**SB FOODS-FOOD ORDERING APP USING MERN**  
**PROJECT REPORT**

**Submitted by**

**TEAM NM ID: NM2024TMID00982**

**NAME: NM ID:**

BHUVANESWARI S 89D37373B5CCEEDA2A444AC504DEAC44

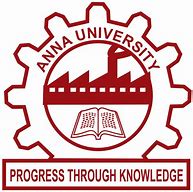
REENA SHARON D 9F68BFD0666AB599273442025A46D04B

CHANDURU P B1112CB01AB86AFBBED4CF0FD4EC3A9A

LEKHYA S 3CF42A1F707FD329C4F4BCC6C2998DC6

In partial completion towards the attainment of the degree of

**BACHELOR OF ENGINEERING IN  
COMPUTER SCIENCE AND ENGINEERING**



**LOYOLA INSTITUTE OF TECHNOLOGY**

**PALANCHUR, CHENNAI - 600123**

**ANNA UNIVERSITY**

**CHENNAI - 600025**

|  |  |
| --- | --- |
| S.NO | TABLE OF CONTENTS |
| 1. | **Introduction** |
| 2. | **Objective** |
| 3. | **Software Requirements** |
| 4. | **API Design** |
| 5. | **System design** |
| 6. | **Setup Instructions** |
| 7. | **Folder Structure** |
| 8. | **User Interface** |
| 9. | **Conclusion** |

**1.Introduction:**

SB Foods is a modern food ordering platform designed to provide users with a seamless and efficient way to browse, order, and enjoy their favorite meals from local restaurants. Built using the **MERN stack**, the application combines MongoDB, Express.js, React.js, and Node.js to create a highly interactive and scalable web application.

The app is aimed at enhancing the customer experience by offering a user-friendly interface, smooth navigation, real-time updates, and secure payment processing. Whether you're craving a pizza, sushi, or a home-cooked meal, SB Foods brings the convenience of ordering food right to your fingertips.

**Key Features:**

1. **User Authentication:** Secure login and registration process with session management.
2. **Restaurant Listings:** Browse through a wide range of restaurants, view menus, and check restaurant ratings.
3. **Real-Time Order Tracking:** Real-time updates of order status and estimated delivery time.
4. **Search & Filter:** Search for dishes by name, type, or category and filter by cuisine, price, and rating.
5. **Payment Integration:** Multiple payment options (credit card, PayPal, etc.) for easy and secure transactions.
6. **Admin Panel:** An intuitive dashboard for restaurant owners to manage their menu, orders, and customers.
7. **Responsive Design:** Fully responsive UI ensuring an optimal experience on both desktop and mobile devices.

**Technology Stack (MERN):**

* **MongoDB:** NoSQL database for storing user profiles, restaurant details, menus, and orders.
* **Express.js:** Web application framework for building the back-end API to handle HTTP requests.
* **React.js:** Front-end library for creating interactive and dynamic user interfaces.
* **Node.js:** JavaScript runtime to run server-side code, providing an efficient backend for the app.

SB Foods is designed to scale as a platform, allowing it to handle a growing number of users and restaurants. Whether you're an individual looking to satisfy your hunger or a restaurant owner looking to expand your reach, SB Foods provides an all-in-one solution to meet your food ordering needs.

**2.Objective for SB Foods - Food Ordering App (MERN Stack)**

The primary objective of the **SB Foods** food ordering app is to provide an easy-to-use, scalable platform for users to browse, order, and pay for food from local restaurants. Built using the **MERN stack** (MongoDB, Express.js, React.js, Node.js), the project aims to achieve the following:

1. **Seamless User Experience:** Offer an intuitive platform where customers can easily view menus, place orders, and track their food in real time.
2. **Secure Payment Integration:** Enable multiple secure payment options for users to easily complete their transactions.
3. **Restaurant Management:** Provide restaurant owners with an admin panel to manage menus, track orders, and interact with customers.
4. **Scalability:** Build a platform that can handle growing numbers of users, orders, and restaurants while maintaining performance and reliability.
5. **Mobile-First Design:** Ensure the app is responsive and optimized for both mobile and desktop devices.
6. **Real-Time Updates:** Implement features like real-time order tracking and push notifications to keep users informed.

This app will serve as a comprehensive solution for both users looking for a convenient food ordering experience and restaurant owners seeking to expand their customer reach.

**3.Software Requirements**

**1. Backend (Node.js / Express)**

* **Node.js** (v14 or later)
  + JavaScript runtime for building the server-side application.
* **Express.js** (v4 or later)
  + Web application framework for Node.js to handle routing and HTTP requests.
* **MongoDB** (Community Edition)
  + NoSQL database to store data related to users, orders, and restaurant information.
* **Mongoose** (v5 or later)
  + ODM (Object Data Modeling) library for MongoDB and Node.js to interact with the database.
* **JWT (jsonwebtoken)** (v8 or later)
  + For handling user authentication and generating JSON Web Tokens (JWT) for secure API access.
* **Bcrypt.js** (v5 or later)
  + For hashing passwords before storing them in the database.
* **dotenv** (v10 or later)
  + To manage environment variables securely, such as database credentials, API keys, and secrets.
* **Cors** (v2 or later)
  + Middleware to enable cross-origin resource sharing, allowing the frontend and backend to communicate.
* **Body-parser** (v1.19 or later)
  + Middleware to parse incoming request bodies in various formats, like JSON.
* **Stripe or PayPal SDK**
  + For handling online payment transactions securely.
* **Nodemon** (v2 or later)
  + Development tool for automatically restarting the backend server when code changes.

**2. Frontend (React.js)**

* **React.js** (v18 or later)
  + JavaScript library for building the user interface with reusable components.
* **React Router** (v6 or later)
  + For handling client-side routing and navigation between different pages (e.g., HomePage, CartPage, ProfilePage).
* **Axios** (v0.21 or later)
  + For making HTTP requests to the backend API (fetching data, submitting orders, etc.).
* **React Context or Redux**
  + For managing global state like user authentication status, cart items, etc.
* **React Bootstrap or Material-UI**
  + Component libraries for building responsive and visually appealing user interfaces.
* **React Helmet**
  + For managing metadata and dynamic page titles in the React app.
* **React Icons**
  + For adding icons to various components in the app (e.g., navigation bar, buttons).
* **SASS/SCSS or CSS Modules**
  + For writing modular and scalable styles for the frontend application.
* **Formik or React Hook Form**
  + For managing forms such as login, registration, and order forms.

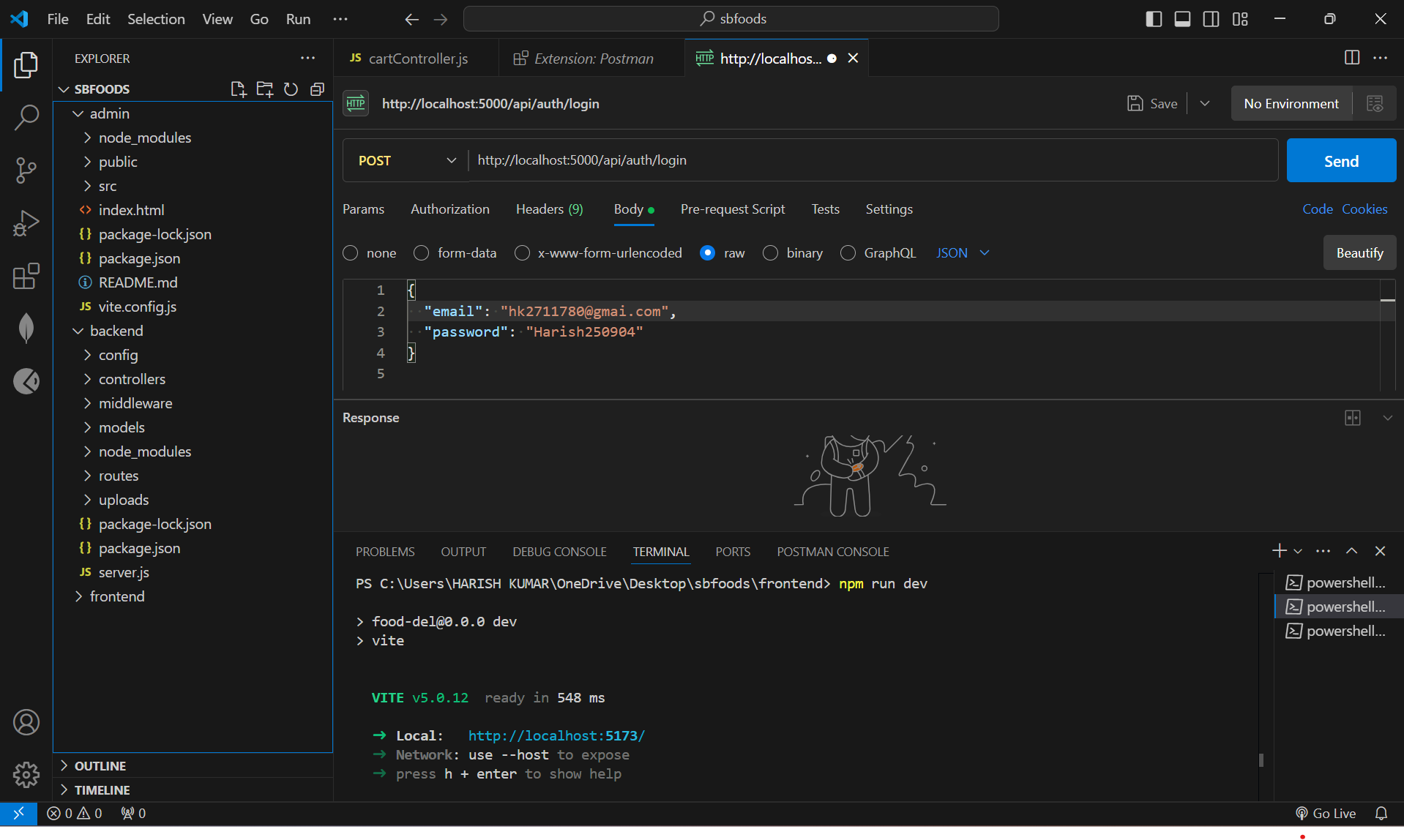
**3. Admin Panel (React.js)**

* **React.js** (same as frontend)
  + React-based admin dashboard for managing restaurants, orders, and users.
* **Chart.js or Recharts**
  + For visualizing data (e.g., order statistics, revenue reports) in the admin dashboard.
* **Admin Dashboard Libraries (React Admin or Material UI)**
  + For building the admin interface with pre-built components like tables, charts, and forms.

**4. Development Tools**

* **Visual Studio Code (VS Code)**
  + Code editor with support for JavaScript, React, Node.js, and relevant plugins for faster development.
* **Git**
  + Version control system for managing source code changes and collaboration.

**4.API Design**

****

**1. User Authentication & Management**

POST /api/auth/register

* Description: Register a new user (customer or admin).
* Request: User details (name, email, password, role).
* Response: Confirmation of successful registration.

POST /api/auth/login

* Description: Login an existing user.
* Request: User credentials (email, password).
* Response: JWT token for authentication.

GET /api/auth/me

* Description: Get details of the authenticated user.
* Headers: Authorization token.
* Response: User information (name, email, role).

**2. Restaurant Management (Admin)**

POST /api/restaurants

* Description: Add a new restaurant.
* Request: Restaurant details (name, address, contact).
* Response: Confirmation of restaurant creation.

GET /api/restaurants

* Description: Get a list of all restaurants.
* Response: List of restaurant details (name, address, contact).

GET /api/restaurants/

* Description: Get details of a specific restaurant.
* Response: Restaurant information (name, menu items, etc.).

**3. Menu Item Management (Admin)**

**POST /api/restaurants/**

/menu

* Description: Add a menu item to a restaurant.
* Request: Menu item details (name, description, price).
* Response: Confirmation of menu item addition.

PUT /api/restaurants/

/menu/

* Description: Update a menu item.
* Request: Updated menu item details.
* Response: Confirmation of menu item update**.**

**4. Order Management (User & Admin)**

POST /api/orders

* Description: Place a new order.
* Request: Order details (restaurant, items, quantity, payment method).
* Response: Confirmation of order placement.

GET /api/orders

* Description: Get all orders for the authenticated user.
* Headers: Authorization token.
* Response: List of orders (status, total amount).

PUT /api/orders/

* Description: Update the status of an order (Admin only).
* Request: New order status (e.g., "completed").
* Response: Confirmation of order status update.

**5. Payment Integration**

POST /api/payment

* Description: Process payment for an order.
* Request: Payment details (order ID, payment method).
* Response: Payment status (e.g., "completed").

Summary of Key Endpoints:

1. User Authentication: Registration, login, and fetching user data.
2. Restaurant Management: Add, update, and list restaurants.
3. Menu Management: Add, update menu items for restaurants.
4. Order Management: Place, view, and manage orders.
5. Payment: Process payments for orders.

This API design provides the core endpoints needed for the SB Foods - Food Ordering App, focusing on functionality and data flow without any code details. This structure covers user management, restaurant and menu handling, orders, and payment processing.

**5.System design**

**Updated System Design for the Online Learning Platform**

**1. Architecture Overview**

The architecture for the Online Learning Platform follows a **Microservices** or **Monolithic** structure depending on scalability requirements. We will assume a **Microservices** architecture here, as it provides better modularity, scalability, and easier integration with third-party services.

**1.1 Three-Tier Architecture:**

* **Frontend (Client-Side)**:
  + **React.js** with **Redux** for state management or **React Context API** for smaller applications.
  + The frontend will handle user interface (UI), authentication, and communication with backend APIs.
  + Responsive design for accessibility on desktop and mobile devices.
* **Backend (Server-Side)**:
  + **Node.js** with **Express.js** for creating RESTful APIs.
  + Business logic, API routes, user authentication, and data processing.
  + **JWT (JSON Web Tokens)** for secure token-based authentication.
  + **Third-party services** integration (e.g., email notifications, payments via Stripe or PayPal).
  + **WebSocket** or **Socket.io** for real-time notifications or chats in a course setting.
* **Database**:
  + **MongoDB** for NoSQL data storage.
  + Data modeling based on **document-oriented** collections that can scale horizontally and handle complex relationships.
  + **Mongoose** for Object Data Modeling (ODM) to interact with MongoDB more efficiently.

**2. Database Design**

**2.1 Collections and Schemas:**

**1. User Collection**:

* **Schema Fields**:
  + userId: Unique identifier (auto-generated).
  + name: Full name of the user.
  + email: User email (unique).
  + password: Encrypted password (use **bcrypt.js** for hashing).
  + role: User role (Enum: student, instructor, admin).
  + coursesEnrolled: Array of course IDs (for students).
  + coursesCreated: Array of course IDs (for instructors).
  + profilePic: URL of the profile picture.
  + bio: Optional short bio for the user.
  + lastLogin: Timestamp of the last login (for user activity tracking).
  + notifications: Array of unread notification IDs.

**2. Course Collection**:

* **Schema Fields**:
  + courseId: Unique identifier for each course.
  + title: Course title (e.g., "Mastering Python").
  + description: A brief description of the course.
  + instructorId: Reference to the userId of the course creator (instructor).
  + studentsEnrolled: Array of student userIds who are enrolled in the course.
  + content: Array of module IDs. Each module has:
    - Module name/title.
    - List of lessons (e.g., videos, text content, assignments).
    - Type of lesson (video, quiz, text, etc.).
  + tags: Array of course tags (e.g., programming, data science).
  + price: Course price (if paid).
  + category: Course category (e.g., Tech, Arts, Business).
  + isPublished: Boolean to indicate if the course is publicly available.
  + rating: Average course rating based on user feedback.

**3. Feedback Collection**:

* **Schema Fields**:
  + feedbackId: Unique identifier for each feedback.
  + courseId: Reference to the courseId from the Course collection.
  + userId: Reference to the userId of the student who provides the feedback.
  + rating: Numerical rating (1-5).
  + comment: Optional written feedback.
  + timestamp: Date and time when the feedback was provided.

**4. Lesson Content Collection**:

* **Schema Fields**:
  + lessonId: Unique identifier for each lesson.
  + moduleId: Reference to the module it belongs to.
  + type: Type of lesson (video, text, quiz, etc.).
  + contentUrl: URL of the lesson content (for videos, links to videos; for text, the content itself).
  + duration: Length of the lesson in minutes (for video lessons).
  + quiz: Array of quiz questions, if applicable.
  + isActive: Boolean to indicate if the lesson is active or archived.

**5. Payment Collection** (if applicable for paid courses):

* **Schema Fields**:
  + paymentId: Unique identifier for the payment.
  + userId: Reference to the userId of the student who made the payment.
  + courseId: Reference to the courseId the payment is for.
  + amount: The amount paid.
  + status: Payment status (e.g., completed, pending).
  + paymentMethod: Type of payment (e.g., credit card, PayPal).
  + paymentDate: Date and time when the payment was made.
  + transactionId: Transaction ID from the payment provider.

**3. Authentication and Authorization**

* **User Authentication**:
  + **JWT-based authentication** for secure token handling and stateless session management.
  + Authentication middleware on the backend to protect sensitive routes (e.g., creating a course, enrolling in a course).
* **Roles**:
  + **Student**: Can browse courses, enroll in them, complete lessons, and leave feedback.
  + **Instructor**: Can create and manage courses, track students' progress, and receive feedback.
  + **Admin**: Manages users, courses, and platform settings.

**4. Frontend (Client-Side)**

* **React.js**:
  + Dynamic UI to display courses, user profiles, and content.
  + Use **React Router** for navigation between pages (Home, Course Detail, User Profile, etc.).
  + **State Management**: Use **Redux** or **React Context API** to manage the global state (user data, course enrollment, feedback).
  + **Axios** for API calls to the backend services (courses, user profiles, feedback submission).
  + **Real-time updates** (optional): For course progress or notifications, you could implement **WebSockets** using **Socket.io**.
* **UI/UX**:
  + **Material UI** or **Tailwind CSS** for consistent, responsive styling.
  + User-friendly dashboard for instructors to manage their courses and track student progress.
  + Interactive course player (video, quizzes, assignments).
  + Real-time chat or discussion board for students and instructors.

**5. Backend (Server-Side)**

* **Node.js** with **Express.js** to handle HTTP requests and define API routes.
* **API Routes**:
  + **/auth**: For login, signup, and user authentication (JWT).
  + **/courses**: CRUD operations for courses (GET, POST, PUT, DELETE).
  + **/enrollment**: To enroll students in courses (POST).
  + **/feedback**: To submit or fetch course feedback (POST, GET).
  + **/user**: For user profile management (GET, PUT).
* **Security Considerations**:
  + **Rate limiting** for API endpoints to prevent abuse.
  + **Input validation** using libraries like **Joi** or **express-validator** to prevent malicious data.
  + **CORS** configuration to handle cross-origin requests.

**6. Scalability & Performance**

* **Horizontal Scalability**:
  + Use cloud platforms like **AWS** or **Azure** to deploy services and databases.
  + Consider using **containerization (Docker)** and **Kubernetes** for managing microservices.
* **Caching**:
  + Use **Redis** for caching frequently accessed data, like course listings and user profiles, to reduce database load.
* **Load Balancer**:
  + A **load balancer** to distribute traffic evenly across multiple instances of your backend services.

**6.Setup Instructions for Food Ordering Website**

**Prerequisites:**

Before you begin setting up the food ordering website, ensure you have the following software dependencies installed on your local machine:

* **Node.js** (version 12.x or later)
* **npm** (Node package manager, which comes with Node.js)
* **MongoDB** (preferably MongoDB Atlas for a cloud-based database)
* **Express.js** (for backend server development)
* **React.js** (for frontend development)
* **Stripe** (if implementing payment integration)

**Installation: Step-by-Step Guide**

**Step 1: Clone the Repository**

1. Open your terminal or command prompt.
2. Clone the GitHub repository to your local machine by running the following command:

bash

Copy code

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

cd food-ordering-website

**Step 2: Install Dependencies**

Navigate to the backend, frontend, and admin directories and install the required dependencies for each:

1. Install the backend dependencies:

bash

Copy code

cd backend

npm install

1. Install the frontend dependencies:

bash

Copy code

cd ../frontend

npm install

1. Install the admin dependencies:

bash

Copy code

cd ../admin

npm install

**Step 3: Set Up Environment Variables**

Create a .env file in the backend directory to store sensitive data and configurations like your secret keys, JWT secret, etc.

1. Navigate to the backend directory:

bash

Copy code

cd ../backend

1. Create the .env file:

bash

Copy code

touch .env

1. Open the .env file and add the following environment variables:

env

Copy code

JWT\_SECRET="random#secret" # Replace with a strong secret key

STRIPE\_SECRET\_KEY="Paste your Stripe secret key here" # Replace with your Stripe secret key

**Note**: Make sure to replace the JWT\_SECRET and STRIPE\_SECRET\_KEY with your actual values. These keys are used for secure JWT token signing and Stripe payment processing.

**Step 4: Configure MongoDB Connection**

You will need to set up a MongoDB Atlas cluster or use a local MongoDB database.

1. Ensure you have a MongoDB Atlas account and a connection string for your cluster (or use a local MongoDB instance).
2. In the backend directory, open the db.js file and update it with your MongoDB connection string:

js

Copy code

import mongoose from "mongoose";

export const connectDB = async () => {

await mongoose.connect('mongodb+srv://<username>:<password>@cluster0.tiebsgz.mongodb.net/food-del')

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

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

};

**Note**: Replace the '<username>' and '<password>' placeholders with your MongoDB Atlas credentials.

**Step 5: Start the Development Servers**

Once all dependencies are installed and environment variables are set up, it's time to start the development servers for the backend, frontend, and admin sections.

1. **Start the backend server**:

bash

Copy code

cd backend

npm start

1. **Start the frontend development server**:

bash

Copy code

cd ../frontend

npm run dev

1. **Start the admin panel server**:

bash

Copy code

cd ../admin

npm run dev

**Step 6: Access the Application**

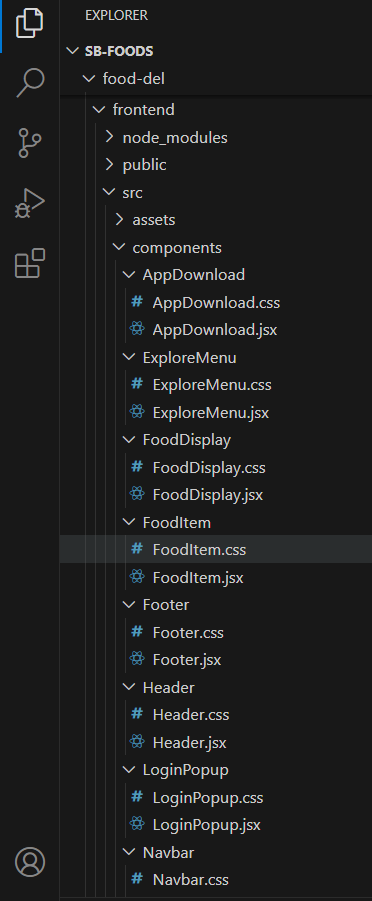
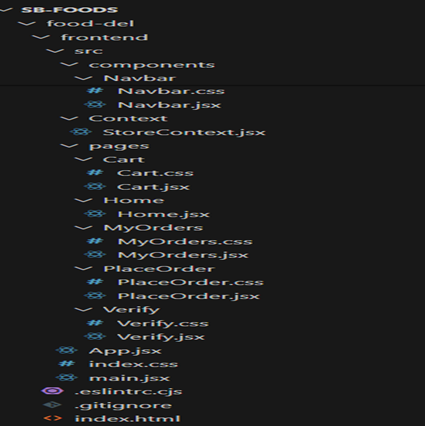
Once the servers are running, you can access the different parts of the application via your web browser:

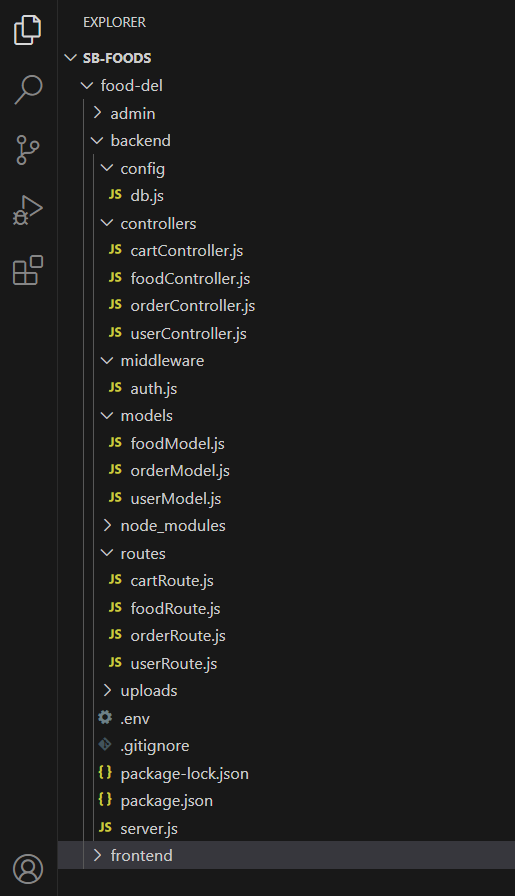
* **Frontend (Customer-facing website)**: Open your browser and navigate to:
  + http://localhost:5173
* **Backend (API server)**: The backend server should be running on:
  + http://localhost:8080
* **Admin Panel (Restaurant and Admin Dashboard)**: The admin panel should be running on:
  + http://localhost:5174

**Troubleshooting Tips:**

* **MongoDB Connection Errors**:
  + Make sure your MongoDB Atlas cluster is set to allow connections from your IP address.
  + Double-check your MongoDB credentials in the .env file.
* **Port Conflicts**:
  + If the default ports (e.g., 5173, 8080, 5174) are already in use, you can modify them by editing the respective package.json file or the backend server configuration.
* **Stripe Integration Issues**:
  + Ensure your STRIPE\_SECRET\_KEY is correct in the .env file.
  + If testing payments, use the **Stripe test keys** to avoid real charges.

**7. Folder Structure**

****

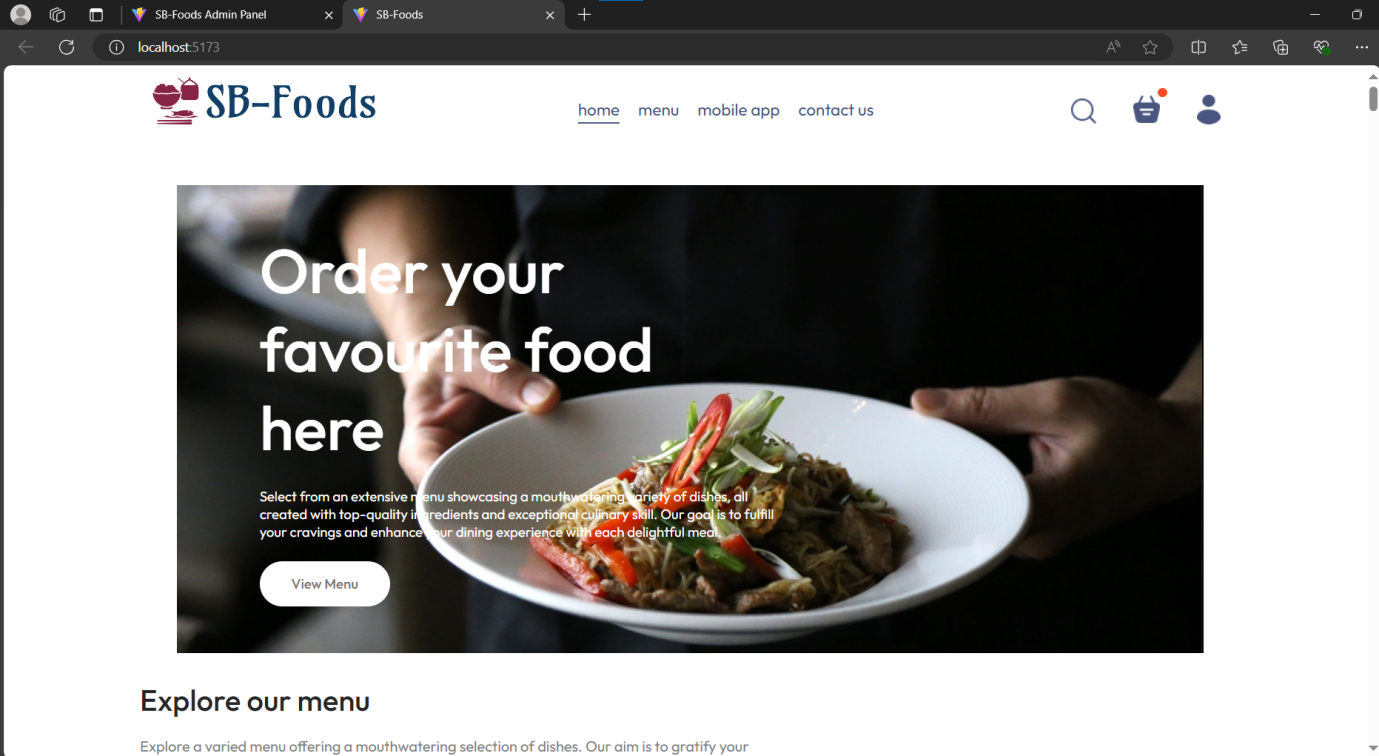
**Server:** Explain the organization of the Node.js backend.

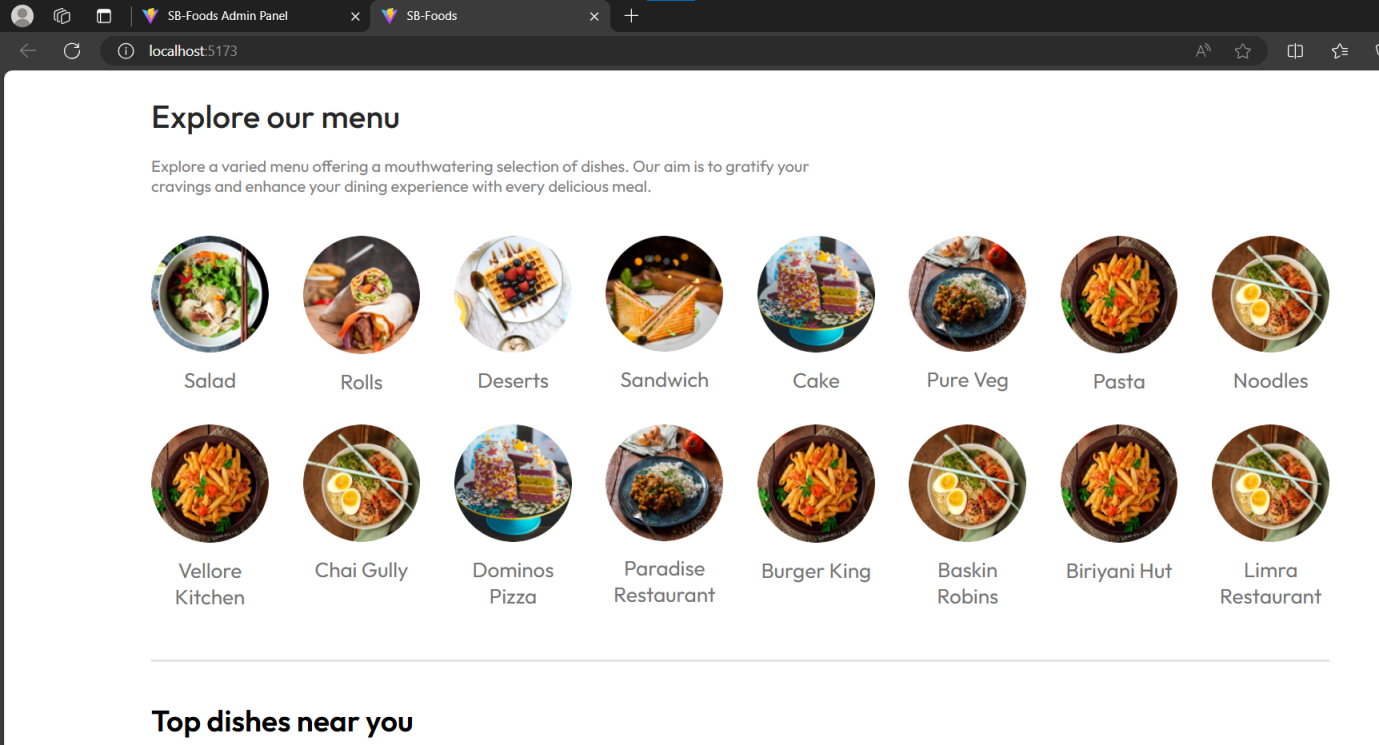
**6. Running the Application**

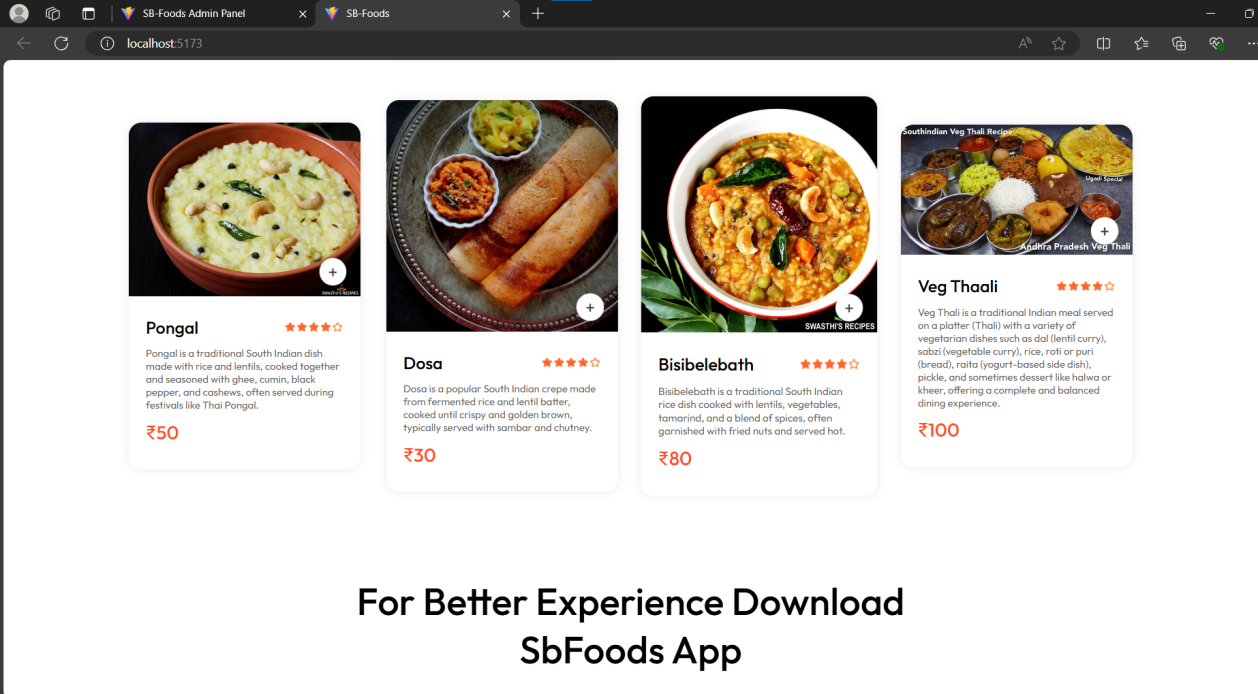
* Provide commands to start the frontend and backend servers locally.
  + **Frontend:**npm run dev in the client directory.
  + **Backend:**npm start in the server directory.
  + **Admin:**npm run dev in the server directory.

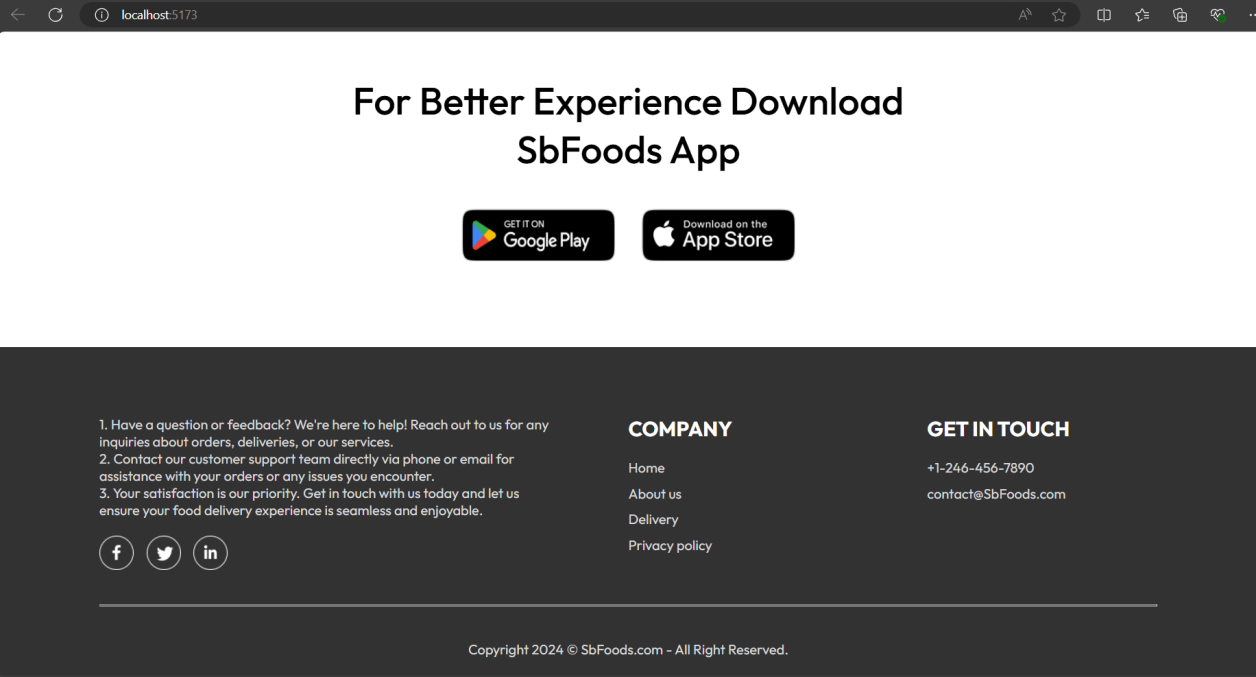
**8.User Interface**

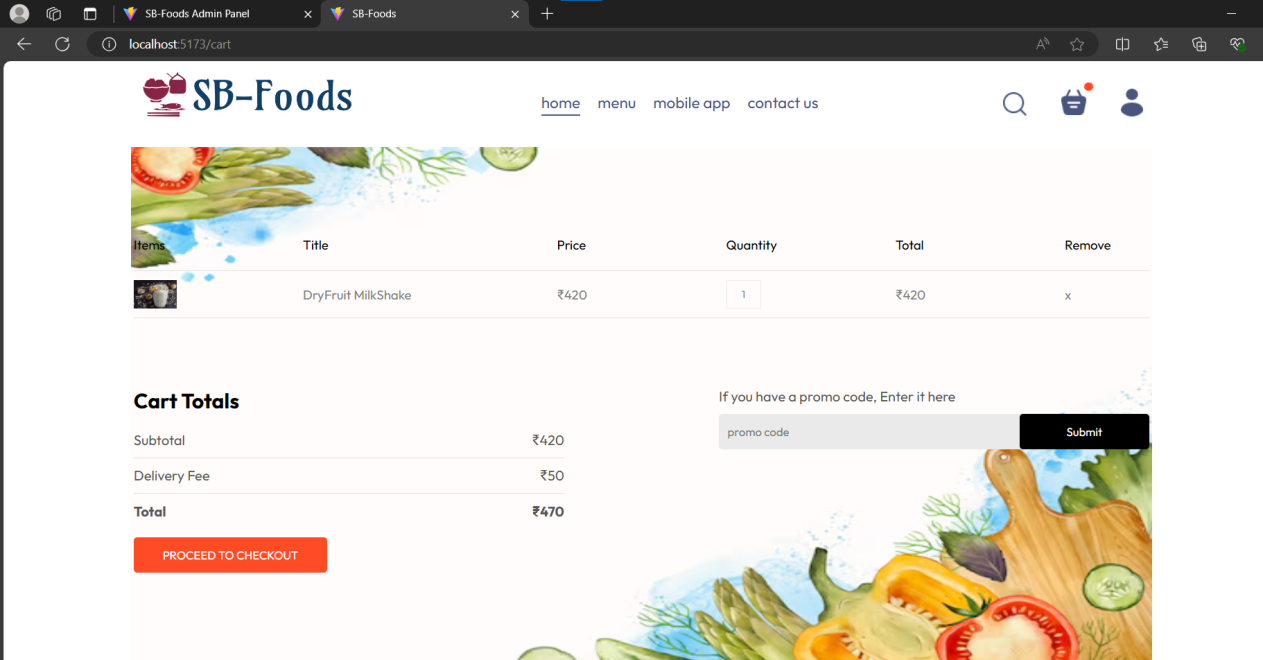
* Provide screenshots or GIFs showcasing different UI features.

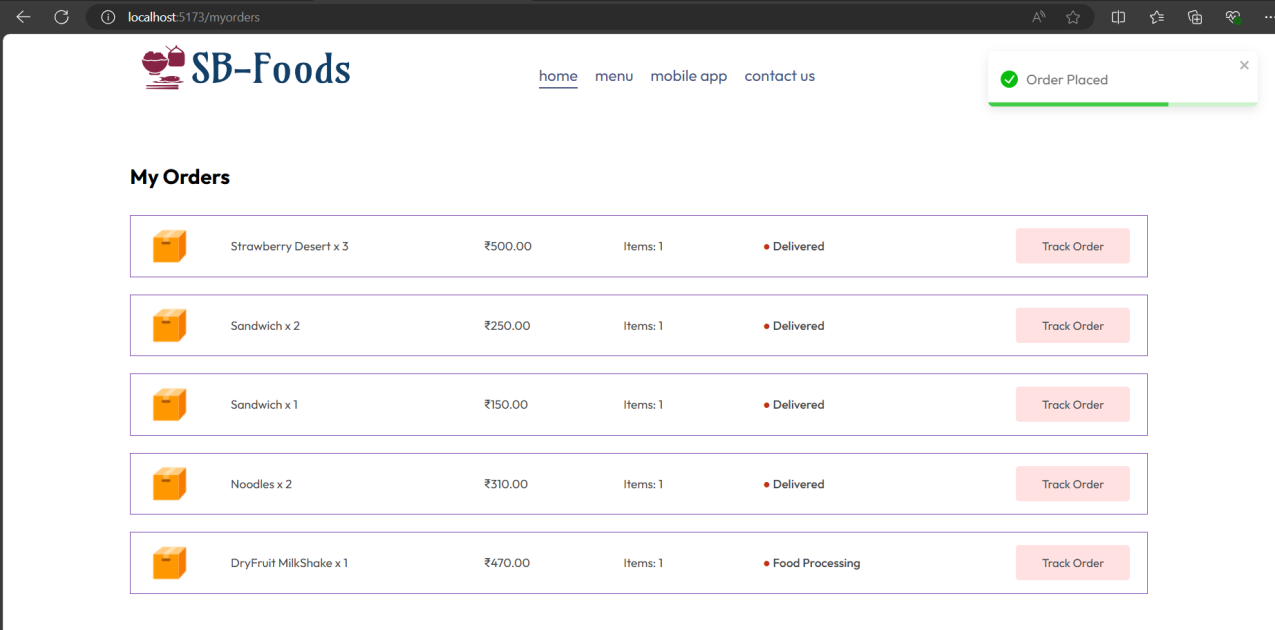
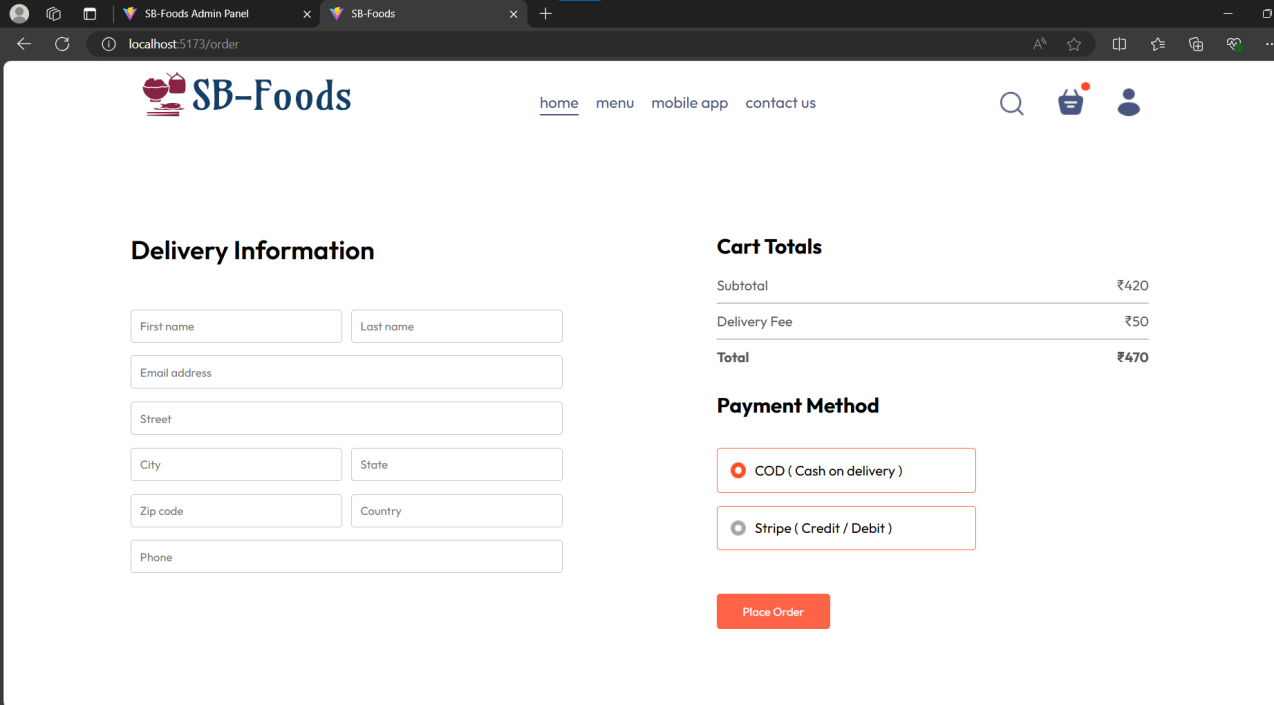


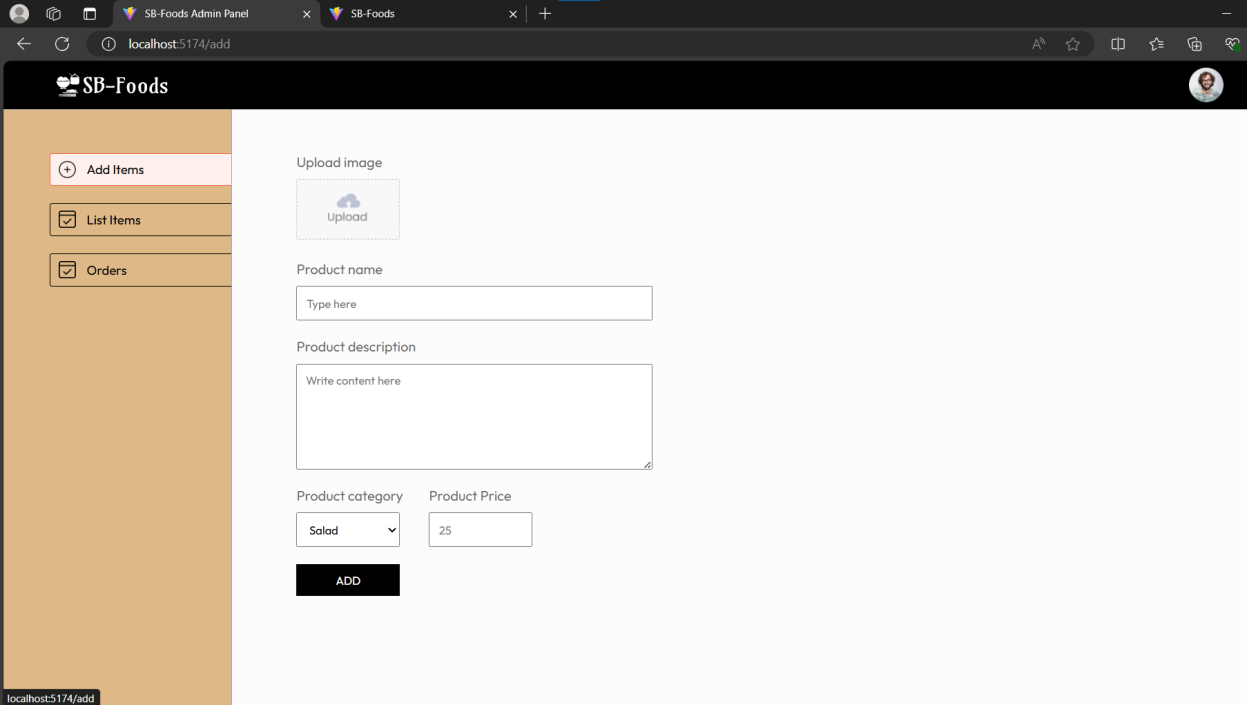


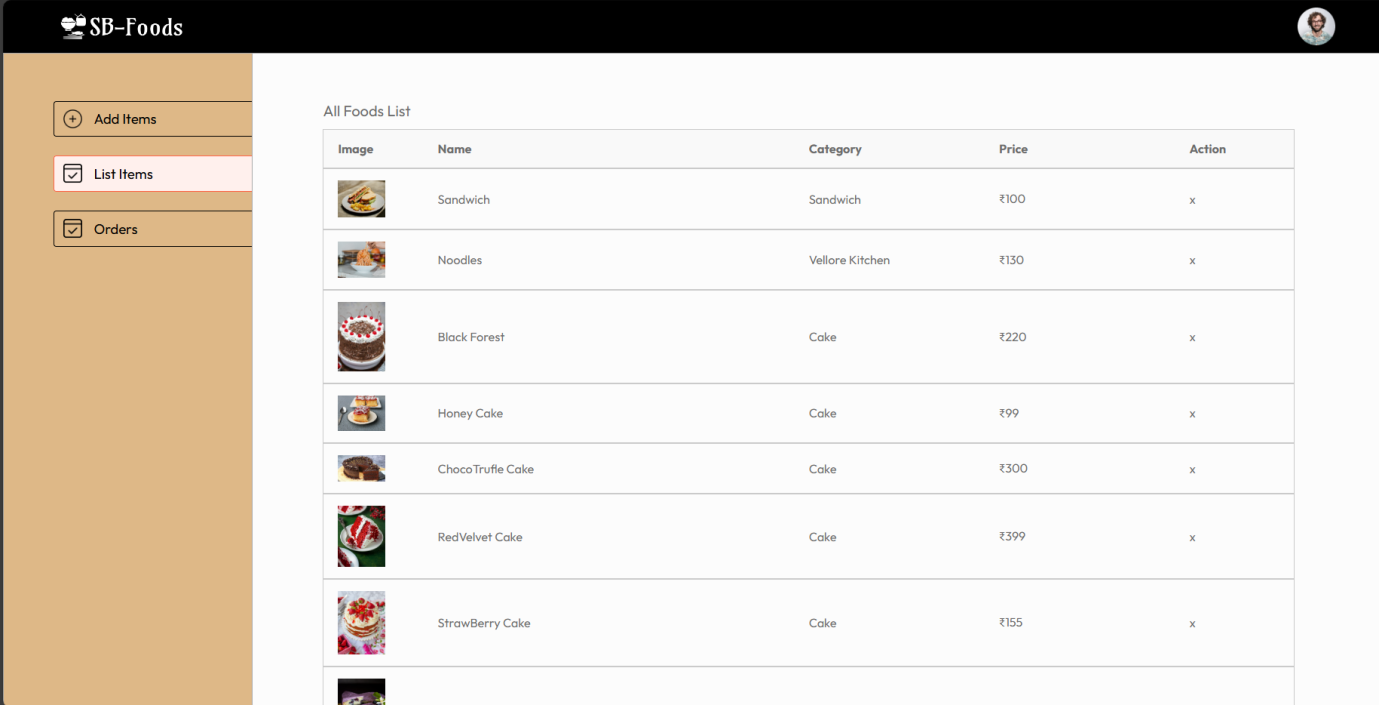


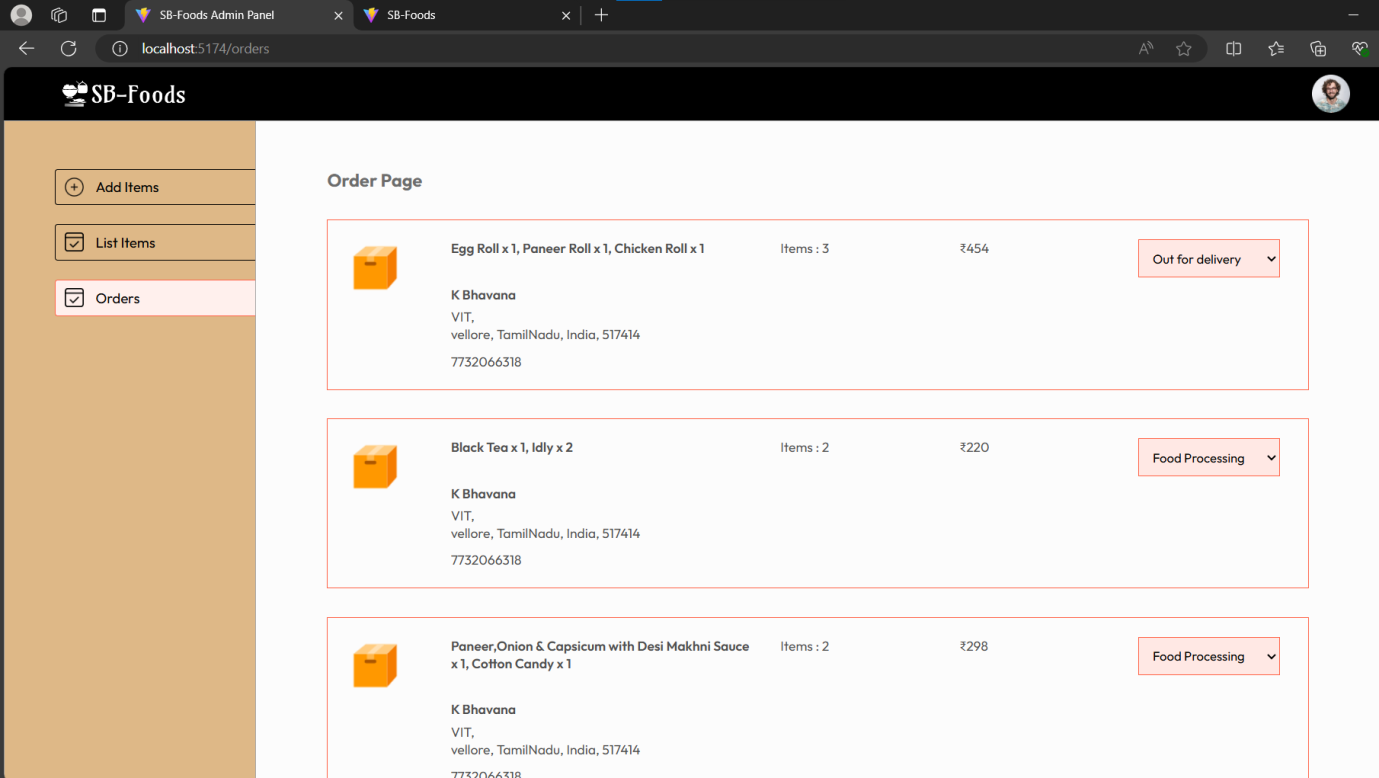












**10.CONCLUSION:**

The Food Ordering Website project is a full-stack MERN application (MongoDB, Express.js, React.js, Node.js) designed to connect customers, restaurants, and administrators, offering features such as secure user authentication, menu browsing, order placement, and real-time order tracking. The backend is built with Node.js and Express.js, while the frontend utilizes React.js for dynamic user interfaces, and MongoDB serves as the database for storing user data, orders, and restaurant information. Integrated Stripe payment processing enables secure transactions, and restaurant owners can manage their menus and orders. The system is scalable and user-friendly, with potential for future enhancements like mobile app support, advanced search, and real-time updates. With detailed setup instructions and a robust tech stack, this project provides a solid foundation for a modern online food delivery platform and serves as an excellent learning resource for developers working with the MERN stack.