customization

**Description:**

Introduce a dark mode and customizable themes to improve user experience and accessibility, especially for users who prefer low-light conditions or want more personalization.

- **Benefits:** Enhances user satisfaction by providing personalization options, making the app easier to use in different lighting conditions.
- **Implementation:**
  - Add a toggle for dark mode in the user settings using CSS variables.
  - Allow users to select custom themes (e.g., font size, color schemes).
  - Store user preferences in localStorage or their profile settings.

## 11. Inventory Forecasting and Automation for Admins

**Description:**

Implement inventory forecasting tools to help admins predict demand for products and automate stock reordering based on sales trends.

- **Benefits:** Reduces the risk of stockouts and ensures a steady supply of popular products, optimizing inventory management.
- **Implementation:**
  - Use machine learning algorithms to predict sales trends based on historical data.
  - Automate reordering notifications or direct integrations with suppliers for automatic stock replenishment.
  - Provide an admin interface for viewing forecasted product demand and low-stock alerts.

## 12. Advanced Search with Filters and Sorting

**Description:**

Enhance the search functionality by adding more advanced filters (e.g., price range, ratings, brand) and sorting options (e.g., price, popularity, latest).

- **Benefits:** Improves the user's ability to find exactly what they are looking for quickly and efficiently.
- **Implementation:**
  - Add filter components for product attributes such as price, rating, and category.
  - Provide sorting options for users to order search results by relevance, price (ascending/descending), and popularity.
  - Use Elasticsearch or Algolia for faster and more flexible search capabilities.