SB FOOD ORDERING

1.Introduction

Project Title: SB Foods: Online Food Ordering and Management System

Team Members:

1. **LekhaShree K A** - Backend
2. **Poorany Priyadharshiny T** – Frontend, Backend
3. **Sathya Devi N** – Admin, Backend
4. **Sunitha Chanda** – UI/UX, Frontend
5. **Divya P** – Backend, Testing

2.Project Overview

The **SB Food Ordering Platform** is a comprehensive and dynamic food delivery solution built with the MERN stack. It aims to streamline the process of ordering food, ensuring a seamless experience for customers, restaurant partners, and delivery personnel. The platform emphasizes scalability, responsiveness, and secure transaction handling to deliver a reliable service.

Features:

- Comprehensive Product
- Catalog: Browse food items across diverse restaurants with detailed descriptions, reviews, pricing, and discounts.
- Secure Checkout Process: Ensure safe transactions with a seamless user interface.
- Order Details and History: Track orders, including payment methods, shipping addresses, and order summaries.
- Admin Management: Control over users, products, and restaurant approvals.
- Restaurant Dashboard: Manage listings, monitor order activity, and view order details.

Frontend (React.js) :

Offers a modern and responsive interface for browsing restaurants, viewing menus, placing orders, and managing user accounts, ensuring an optimal user experience across devices.The frontend is built with React.js, employing reusable components, state management using React Context API or Redux,

and responsive design with Material UI and CSS. Pages include:

- Home
- Product Catalog
- Cart
- Checkout
- Order History

Backend (Node.js & Express.js) :

Provides robust API endpoints for user authentication, order placement, payment processing, and real-time updates. It ensures smooth communication between the frontend and database, delivering reliable services. The backend uses Node.js with Express.js to handle REST API calls, middleware functions, and server-side operations. Key functionalities include:

- User authentication (JWT-based).
- Order and cart management APIs.
- Admin operations for user and product management.

Restaurant Dashboard :

Empowers restaurant partners with tools to manage menus, track orders, update availability, and analyze sales data to optimize operations.

Customer Dashboard :

Simplifies food ordering with features for browsing restaurants, tracking orders, saving favorite dishes, and accessing order history while providing personalized recommendations.

Admin Panel :

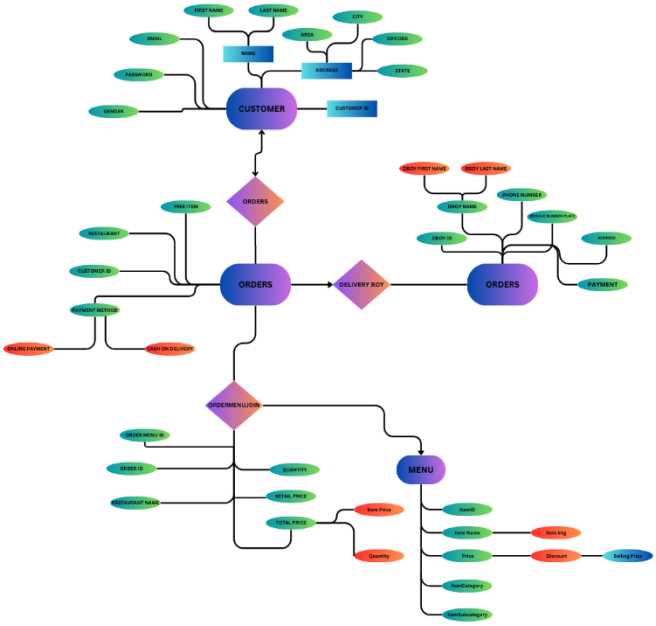
Centralized management of the platform, including overseeing user and restaurant accounts, managing orders, monitoring system analytics, and resolving customer feedback.

Database (MongoDB) :

A scalable and secure repository for managing restaurant details, customer data, menus, orders, and reviews, ensuring consistency and high availability.

II KeyFeatures :

ER Diagram :



For Customers:

- **User Registration & Login:** Secure authentication system for creating and managing accounts.
- **Restaurant Search:** Search and filter restaurants by cuisine, location, ratings, and availability.
- **Order Placement:** Browse menus, add items to the cart, and place orders with real-time availability updates.
- **Order Management:** View, cancel, or modify orders conveniently.
- **Notifications:** Receive email or SMS alerts for order confirmations, status updates, and delivery notifications.

For Restaurants:

- **Profile Management:** Manage restaurant details, including menu items, pricing, and operational hours.
- **Order Management:** View and process incoming orders with real-time updates.
- **Revenue Tracking:** Monitor earnings and transaction history from completed orders.

For Administrators:

- **User Management:** Oversee and manage customer and restaurant accounts.
- **Order Monitoring:** Track and analyze orders, cancellations, and overall system performance.
- **Platform Analytics:** Gain insights into user behavior, popular dishes, restaurant performance, and revenue growth.

General Features:

- **Responsive Design:** Optimized for all devices, providing a seamless experience for users.
- **Secure Payments:** Integration with trusted payment gateways for a secure checkout process.
- **Real-Time Order Tracking:** Enable customers to track their order status and delivery progress.
- **Data Security:** End-to-end encryption and robust handling of sensitive user and transaction information.

2.Architecture :

Frontend: React.js Architecture

The frontend is developed using React.js to deliver a dynamic, responsive, and user-friendly interface. The architecture includes:

- **Component-Based Design:**
  - Reusable components like Header, Footer, Navbar, RestaurantList, MenuCard, Cart, OrderSummary, etc.
  - Organized component hierarchy for maintainability and scalability.
- **State Management:**
  - Uses React Context API or Redux for managing global states like user authentication, cart data, and order status.
- **Routing:**
  - React Router for single-page application (SPA) navigation with protected routes for customer and restaurant dashboards.
- **API Integration:**
  - Utilizes Axios or Fetch API to interact with backend services for fetching restaurant menus, placing orders, and managing user sessions.
- **Responsive Design:**
  - Styled with CSS frameworks like Tailwind CSS or Material-UI for optimal usability across devices.

Backend: Node.js and Express.js Architecture

The backend is powered by Node.js and Express.js to ensure a robust and secure API layer.

- **RESTful API Design:**
  - Endpoints like /api/restaurants, /api/orders, /api/users, /api/cart.
  - Follows REST principles for modular and scalable API structure.
- **Authentication & Authorization:**
  - Implements JWT (JSON Web Tokens) for secure token-based authentication.
  - Role-based access control (RBAC) for customers, restaurants, and admins.
- **Middleware:**
  - Custom middleware for request validation, error handling, and logging.
  - Third-party middleware like Multer for handling file uploads (e.g., menu images).
- **Routes:**
  - Maps API endpoints to corresponding controller functions.
- **Payment Integration:**
  - Secure integration with payment gateway APIs like Stripe or RazorPay for order transactions.

Database: MongoDB Schema & Interactions

The database is designed in MongoDB for flexibility and scalability with well-structured collections and relations.

- **Collections & Schemas:**

  1.Users Collection:
  - userId: Unique identifier
  - name, email, password, role (customer/restaurant/admin)
  - address, contact, profileImage, etc.

  2.Restaurants Collection:
  - restaurantId: Unique identifier
  - name, cuisine, menu (array of menu items), rating, location, availability

  3.Orders Collection:
  - orderId: Unique identifier
  - customerId, restaurantId (references Users and Restaurants)
  - items (array of ordered items), totalPrice, status (e.g., pending, preparing, delivered)
  - timestamp, deliveryAddress

  4.Admin Collection:
  - Manages platform analytics, user feedback, and restaurant performance data.

- **Database Interactions:**
  - **CRUD Operations:**
    - Mongoose for schema definitions and performing operations.
  - **Indexing:**
    - Indexes on restaurantId, userId, and timestamp for faster query performance.
  - **Data Validation:**
    - Ensures data integrity with Mongoose Validators (e.g., valid email formats, unique IDs).

3.Setup Instructions :

I. Prerequisites

Before you begin, ensure you have the following software installed:

1. **Node.js** (v16 or higher)
   - Download and install Node.js
2. **MongoDB** (latest version)
   - Install MongoDB locally or use a cloud-based service like MongoDB Atlas.
3. **Git**
   - Download and install Git.
4. **Package Manager**
   - **npm** (bundled with Node.js) or **yarn**.
5. **Code Editor**
   - Recommended: VS Code.

II. Installation Steps

i. Clone the Repository:

```
git clone <repository-url>
cd <project-folder>
```

ii. Install Dependencies:

- Navigate to the **frontend** folder and install dependencies:

```
cd frontend
npm install
```

- Navigate to the **backend** folder and install dependencies:
  cd backend

  npm install

iii. Set Up Environment Variables:

- Create a .env file in the **backend** folder and add the following variables:

PORT=5000

MONGO_URI=<Your MongoDB Connection String>
JWT_SECRET=<Your JWT Secret>

NODE_ENV=development

iv. Run the Application:

- Start the backend server:
  cd backend

  npm run dev

- Start the frontend server:
  cd frontend

  npm run dev

v. Access the Application:

- Open your browser and navigate to:

  - Frontend: http://localhost:5173
  - Backend (API): http://localhost:5454
  - Admin: http://localhost:5000

4.Folder Structure :

Frontend (Client):

The client-side of the application is organized as follows:

- **node_modules:** Contains npm packages required for the frontend.
- **public:** Stores static files like index.html and assets (e.g., images, fonts).
- **Components:** Houses reusable UI components such as:
  - Navbar.jsx
  - Footer.jsx
  - MenuCard.jsx

- **Pages:** Contains specific pages for the platform, such as:
  - Home.jsx
  - Restaurant.jsx
  - OrderSummary.jsx
- **Context:** Manages application state using context, e.g., AppContext.jsx.
- **Css:** Holds styling files, such as index.css.
- **.gitignore:** Lists files and folders to be ignored by Git (e.g., node_modules, .env).
- **package.json:** Lists the project's dependencies, scripts, and metadata.
- **package-lock.json:** Ensures consistent dependency installation across environments.

Backend (Server):

The server-side of the application is organized as follows:

- **node_modules:** Contains npm packages for backend functionalities.
- **index.js:** Entry point of the server, includes:
  - Configuration
  - Middleware setup
  - API routing

- **Routes:** Handles API endpoints like /api/restaurants and /api/orders.
- **Controllers:** Contains logic for handling requests and responses.
- **Models:** Mongoose schemas for MongoDB collections (e.g., User, Restaurant, Order).
- **Utils:** Helper functions for utilities like token generation or email notifications.
- **package-lock.json:** Ensures consistent backend dependency installation.

- **.gitignore:** Specifies files to ignore, such as node_modules and .env.
- **package.json:** Lists backend dependencies and scripts.

Admin Panel:

The admin interface is organized as follows:

- **node_modules:** Contains npm packages for admin functionality.
- **public:** Includes static files like index.html and assets.
- **Components:** Houses reusable UI components, such as:
  - Navbar.jsx
  - Table.jsx
  - Sidebar.jsx
- **Pages:** Contains specific admin-related pages, such as:
  - AdminDashboard.jsx
  - ManageRestaurants.jsx
  - ManageOrders.jsx
- **Context:** Manages state for the admin panel (e.g., AdminContext.jsx).
- **Css:** Contains styling files specific to the admin interface (e.g., index.css).
- **.gitignore:** Specifies files to ignore, such as node_modules and .env.
- **package.json:** Lists dependencies and scripts for the admin interface.
- **package-lock.json:** Ensures consistent dependency versions across installations.

5.Running the Application :

- To run the application locally, you'll need to start both the frontend and backend servers. Follow the commands below to launch each part of the application:

Frontend :

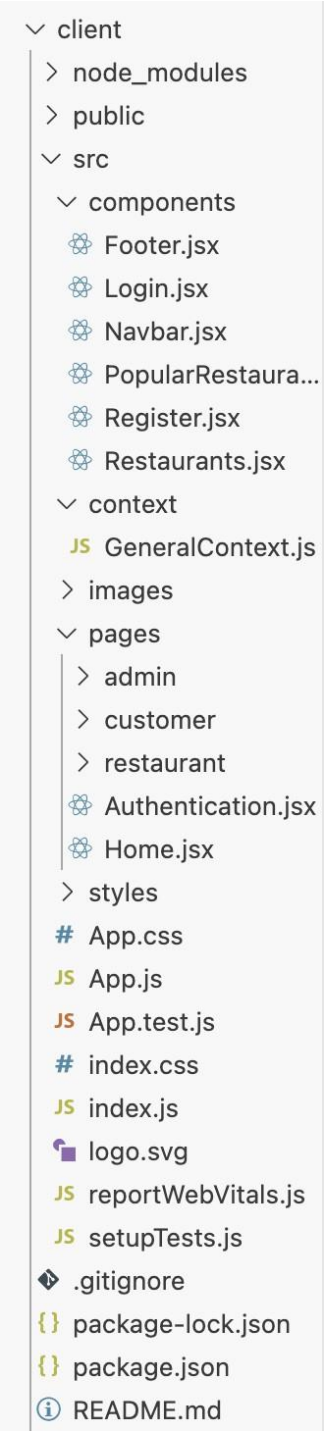
1. Navigate to the client directory:

   cd frontend

2. Start the React development server:

   npm start

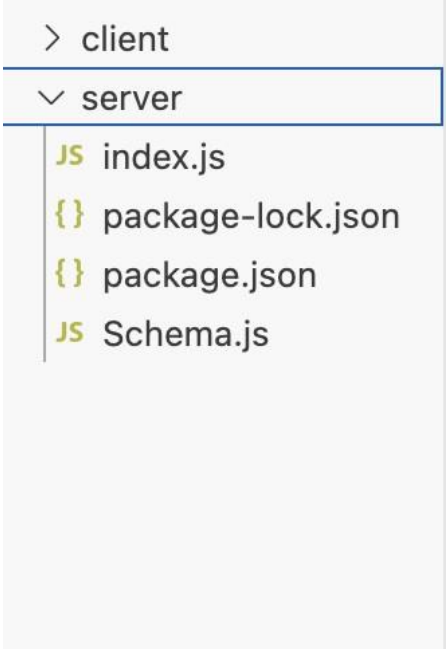
This will run the frontend application on : http://localhost:5173



Backend (Node.js) :

1. Navigate to the server directory:
   cd backend
2. Start the Node.js server:
   npm run dev

This will run the backend server on : http://localhost:5454



ADMIN :

1. Navigate to the server directory:
   cd admin
2. Start the Node.js server:
   npm run dev

This will run the backend server on: http://localhost:5000

6.API Documentation :

General Routes

Root Endpoint

- Endpoint: /
- Method: GET
- Description: Verifies the API is operational.

Example Response:

```
{
  "message": "API Working"
}
```

Authentication Routes

Base Path: /auth

1. **POST /auth/login**
   - Description: Logs in a user and provides a token.
   - Request Body:

```
{
  "email": "user@example.com",
  "password": "securepassword"
}
```

   - **Example Response:**

```
{
  "token": "jwt-token",
  "user": {
    "id": "user123",
    "name": "John Doe",
    "email": "user@example.com"
  }
}
```

2. **POST /auth/register**
   - Description: Registers a new user.
   - Request Body:

```
{
  "name": "John Doe",
  "email": "user@example.com",
```

```
  "password": "securepassword"
}
```

Authentication Routes

Base Path: /auth

3. **POST /auth/login**
    - Description: Logs in a user and provides a token.
    - Request Body:

```
{
  "email": "user@example.com",
  "password": "securepassword"
}
```

    - **Example Response:**

```
{
  "token": "jwt-token",
  "user": {
    "id": "user123",
    "name": "John Doe",
    "email": "user@example.com"
  }
}
```

4. **POST /auth/register**
    - Description: Registers a new user.
    - Request Body:

```
{
  "name": "John Doe",
  "email": "user@example.com",
  "password": "securepassword"
}
```

Example Response:


```
{
  "message": "User registered successfully",
  "user": {
    "id": "user123",
    "name": "John Doe",
    "email": "user@example.com"
  }
}
```


User Routes

Base Path: /api/user

1. **GET /api/user**
   - Description: Retrieves all users.
   - Example Response:


```
{
  "users": [
    {
      "id": "user123",
      "name": "John Doe",
      "email": "user@example.com"
    }
  ]
}
```


2. **GET /api/user/:id**
   - Description: Retrieves user details by ID.
   - Request Parameters: id (string) - User ID.
   - Example Response:

{
"id": "user123",
"name": "John Doe",
"email": "user@example.com"
}

3. **PUT /api/user/:id**
    - Description: Updates user details.
    - Request Body:

{
"name": "Updated Name",
"email": "updated@example.com"
}

Example Response:

{
"message": "User updated successfully",
"user": {
"id": "user123",
"name": "Updated Name",
"email": "updated@example.com"
}
}

4. **DELETE /api/user/:id**
    - Description: Deletes a user by ID.
    - Request Parameters: id (string) - User ID.

{
"message": "User deleted successfully"
}

Restaurant Routes

Base Path: /api/restaurants

1. **GET /api/restaurants**
    - Description: Retrieves a list of all restaurants.
    - Example Response:


```
{
  "restaurants": [
    {
      "id": "rest123",
      "name": "Italian Bistro",
      "location": "New York",
      "cuisine": "Italian"
    }
  ]
}
```


2. **GET /api/restaurants/:id**
    - Description: Retrieves details of a specific restaurant.
    - Request Parameters: id (string) - Restaurant ID.


```
{
  "id": "rest123",
  "name": "Italian Bistro",
  "location": "New York",
  "cuisine": "Italian"
}
```

Admin Restaurant Routes

Base Path: /api/admin/restaurants

1. **GET /api/admin/restaurants**
    - Description: Retrieves all restaurants for admin purposes.
    - Example Response:


```
{
  "restaurants": [
    {
      "id": "rest123",
      "name": "Italian Bistro",
      "location": "New York",
      "cuisine": "Italian"
    }
  ]
}
```


Order Routes

Base Path: /api/order

1. **GET /api/order**
    - Description: Retrieves all orders.
    - Example Response:


```
{
  "orders": [
    {
      "id": "order123",
      "userId": "user123",
      "restaurantId": "rest123",
      "items": [
        {
```

"menuItemId": "item123",
"quantity": 2
}
]
}
]
}

2. **POST /api/order**
   - Description: Places a new order.
   - Request Body:

{
"userId": "user123",
"restaurantId": "rest123",
"items": [
{
"menuItemId": "item123",
"quantity": 2
}
]
}

Example Response:

{
"message": "Order placed successfully",
"order": {
"id": "order123",
"userId": "user123",

"restaurantId": "rest123"
}
}

Cart Routes

Base Path: /api/cart

1. **GET /api/cart/:userId**
    - Description: Retrieves the cart of a specific user.
    - Request Parameters: userId (string) - User ID.

{
"userId": "user123",
"items": [
{
"menuItemId": "item123",
"quantity": 2
}
]
}

2. **POST /api/cart**
    - Description: Adds an item to the cart.

{
"userId": "user123",
"menuItemId": "item123",
"quantity": 1
}

Example Response:


```
{
"message": "Item added to cart"
}
```


Category Routes

Base Path: /api/category

1. **GET /api/category**
    - Description: Retrieves all food categories.
    - Example Response:


```
{
  "categories": [
    {
      "id": "cat123",
      "name": "Desserts"
    }
  ]
}
```


Error Responses

- **Standard Error Format:**


```
{
  "error": "Resource not found",
  "status": 404
}
```

7.Authentication :

JWT Authentication: Tokens are generated upon login and stored in localStorage.

Role-based Access Control:

Users: Restricted to product browsing and ordering.

Admins: Full access to user, product, and order management.

8. User Interface :

Key pages include:

- Home Page: Displays available restaurants and products.
- Cart Page: Showcases items added by the user with options to edit/remove.
- Checkout Page: Captures address and payment details.
- Order History: Displays previous orders.

9. Testing :

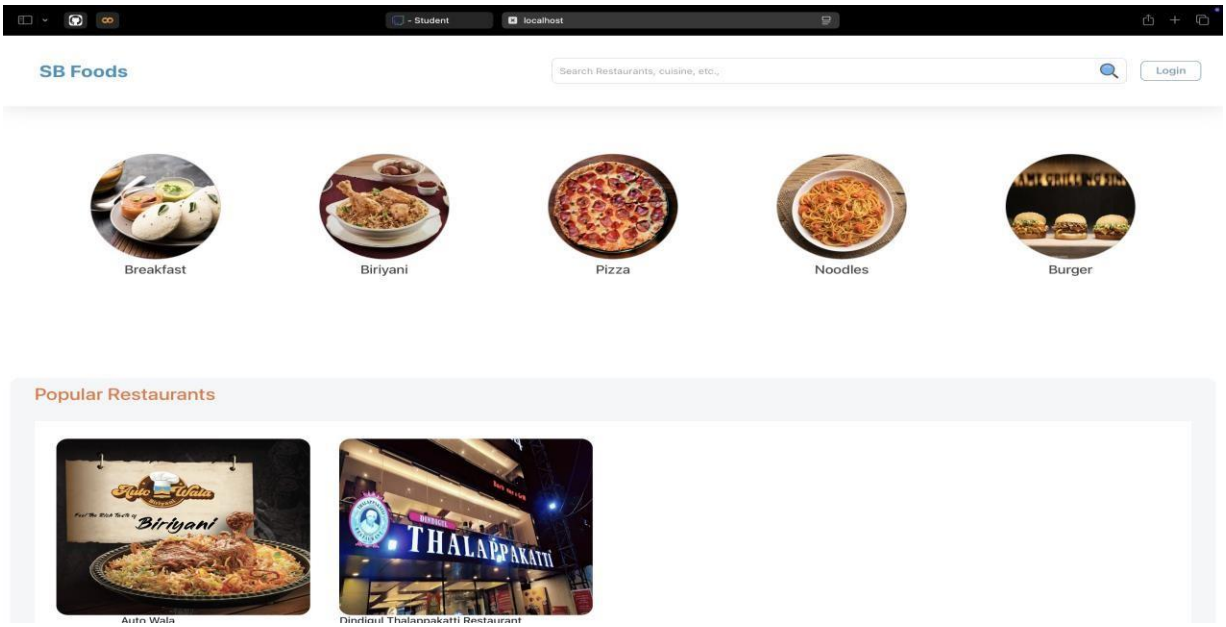
Frontend: Jest and React Testing Library for component testing.

Backend: Mocha and Chai for API endpoint testing.

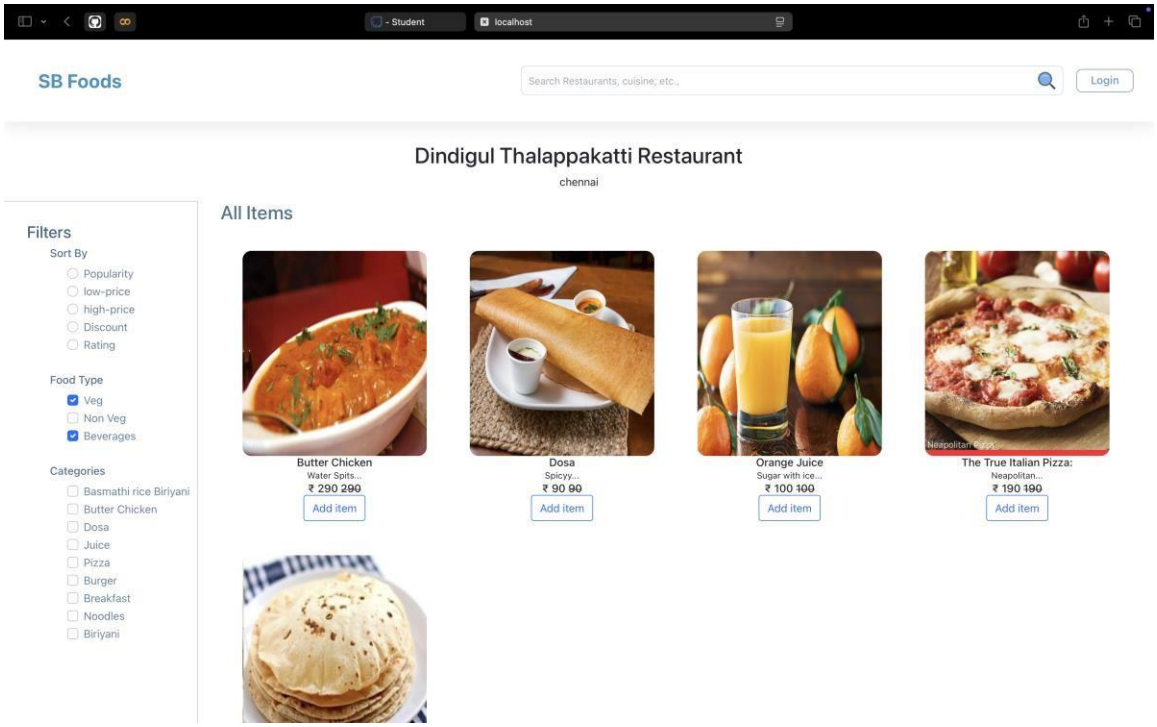
10. Screenshots or Demo :

Screenshots:

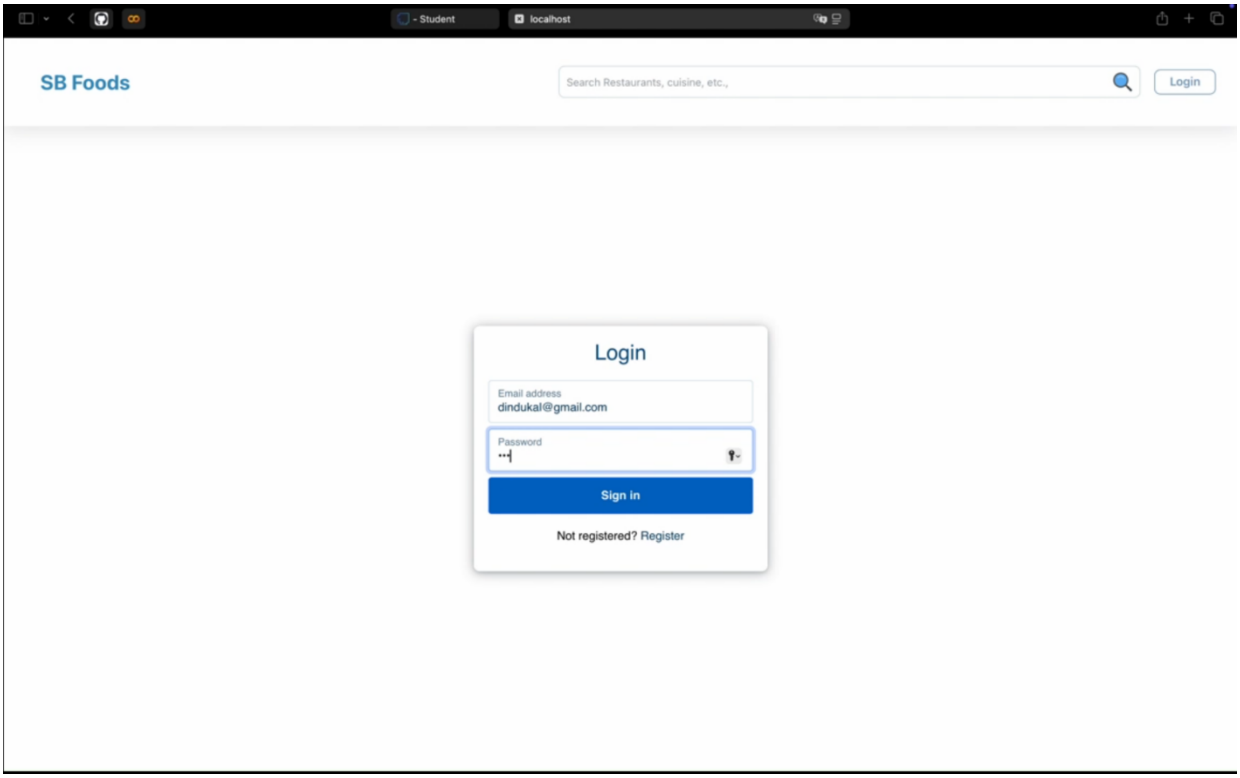
Home Page



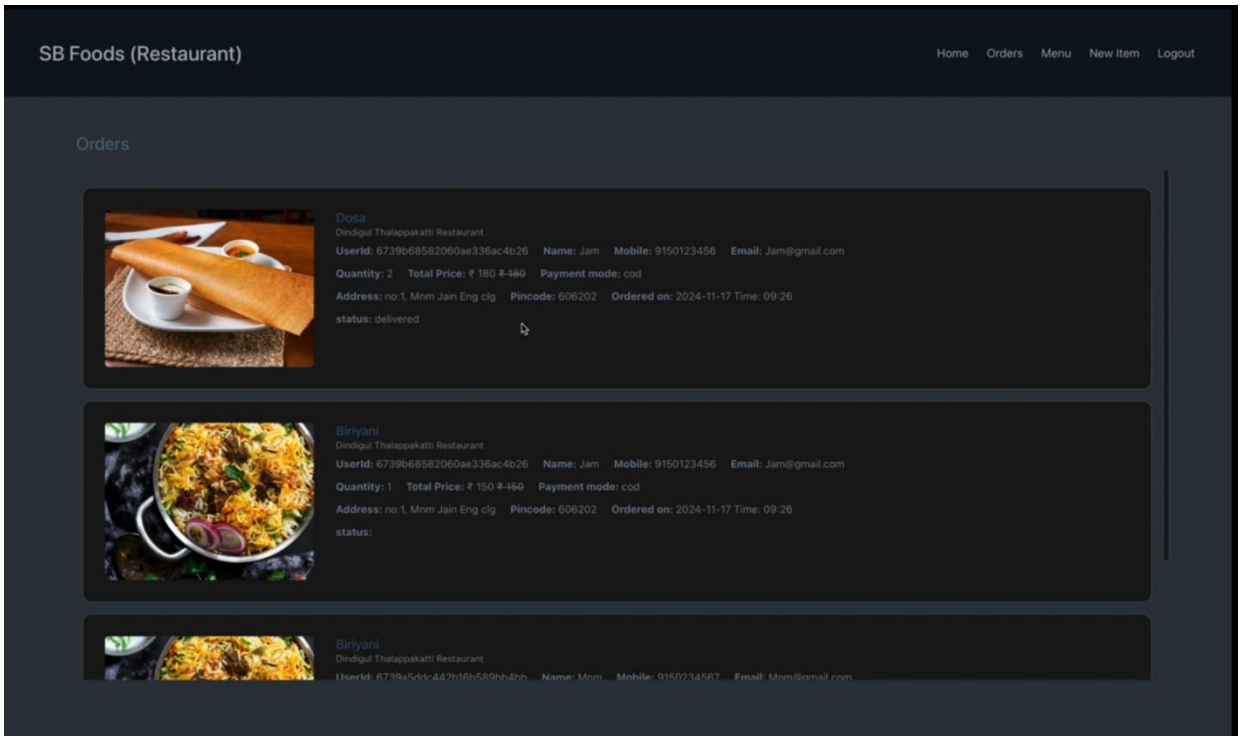
Cart Page



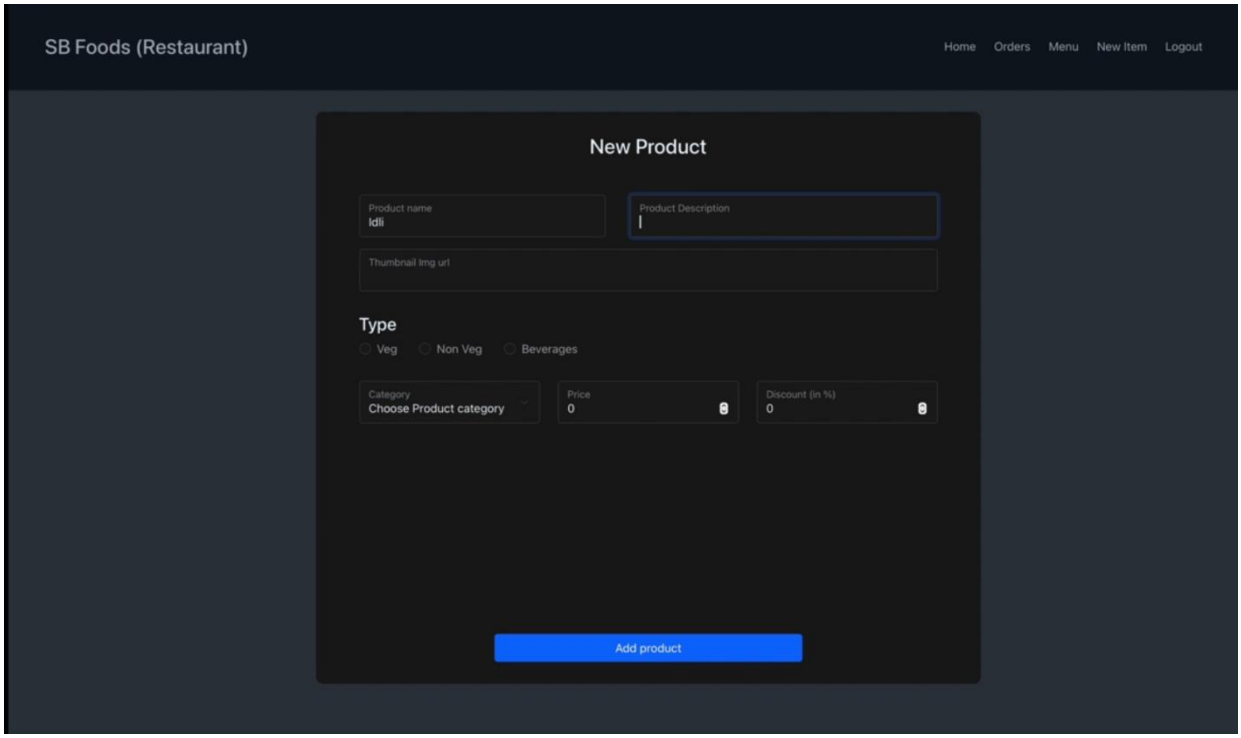
Login Page



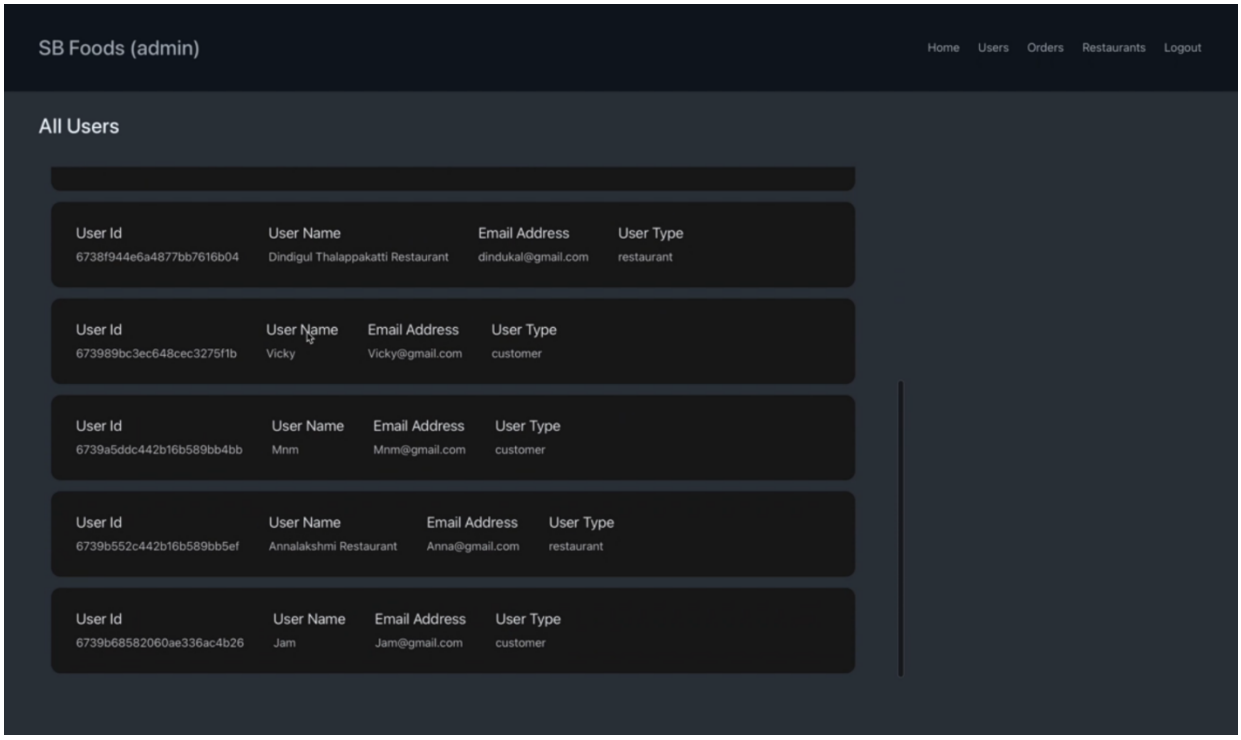
Order Page



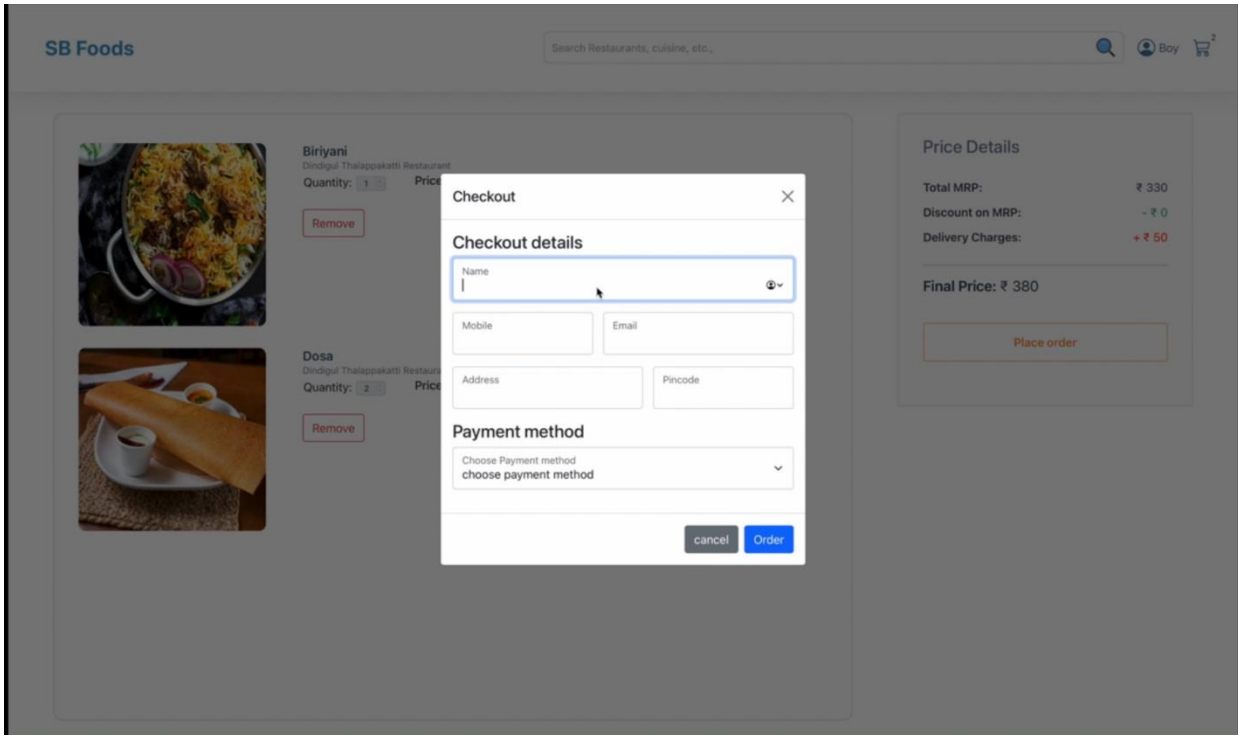
New Product Page



All User Page



Checkout Page



Checkout Flow

Demo:

https://drive.google.com/file/d/1cufDXGSTA-MYJApdfTWGRXjKYSphXm0F/view?usp=sharing

11. Known Issues :

- Occasionally slow response times when querying large datasets.
- Lack of email notifications for order confirmation.

12. Future Enchancements :

- Mobile Application : Develop a companion app for ios and Android.
- Recommendation System : Suggest popular products based on user history.
- Payment Integration : Add PayPal and Apple Pay.
- Real-Time Order Tracking : Enable GPS-based delivery track