**Full Stack Development with MERN**

**Project Documentation format**

**1. Introduction**

* **Project Title:** SB Foods-Food Ordering App
* **Team Members:** K Bhavana
  + - * Bodagala Gayathri
      * K Bhuvaneswari
      * P Lahari

**2. Project Overview**

**Purpose:**

**The primary goal of this project is to create a responsive and user-friendly food ordering website where users can browse various food items, place orders, and manage their accounts. Additionally, the website includes an admin panel for managing food items and orders, ensuring smooth operations and efficient order handling.**

**Features:**

1. Homepage:

* Food Item List: Display a comprehensive list of available food items.
* Category Filtering: Users can filter items by categories such as desserts, noodles, etc.
* Responsive Design: The website adapts to different screen sizes for optimal viewing on various devices.

2. User Authentication:

* Sign Up: New users can create an account by providing their name, email, and a strong password.
* Login: Existing users can log in with their email and password.

3. Product Management:

* Add to Cart: Users can add food items to their cart.
* Quantity Management: Increase or decrease the quantity of items in the cart.
* Cart Page: View the list of selected items, total price, and delivery charges.

4. Checkout Process:

* Delivery Information: Users can enter their name, address, and phone number.
* Payment Integration: Process payments using Stripe with order details and a dummy card number.

5. Order Management:

* Order Confirmation: After payment, users are redirected to the order details page showing food items, quantity, total price, and order status.
* Order Status: Status updates such as "processing," "out for delivery," and "delivered."

6. Admin Panel:

* Order Management: View and update order statuses.
* Food Item Management: List, add, and delete food items.
* Add New Products: Upload product images, names, descriptions, categories, and prices.

**3. Architecture**

* **Frontend Architecture using React**

The frontend of the food ordering website is built using React, structured into several key components, state management, and routing. Here's a detailed description of the frontend architecture based on the given code:

Component Structure:

* The frontend is divided into multiple components, each responsible for a specific part of the user interface. These components are:

Cart Component (cart.jsx):

* Displays the items in the cart, along with their quantities, prices, and a remove button.
* Calculates the total amount and delivery charges.
* Provides a button to proceed to the checkout page.
* Allows entering and submitting promo codes.

Home Component (home.jsx):

* Acts as the landing page for the website.
* Includes the Header, ExploreMenu, FoodDisplay, and AppDownload components.
* Uses state to manage the selected category for displaying food items.

MyOrders Component (myOrders.jsx):

* Displays the user's past orders with details like items, quantities, total amount, and status.
* Fetches orders from the backend using an API call.

PlaceOrder Component (placeOrder.jsx):

* Collects delivery information and payment method from the user.
* Calculates the cart totals and handles the order placement process.
* Redirects users based on their authentication status and cart contents.

Verify Component (verify.jsx):

* Handles payment verification and redirects the user based on the verification result.
* Uses query parameters to get payment status and order ID.

Navbar Component (navbar.jsx):

* Provides navigation links to different sections of the website.
* Includes a search icon, cart icon with item count indicator, and user profile dropdown.
* Manages user authentication status and displays sign-in or profile options accordingly.
* State Management
* The application uses React's useState and useContext hooks for state management:

Global State:

* The StoreContext provides global state management for cart items, food list, user authentication, and other shared data across components.
* useContext(StoreContext) is used to access the global state and methods like removeFromCart, getTotalCartAmount, setCartItems, etc.

Local State:

* Components use useState for managing local state, such as form inputs in the PlaceOrder component and category selection in the Home component.

Routing

* The application uses react-router-dom for routing between different pages:

Navigation:

* The useNavigate hook is used to programmatically navigate between pages, such as redirecting users after order placement or payment verification.

Links and Routes:

* The Link component is used for client-side navigation, allowing users to navigate to different routes without a full page reload.
* **Backend:** Outline the backend architecture using Node.js and Express.js.

The backend of the food ordering website is built using Node.js and Express.js, providing a robust, scalable, and efficient environment for handling server-side logic and API endpoints. This architecture ensures smooth interactions between the frontend, the database, and the payment gateway.

**Server Setup**:

* + The server is initialized using Express.js, which handles HTTP requests and responses.
  + Middleware is used for parsing JSON, handling CORS, and logging requests.
* **API Endpoints**:
  + **Authentication**:
    - POST /signup: Creates a new user account.
    - POST /login: Authenticates an existing user.
  + **Food Items**:
    - GET /food-items: Retrieves all food items.
    - POST /food-items: Adds a new food item (admin only).
    - DELETE /food-items/:id: Deletes a food item (admin only).
  + **Cart**:
    - POST /cart: Adds an item to the cart.
    - GET /cart: Retrieves the current cart items.
  + **Orders**:
    - POST /orders: Creates a new order.
    - GET /orders/:userId: Retrieves orders for a specific user.
    - PUT /orders/:orderId: Updates order status (admin only).
* **Payment Integration**:
  + Stripe is integrated to handle payments securely.
  + Endpoint for processing payments and confirming transactions.
* **Error Handling**:
  + Custom middleware to handle errors and send appropriate responses.
* **Security**:
  + JWT (JSON Web Token) is used for authentication and securing endpoints.
  + Passwords are hashed using bcrypt before being stored in the database.
* **Database:** Detail the database schema and interactions with MongoDB.

**Database Interactions**

1. **User Operations**:
   * Creating and saving new users during signup.
   * Finding and authenticating users during login.
2. **Food Item Operations**:
   * Fetching all food items for display on the homepage.
   * Adding new food items via the admin panel.
   * Deleting food items via the admin panel.
3. **Cart Operations**:
   * Adding items to the cart and updating quantities.
   * Retrieving the current cart for a user.
4. **Order Operations**:
   * Creating new orders after successful checkout.
   * Fetching user-specific orders for order history.
   * Updating order status via the admin panel.

**Data Flow**

1. **Signup/Login**:
   * User submits form -> Server validates and processes -> MongoDB stores/retrieves user data -> Server responds.
2. **Browsing Food Items**:
   * User requests homepage -> Server fetches food items from MongoDB -> Server sends data to frontend -> Frontend renders items.
3. **Adding to Cart**:
   * User adds item to cart -> Server updates cart in MongoDB -> Server sends updated cart data to frontend -> Frontend updates UI.
4. **Checkout**:
   * User proceeds to checkout -> Server validates and processes payment via Stripe -> Server creates order in MongoDB -> Server sends order confirmation.
5. **Admin Operations**:
   * Admin adds/updates/deletes food items -> Server updates MongoDB -> Frontend reflects changes.
   * Admin updates order status -> Server updates MongoDB -> User sees updated status.

**4. Setup Instructions**

* **Prerequisites:** List software dependencies (e.g., Node.js, MongoDB).

**Node.js** (version 12.x or later)

**npm** (Node package manager, comes with Node.js)

**MongoDB** (preferably MongoDB Atlas for a cloud-based database)

**Express.js** (for backend server)

**React.js** (for frontend development)

* **Installation:** Step-by-step guide to clone, install degipendencies, and set up the environment variables.

**Step 1:** Clone the Repository

First, clone the repository from GitHub to your local machine.

git clone https://github.com/your-username/food-ordering-website.git

cd food-ordering-website

**Step 2:** Install Dependencies

Navigate to the project directory and install the required dependencies for both the frontend and backend.

cd backend

npm install

cd ../frontend

npm install

cd admin

npm install

**Step 3:** Set Up Environment Variables

Create a `.env` file in the backend directory and add the required environment variables.

cd ../backend

touch .env

2. Add the following environment variables to the `.env` file:

JWT\_SECRET="random#secret"

STRIPE\_SECRET\_KEY="Paste your stripe secret key here"

**Step 4:** Configure MongoDB Connection

Ensure you have your MongoDB Atlas connection string. Update the `db.js` file with your MongoDB connection details.

Backend (`db.js` file)

1. Open the `db.js` file and add your MongoDB connection string:

import mongoose from "mongoose";

export const connectDB = async () => {

await mongoose.connect('mongodb+srv://Gayathri:Gundu2003@cluster0.tiebsgz.mongodb.net/food-del')

.then(() => console.log("DB Connected"))

.catch(err => console.error("DB Connection Error: ", err));

};

**Step 5**: Start the Development Servers

Navigate to the backend directory and start the server. Then, do the same for the frontend and admin.

cd backend

npm start

cd ../frontend

npm run dev

cd admin

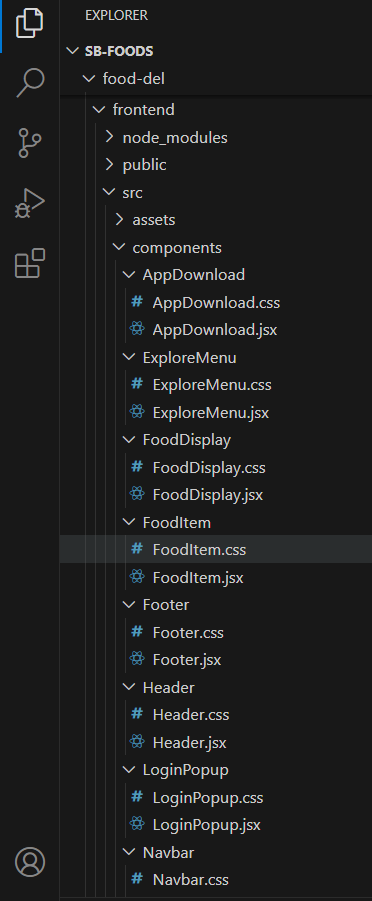
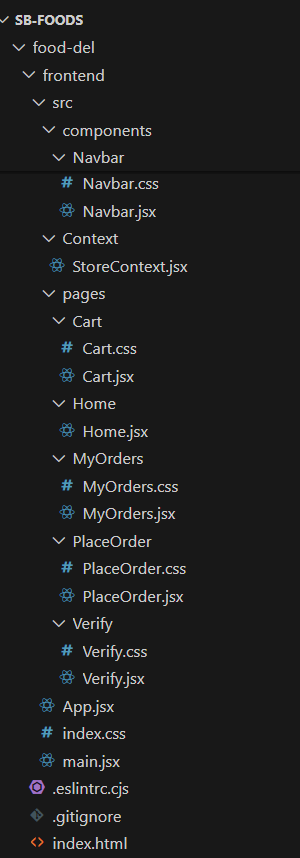
npm run dev

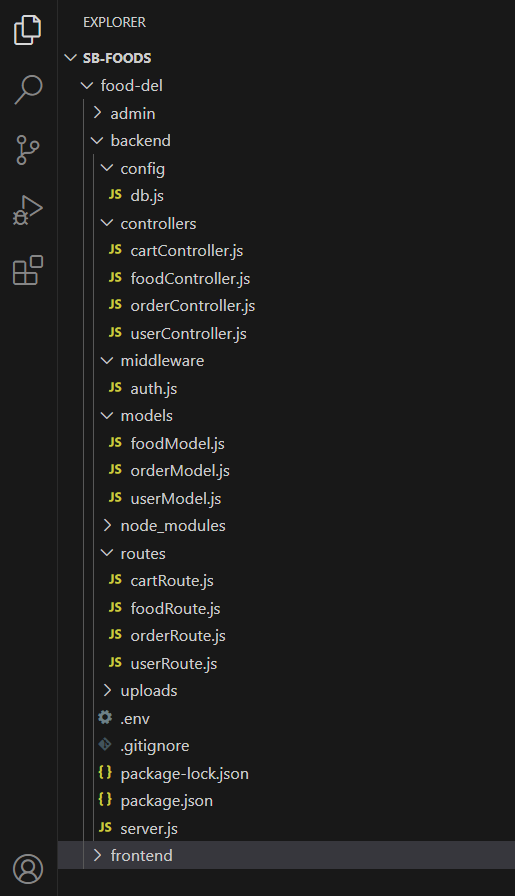
**Step 6:** Access the Application

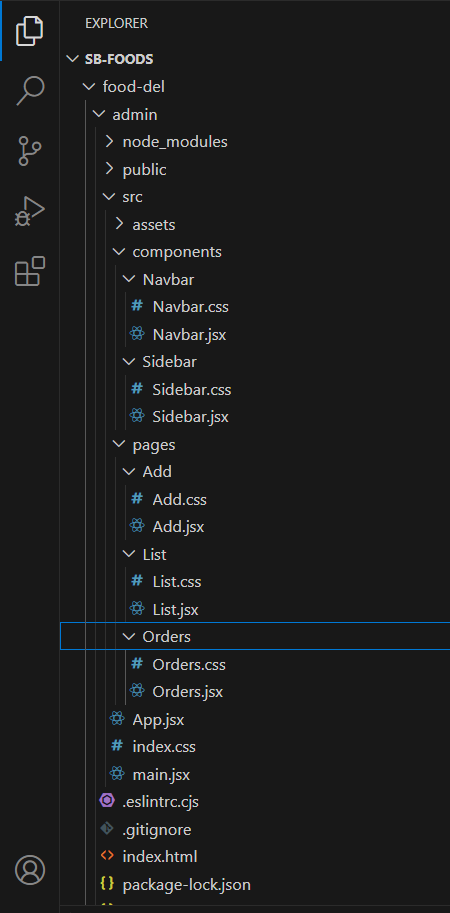
Open your web browser and navigate to ` http://localhost:5173` to access the frontend of the food ordering website. The backend server will be running on `http://localhost:8080`.and admin runs on to ` http://localhost:5174`

By following these steps, we can have the food ordering website up and running on your local machine, complete with all necessary dependencies and configurations.

**5. Folder Structure**

* **Client:** Describe the structure of the React frontend.
*  
* **Server:** Explain the organization of the Node.js backend.



* **Admin:** Explain the organization of theAdmin.
* 

**6. Running the Application**

* Provide commands to start the frontend and backend servers locally.
  + **Frontend:**npm run dev in the client directory.
  + **Backend:**npm start in the server directory.
  + **Admin:**npm run dev in the server directory.

**7. API Documentation**

### . ****1. User Endpoints****

#### ****POST /login****

* **Description:** Log in a user.
* **Request Body:**

json

Copy code

{

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

"password": "userpassword"

}

* **Response:**
  + **Success:**

json

Copy code

{

"success": true,

"token": "JWT\_TOKEN"

}

* + **Failure:**

json

Copy code

{

"success": false,

"message": "Invalid credentials"

}

#### ****POST /register****

* **Description:** Register a new user.
* **Request Body:**

json

Copy code

{

"name": "John Doe",

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

"password": "userpassword"

}

* **Response:**
  + **Success:**

json

Copy code

{

"success": true,

"token": "JWT\_TOKEN"

}

* + **Failure:**

json

Copy code

{

"success": false,

"message": "User already exists"

}

### ****2. Cart Endpoints****

#### ****POST /cart/add****

* **Description:** Add an item to the user's cart.
* **Request Body:**

json

Copy code

{

"userId": "USER\_ID",

"itemId": "ITEM\_ID"

}

* **Response:**
  + **Success:**

json

Copy code

{

"success": true,

"message": "Added To Cart"

}

* + **Failure:**

json

Copy code

{

"success": false,

"message": "Error"

}

#### ****POST /cart/remove****

* **Description:** Remove an item from the user's cart.
* **Request Body:**

json

Copy code

{

"userId": "USER\_ID",

"itemId": "ITEM\_ID"

}

* **Response:**
  + **Success:**

json

Copy code

{

"success": true,

"message": "Removed From Cart"

}

* + **Failure:**

json

Copy code

{

"success": false,

"message": "Error"

}

### ****3. Food Endpoints****

#### ****GET /foods****

* **Description:** List all food items.
* **Response:**
  + **Success:**

json

Copy code

{

"success": true,

"data": [

{

"name": "Food Item",

"description": "Description",

"price": 100,

"category": "Category",

"image": "image.jpg"

}

]

}

* + **Failure:**

json

Copy code

{

"success": false,

"message": "Error"

}

#### ****POST /food/add****

* **Description:** Add a new food item.
* **Request Body:**

json

Copy code

{

"name": "Food Item",

"description": "Description",

"price": 100,

"category": "Category",

"image": "image.jpg"

}

* **Response:**
  + **Success:**

json

Copy code

{

"success": true,

"message": "Food Added"

}

* + **Failure:**

json

Copy code

{

"success": false,

"message": "Error"

}

### ****4. Order Endpoints****

#### ****POST /order/place****

* **Description:** Place a new order with Stripe payment.
* **Request Body:**

json

Copy code

{

"userId": "USER\_ID",

"items": [

{

"name": "Food Item",

"price": 100,

"quantity": 1

}

],

"amount": 100,

"address": "Delivery Address"

}

* **Response:**
  + **Success:**

json

Copy code

{

"success": true,

"session\_url": "STRIPE\_CHECKOUT\_SESSION\_URL"

}

* + **Failure:**

json

Copy code

{

"success": false,

"message": "Error"

}

#### ****POST /order/place/cod****

* **Description:** Place a new order with Cash on Delivery.
* **Request Body:**

json

Copy code

{

"userId": "USER\_ID",

"items": [

{

"name": "Food Item",

"price": 100,

"quantity": 1

}

],

"amount": 100,

"address": "Delivery Address"

}

* **Response:**
  + **Success:**

json

Copy code

{

"success": true,

"message": "Order Placed"

}

* + **Failure:**

json

Copy code

{

"success": false,

"message": "Error"

}

**8. Authentication**

In this project, authentication and authorization are handled using JWT (JSON Web Tokens). Here's a detailed explanation of how these mechanisms are implemented:

**JWT (JSON Web Token)**

JWT is a compact, URL-safe means of representing claims to be transferred between two parties. In this project, JWT is used to securely transmit information between the client and the server.

**Token Generation**

When a user logs in or signs up, a JWT token is generated and sent back to the client. This token contains encoded user information and a signature to verify its authenticity.

**`LoginPopup.jsx`:**

const onLogin = async (e) => {

e.preventDefault()

let new\_url = url;

if (currState === "Login") {

new\_url += "/api/user/login";

} else {

new\_url += "/api/user/register"

}

const response = await axios.post(new\_url, data);

if (response.data.success) {

setToken(response.data.token)

localStorage.setItem("token", response.data.token)

loadCartData({token:response.data.token})

setShowLogin(false)

} else {

toast.error(response.data.message)

}

}

When the login or sign-up form is submitted, an API call is made to either `/api/user/login` or `/api/user/register`. If the response is successful, the token received is stored in local storage and the application state is updated.

**Token Storage**

The token is stored in the client's local storage to maintain the user's session across different pages and browser sessions.

**`LoginPopup.jsx`:**

localStorage.setItem("token", response.data.token)

**Authorization Middleware**

On the server side, an authorization middleware (`authMiddleware`) is used to protect routes that require authentication. This middleware verifies the JWT token sent in the request headers.

**`auth.js`:**

import jwt from 'jsonwebtoken';

const authMiddleware = async (req, res, next) => {

const { token } = req.headers;

if (!token) {

return res.json({success:false,message:'Not Authorized Login Again'});

}

try {

const token\_decode = jwt.verify(token, process.env.JWT\_SECRET);

req.body.userId = token\_decode.id;

next();

} catch (error) {

return res.json({success:false,message:error.message});

}

}

export default authMiddleware;

**Token Verification:**

- The middleware extracts the token from the request headers.

- If the token is missing, it responds with a "Not Authorized" message.

- If the token is present, it is verified using `jwt.verify` with the secret key (`process.env.JWT\_SECRET`).

- If the token is valid, the user's ID is decoded and attached to the request body (`req.body.userId`), and the request is passed to the next middleware or route handler.

- If the token is invalid, an error message is returned.

**Flow of Authentication and Authorization:**

1. User Registration/Sign-up:

- User submits the sign-up form with name, email, and password.

- Server registers the user, generates a JWT token, and sends it back to the client.

- Client stores the token in local storage.

2. User Login:

- User submits the login form with email and password.

- Server verifies the credentials, generates a JWT token, and sends it back to the client.

- Client stores the token in local storage.

3. Authenticated Requests:

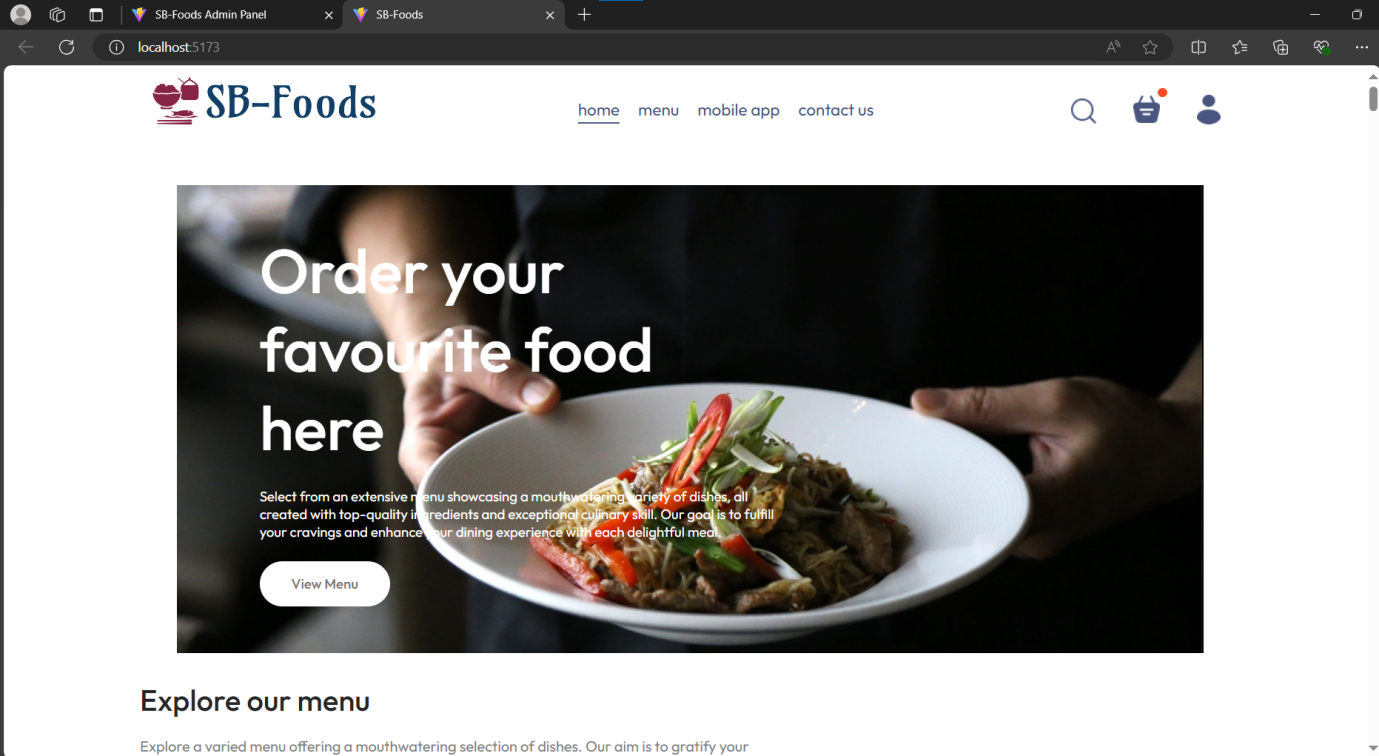
- For requests that require authentication, the client sends the JWT token in the request headers.

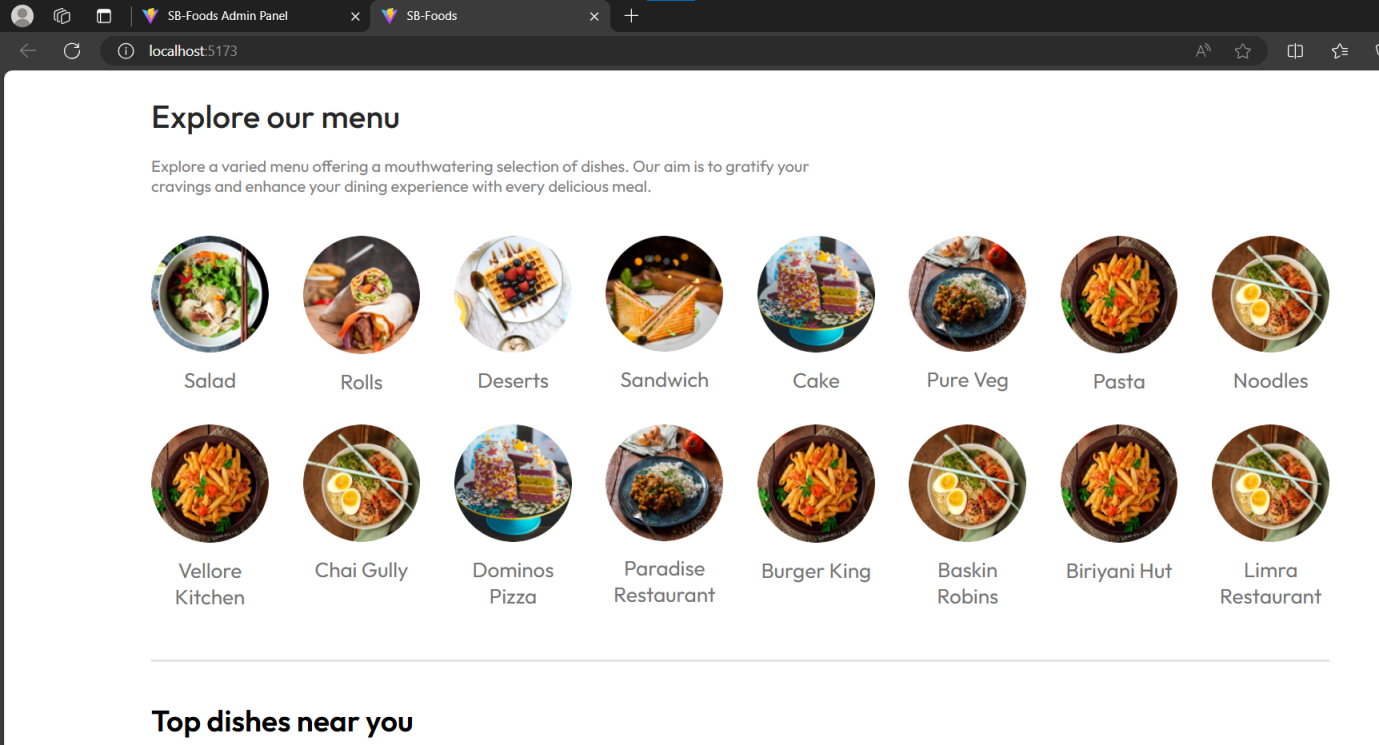
- Server verifies the token using the authorization middleware.

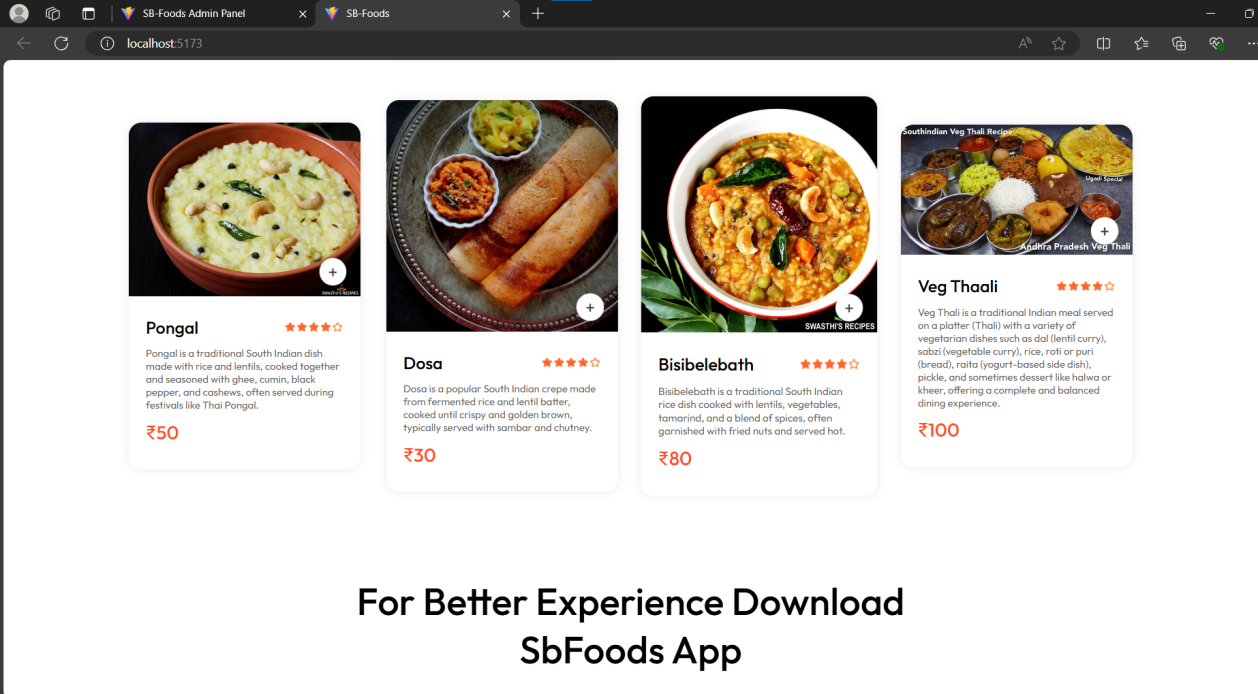
- If the token is valid, the request proceeds. If not, an error is returned.

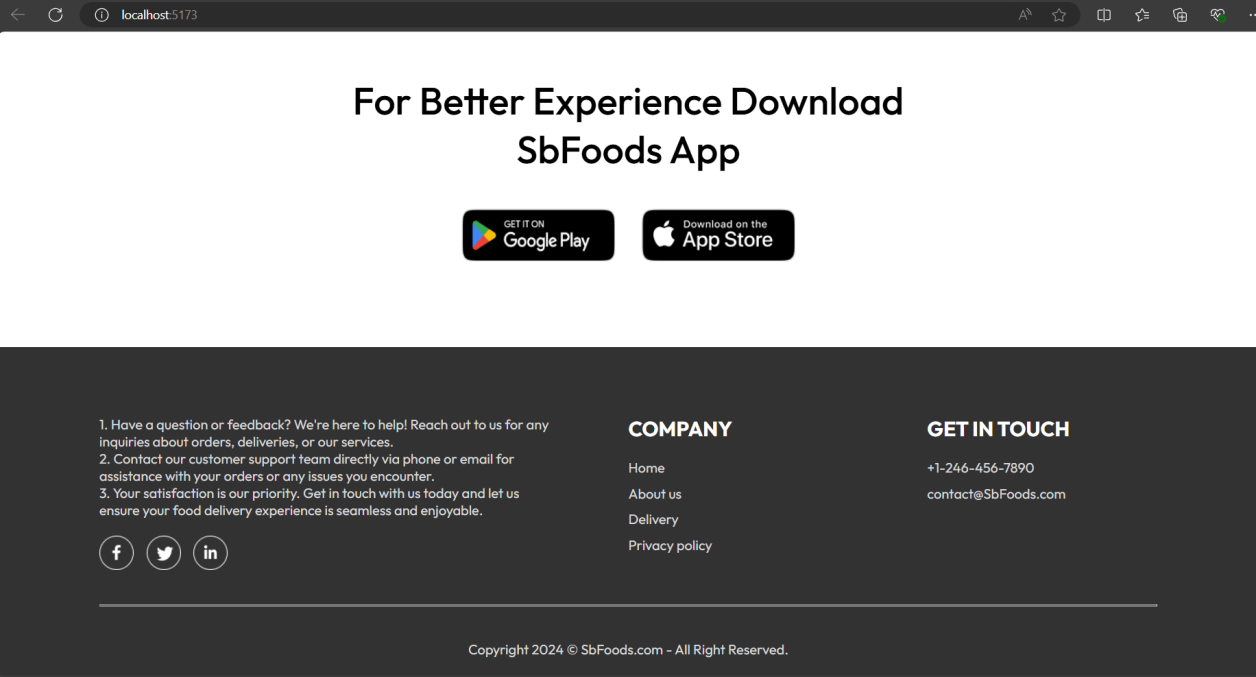
**9. User Interface**

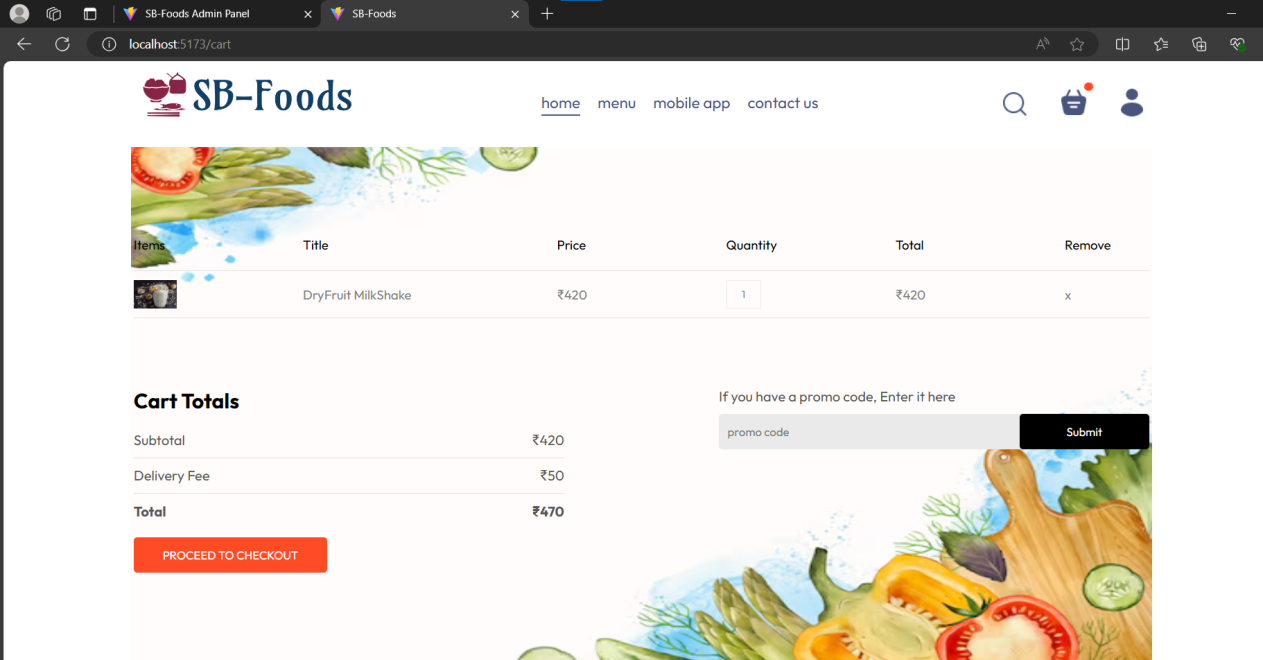
* Provide screenshots or GIFs showcasing different UI features.

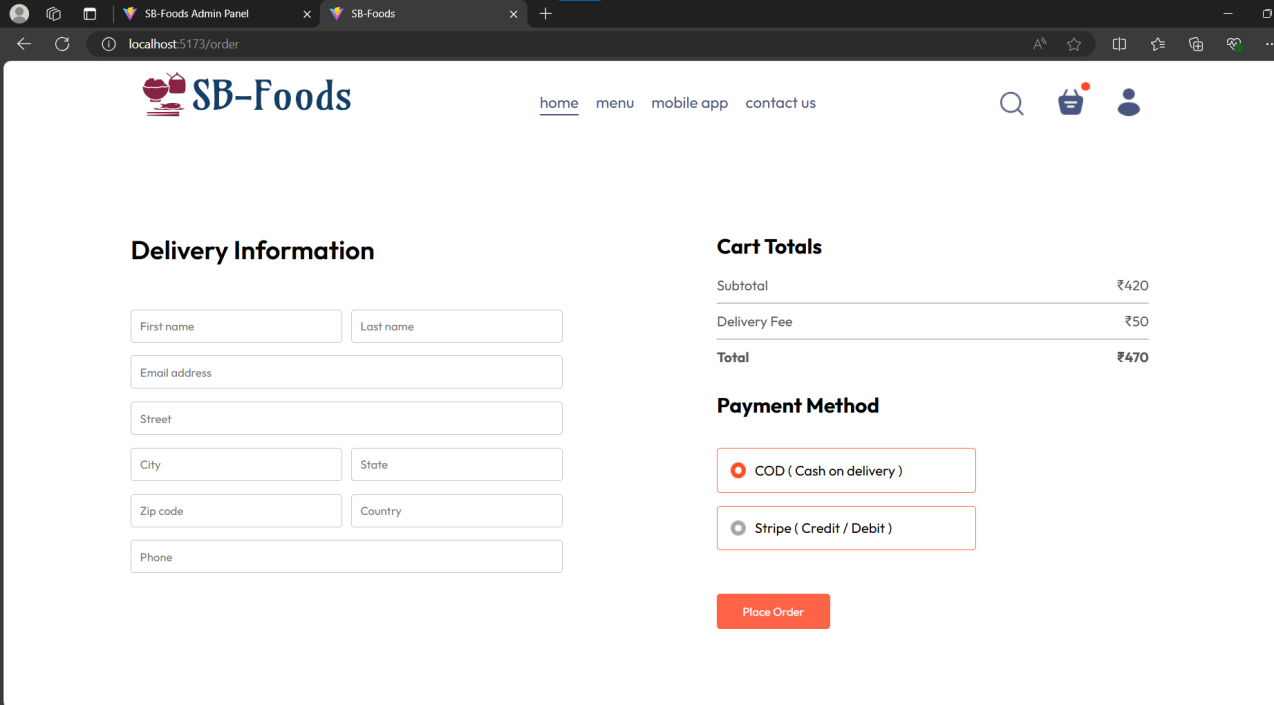


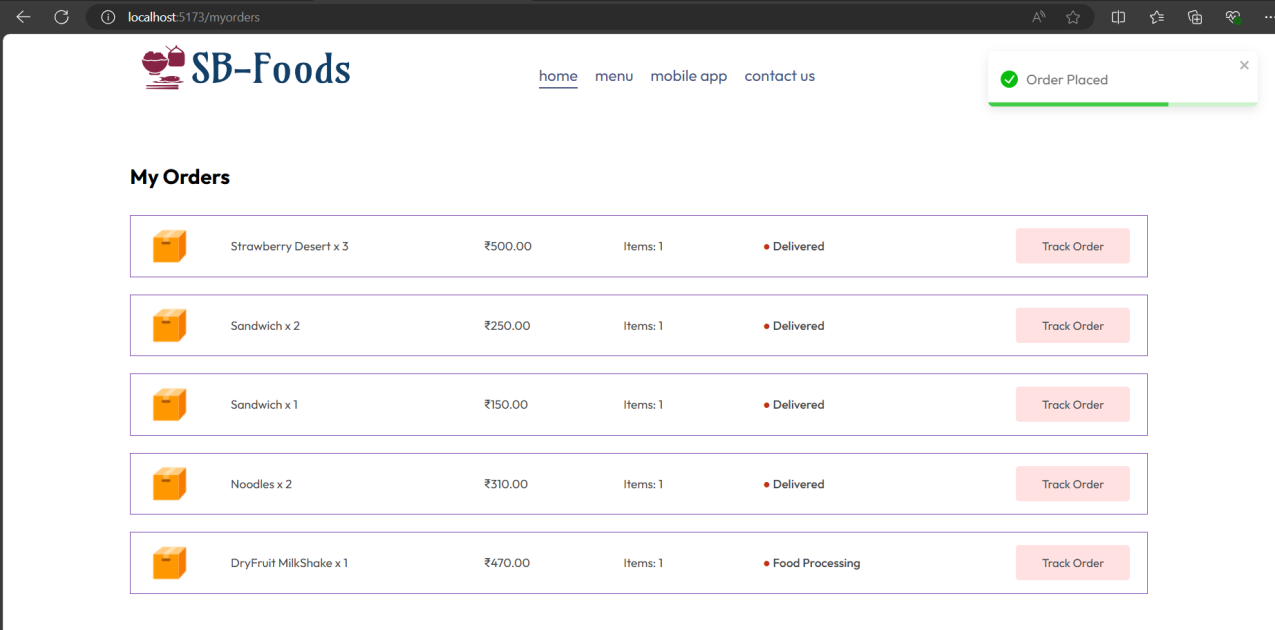


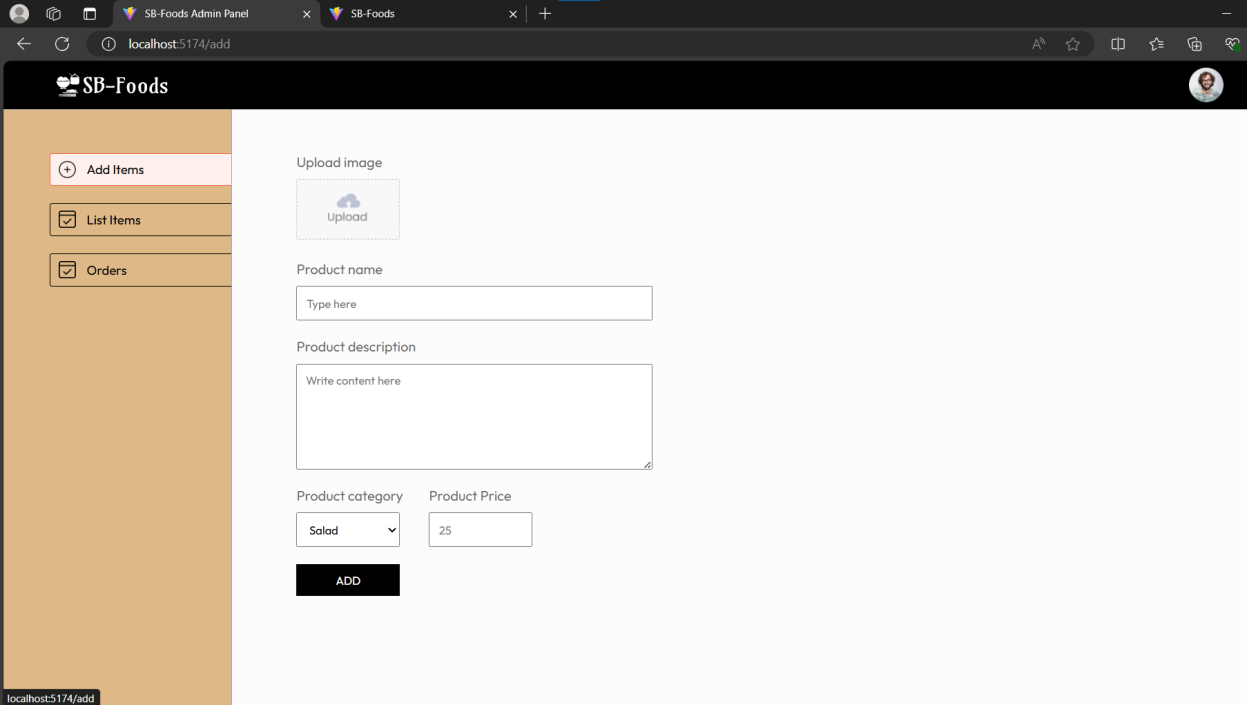


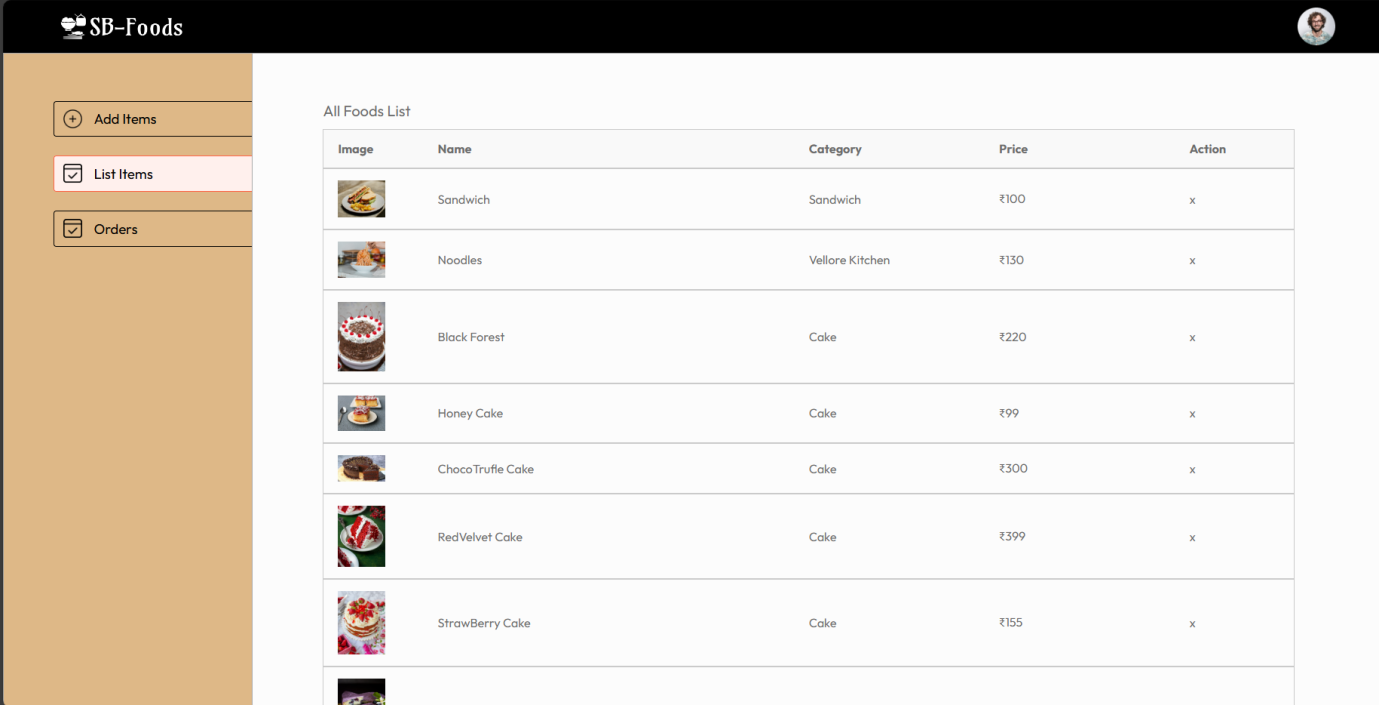


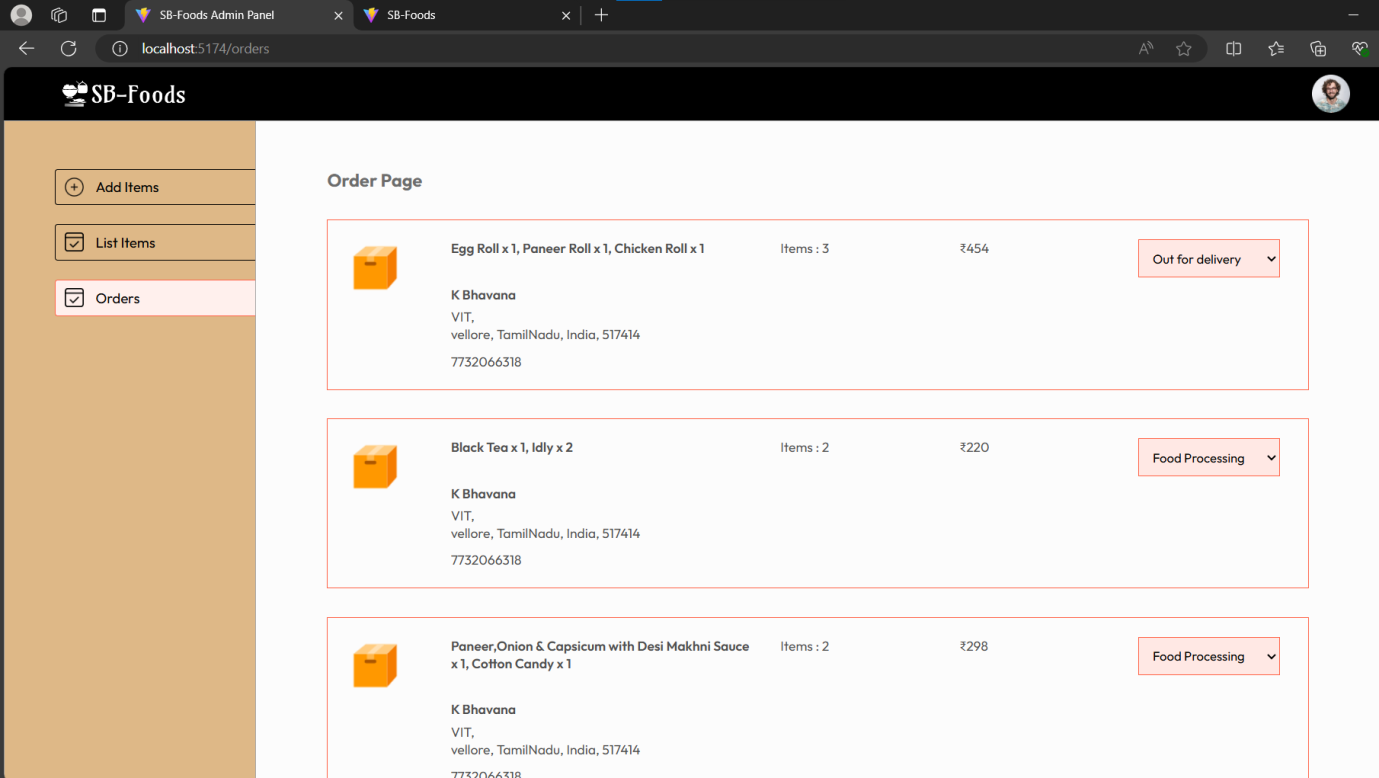












**10. Testing**

Testing is crucial in ensuring the reliability, performance, and security of the food ordering website. Here's an overview of the testing strategy and the tools used for this project:

#### ****1. Unit Testing****

**Purpose**: To verify that individual units or components of the application work as expected.

**Tools Used**:

* **Jest**: A JavaScript testing framework designed to ensure correctness of any JavaScript codebase.
* **React Testing Library**: A testing utility for React components that encourages good testing practices.

#### ****2. Integration Testing****

**Purpose**: To test the interaction between different modules or services to ensure they work together as expected.

**Tools Used**:

* **Supertest**: A library for testing Node.js HTTP servers.
* **Jest**: Also used for integration tests.

#### ****3. End-to-End (E2E) Testing****

**Purpose**: To test the application flow from start to finish to ensure that all parts of the application work together as expected.

**11. Screenshots or Demo**

* Provide screenshots or a link to a demo (if available) to showcase the application.

DEMO Link:

<https://drive.google.com/drive/folders/1Zf1hCEF5JeqHJL5JI-vQRiODkpzLEmG7>

**12. Known Issues**

* **Session Expiry**: Sometimes, user sessions expire prematurely, causing users to be logged out unexpectedly.
* **Cart Persistence**: Items added to the cart may not persist across different devices or sessions.
* **Responsive Design**: While the website is mostly responsive, some elements may not render correctly on very small or very large screens.
* **Performance**: The initial load time can be slow due to the number of assets and data being fetched.
* **Error Handling**: Some error messages are not user-friendly and do not provide enough context for troubleshooting.
* **Payment Gateway Integration**: The Stripe payment gateway occasionally fails to process payments due to network issues or incorrect configurations.
* **Database Connection**: Intermittent database connection issues have been observed, which can cause delays in fetching or saving data.
* **Cross-browser Compatibility**: Some features may not work as expected in older versions of certain browsers.
* **Accessibility**: The website lacks comprehensive accessibility features, making it difficult for users with disabilities to navigate.
* **Email Validation**: The email validation process is not very strict, allowing invalid email addresses to be used during registration.

**13. Future Enhancements**

* Outline potential future features or improvements that could be made to the project.
* **Improved User Authentication**: Implement OAuth-based authentication (e.g., Google, Facebook) for easier and more secure login.
* **Enhanced Cart Functionality**: Add features like saving the cart for later, sharing the cart, and syncing the cart across devices.
* **Optimized Performance**: Implement code splitting and lazy loading to improve initial load times.
* **Better Error Handling**: Provide more descriptive and user-friendly error messages throughout the application.
* **Comprehensive Testing**: Increase test coverage and introduce more end-to-end tests to ensure robustness.
* **Progressive Web App (PWA)**: Convert the website into a PWA to provide a more native app-like experience, including offline functionality.
* **Advanced Search and Filters**: Add more advanced search and filtering options to help users find food items more easily.
* **User Reviews and Ratings**: Allow users to leave reviews and ratings for food items, enhancing the decision-making process for other users.
* **Admin Analytics Dashboard**: Provide an analytics dashboard for admins to track sales, user activity, and other key metrics.
* **Real-time Order Tracking**: Provide real-time order tracking for users to know the status of their delivery.