Full Stack Development with MERN

Team Members

Arssam Basha M – arssambasha82@gmail.com - au11012110414 (Lead)

Abdul Kareem M –abdulkareem920092@gmail.com - au110121104002

Afzal Hameed Faiz V Y – afzalfaiz0110@gmail.com - au110121104006

Mohamed Imthiyaz K – mimthiyaz100@gmail.com - au110121104050

Project Title: ***SB Foods - Food Ordering App***

Project Overview:

The purpose of this project is to develop a full-stack food ordering application that streamlines the process of discovering and ordering meals. By integrating React.js for the frontend, Node.js for the backend, and MongoDB for database management, the application aims to provide a seamless and intuitive user experience. Users can browse a diverse menu with detailed descriptions and reviews, add their desired items to a cart, and place orders effortlessly. Additionally, the platform offers real-time order tracking, ensuring users stay informed about their deliveries. With features such as user profile management and scalable infrastructure, the application is designed to cater to a wide range of user needs while maintaining reliability and performance. This project focuses on delivering a functional and accessible solution that bridges the gap between users and their favourite eateries.

Features:

Comprehensive Product Catalog: SB Foods boasts an extensive catalog of food items from various restaurants, offering a diverse range of items and options for shoppers. You can effortlessly explore and discover various products, complete with detailed descriptions, customer reviews, pricing, and available discounts, to find the perfect food for your hunger.

Order Details Page: Upon clicking the "Shop Now" button, you will be directed to an order details page. Here, you can provide relevant information such as your shipping address, preferred payment method, and any specific product requirements.

Secure and Efficient Checkout Process: SB Foods guarantees a secure and efficient checkout process. Your personal information will be handled with the utmost security, and we strive to make the purchasing process as swift and trouble-free as possible.

Order Confirmation and Details: After successfully placing an order, you will receive a confirmation notification. Subsequently, you will be directed to an order details page, where you can review all pertinent information about your order, including shipping details, payment method, and any specific product requests you specified.

Architecture:

Frontend:

The frontend of the application was built using React.js, employing a modular and component-driven architecture to ensure scalability, maintainability, and efficiency throughout the development process. Each feature of the application, including navigation bars, menu displays, user authentication forms, and order management sections, was encapsulated as a reusable and independent component. This design provided a clear separation of concerns, which not only improved code readability but also simplified debugging and testing efforts as the application evolved.

To deliver an enhanced user experience, the interface was crafted with a fully responsive design, ensuring seamless compatibility across a variety of devices and screen sizes, such as desktops, tablets, and smartphones. This responsiveness was achieved through the use of CSS frameworks and custom styling techniques to create a visually consistent and adaptive design that caters to a diverse audience.

Dynamic routing was managed effectively using React Router, which enabled smooth page transitions and an intuitive navigation flow. This facilitated the creation of a single-page application (SPA) experience, ensuring users could interact with the application quickly and without unnecessary page reloads. Although advanced state management tools like Redux or Context API were considered, the initial development focused on utilizing React's built-in state management capabilities to maintain simplicity while meeting the project requirements.

Additionally, the frontend's modular structure was designed to accommodate future enhancements, such as advanced features and integrations. This forward-thinking approach, coupled with a strong emphasis on user-centric design, aimed to create a visually appealing and highly functional interface that delivers a seamless experience while laying the foundation for future scalability and extensibility.

Backend:

The backend of the application was developed using Node.js, with Express.js providing a robust framework for building a scalable and efficient server. A modular architecture was employed, organizing the codebase into separate layers for routing, controllers, middleware, and database operations. This separation of concerns enhanced the maintainability of the application, allowing each part to function independently and be updated without disrupting the overall structure.

Express.js was used to create RESTful API endpoints for handling core functionalities like user authentication, menu retrieval, order processing, and user profile management. These routes were structured to ensure clarity and efficiency, with each endpoint mapped to its corresponding controller logic. Versioning was considered to future-proof the API, making it adaptable for potential updates or expansions.

MongoDB was chosen as the database for its scalability and flexibility, with Mongoose acting as an ORM to streamline database interactions. Mongoose schemas were defined for collections such as users, orders, and menu items, ensuring structured and consistent data storage. Query optimization techniques and indexing were applied to enhance performance, especially for frequently accessed data like menu items and order histories.

The backend was equipped with robust error handling mechanisms to manage both application-level and server-level errors gracefully. Standardized error responses improved the clarity of API communication. Security was prioritized with practices such as data sanitization, rate limiting, and secure headers, mitigating risks like SQL injection and cross-site scripting (XSS).

The backend was designed to handle concurrent requests efficiently by leveraging Node.js's asynchronous capabilities. Load testing was conducted to evaluate the server's performance under varying traffic conditions, and optimizations were made to minimize response times. The architecture was also designed to be scalable, allowing the addition of new features or handling increased traffic with minimal structural changes.

Database:

The database architecture for the application was built using MongoDB, a NoSQL database known for its scalability, flexibility, and ease of use. MongoDB's schema-less nature allowed us to define a dynamic data model that could evolve alongside the application's growth and changing requirements. The database was structured to store and manage core entities such as users, menu items, orders, and payment details, with each collection representing a different aspect of the application.

Data Modelling with Mongoose:

To provide a structured approach to data storage and retrieval, Mongoose was used to define schemas for each collection. The user schema included fields like name, email, password, and order history, while the menu schema captured details like dish name, description, price, and category. The order schema was designed to store transaction-related information, including the user ID, ordered items, payment status, and timestamps. Using Mongoose allowed for easy validation, population of related data, and interaction with the MongoDB database via a clean and intuitive API.

Database Relationships and Population:

Although MongoDB is a document-based database, relationships between different entities were modelled through references and population. For example, an order could reference the user who placed it and the menu items they selected. Mongoose's populate() method enabled us to efficiently retrieve and merge data from related collections in a single query, reducing the number of requests needed to fetch data and improving application performance.

Query Optimization and Indexing:

Performance was a key consideration when designing the database architecture. To optimize query performance, indexes were created on frequently searched fields such as user email and menu item categories. Indexing helped speed up lookups and reduce the time spent querying large datasets. We also implemented pagination for large collections, ensuring that the application remained responsive even when dealing with extensive data sets like order histories or menu items.

Security and Data Integrity

To ensure data security and integrity, encryption techniques were used to store sensitive user data, such as passwords. Passwords were hashed using bcrypt before being stored in the database, preventing the exposure of plaintext passwords in case of a data breach. Additionally, access control mechanisms were integrated to ensure that users could only modify their own data, such as their profile information or order details.

Scalability and Future-proofing

Scalability was an important consideration in the design of the database. The flexible schema of MongoDB allowed for easy updates and adjustments to the data model as the application expanded. Sharding, a technique that distributes data across multiple servers, was also considered for the future in case the application needed to scale horizontally to handle an increased volume of data or traffic.

ER-Diagram:

The SB Foods ER-diagram represents the entities and relationships involved in a food ordering e-commerce system. It illustrates how users, restaurants, products, carts, and orders are interconnected.

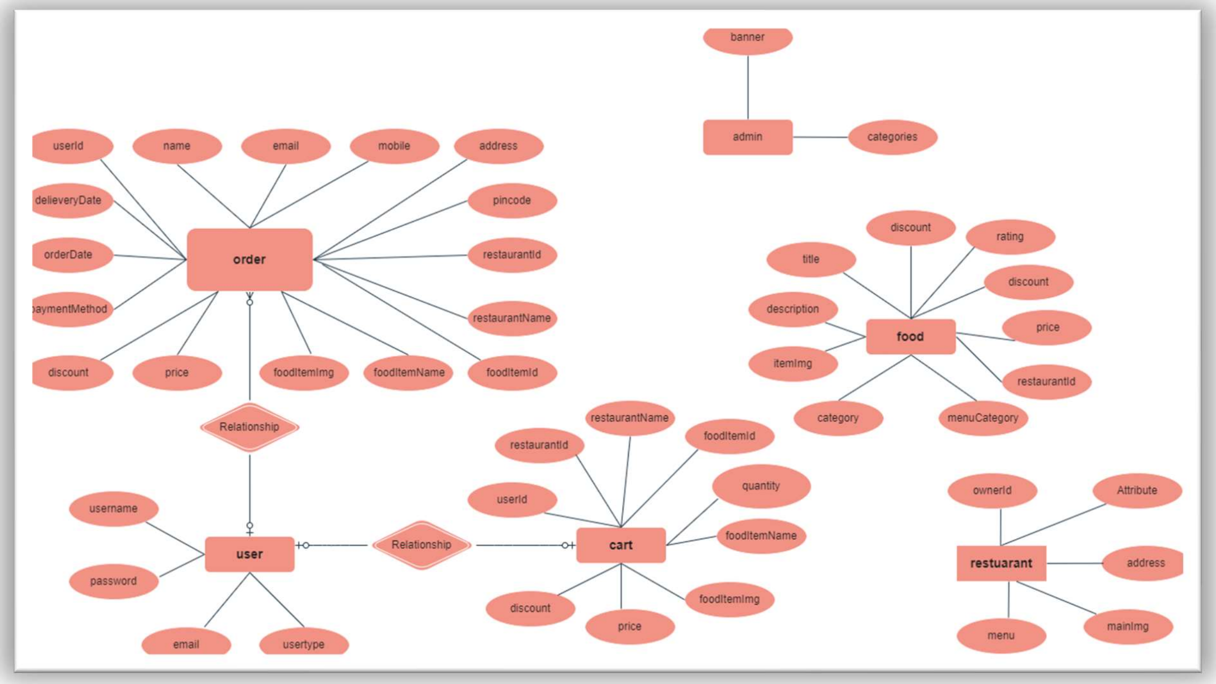


Fig. 1 – Attributes of each entity and their connections

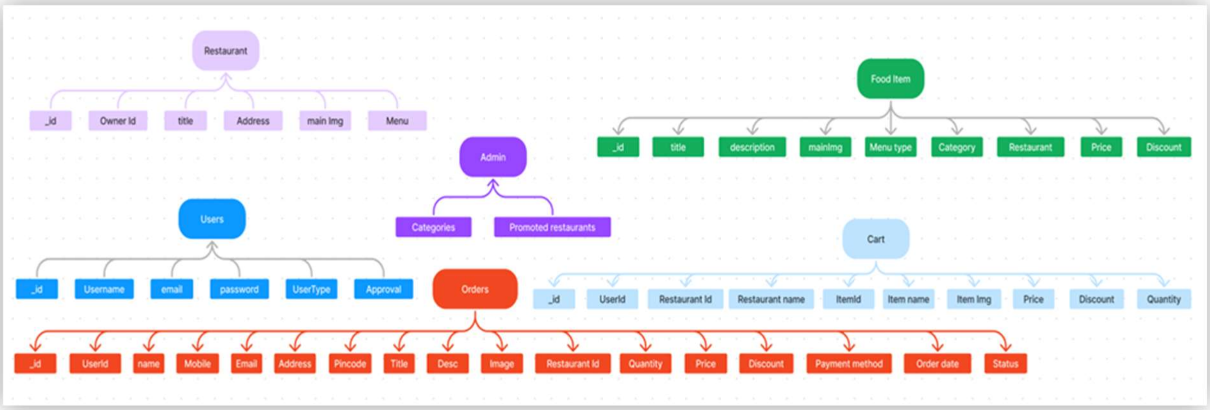


Fig. 2 – Logical Database Schema

SB Foods also provides a robust restaurant dashboard, offering restaurants an array of functionalities to efficiently manage their products and sales. With the restaurant dashboard, restaurants can add and oversee multiple product listings, view order history, monitor customer activity, and access order details for all purchases.

Entities and their Relationships:

User: Represents the individuals or entities who are registered in the platform.

Restaurant: This represents the collection of details of each restaurant in the platform.
Admin: Represents a collection with important details such as promoted restaurants and Categories.

Products: Represents a collection of all the food items available in the platform.

Cart: This collection stores all the products that are added to the cart by users. Here, the elements in the cart are differentiated by the user Id.

Orders: This collection stores all the orders that are made by the users in the platform.

SB Foods is designed to elevate your online food ordering experience by providing a seamless and user-friendly way to discover your desired foods. With our efficient checkout process, comprehensive product catalog, and robust restaurant dashboard, we ensure a convenient and enjoyable online shopping experience for both shoppers and restaurants alike.

Setup Instructions:

Pre-Requisite:

To develop a full-stack food ordering app using React JS, Node.js, and MongoDB, there are several prerequisites should be considered. Here are the key prerequisites for developing such an application:

Node.js and npm: Install Node.js, which includes npm (Node Package Manager), on the development machine. Node.js is required to run JavaScript on the server side.

• Download: https://nodejs.org/en/download/

• Installation instructions: https://nodejs.org/en/download/package-manager/

MongoDB: Set up a MongoDB database to store hotel and booking information. Install MongoDB locally or use a cloud-based MongoDB service.

• Download: https://www.mongodb.com/try/download/community

• Installation instructions: https://docs.mongodb.com/manual/installation/

Express.js: Express.js is a web application framework for Node.js. Install Express.js to handle server-side routing, middleware, and API development.

• Installation: Open your command prompt or terminal and run the following command:

```
npm install express
```

React.js: React.js is a popular JavaScript library for building user interfaces. It enables developers to create interactive and reusable UI components, making it easier to build dynamic and responsive web applications. To install React.js, a JavaScript library for building user interfaces, follow the installation guide: https://reactjs.org/docs/create-a-new-react-app.html

HTML, CSS, and JavaScript: Basic knowledge of HTML for creating the structure of your app, CSS for styling, and JavaScript for client-side interactivity is essential.

Database Connectivity: Use a MongoDB driver or an Object-Document Mapping (ODM) library like Mongoose to connect your Node.js server with the MongoDB database and perform CRUD (Create, Read, Update, Delete) operations.

Front-end Framework: Utilize Angular to build the user-facing part of the application, including product listings, booking forms, and user interfaces for the admin dashboard.

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

• Git: Download and installation instructions can be found at: https://git scm.com/downloads

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

• Visual Studio Code: Download from https://code.visualstudio.com/download

• Sublime Text: Download from https://www.sublimetext.com/download

• WebStorm: Download from https://www.jetbrains.com/webstorm/download

To Connect the Database with Node JS go through the below provided link:

Link: https://www.section.io/engineering-education/nodejs- mongoosejs-mongodb/

Installation:

Clone the repository:

• Open your terminal or command prompt.

• Navigate to the directory where you want to store the project.

• Execute the following command to clone the repository:

Git clone:

https://github.com/basha577/NM_MERN_Food_Ordering_App.git

Dependencies:

• Navigate into the cloned repository directory:

```
cd NM_MERN_Food_Ordering_App
```

• Install the required dependencies by running the following command:

```
npm install
```

Start the Development Server:

• To start the development server, execute the following command:

npm run dev or npm run start

• The project will be accessible at http://localhost:3000 by default. You can change the port configuration in the file if needed.

Access the App:

• Open your web browser and navigate to http://localhost:3000.

• You should see the flight booking app's homepage, indicating that the installation and setup were successful.

You have successfully installed the project.

Application Flow:

1. User Flow:

• Users start by registering for an account.

• After registration, they can log in with their credentials.

• Once logged in, they can check for the available products in the platform. • Users can add the products they wish to their carts and order.

• They can then proceed by entering address and payment details. • After ordering, they can check them in the profile section.

2. Restaurant Flow:

• Restaurants start by authenticating with their credentials.

• They need to get approval from the admin to start listing the products. • They can add/edit the food items.

3. Admin Flow:

• Admins start by logging in with their credentials.

• Once logged in, they are directed to the Admin Dashboard.

• Admins can access the users list, products, orders, etc.

Folder Structure:

The folder structure of a full-stack web application plays a critical role in organizing the codebase in a modular and maintainable way. It ensures that the different aspects of the application—frontend, backend, and shared resources—are clearly separated, making the development process more efficient. Below, we will delve into the rationale behind the folder structure for both the client (frontend) and server (backend) of a full-stack React, Node.js, and MongoDB application.

This structure assumes a React app and follows a modular approach.

Client:

The **frontend** of the application is built using **React**, a popular JavaScript library for building user interfaces. The client folder structure is designed to separate concerns between different components, pages, and services. This modular approach helps in better organization, easier testing, and efficient code maintenance.

Client Folder Structure:

public/
The **public** directory contains static files that do not change during the execution of the app. This includes the index.html file, where the React app is injected, and other assets such as the favicon. The **index.html** file serves as the single entry point for the React application.

src/
The **src** folder contains all the JavaScript and CSS files necessary to build the React application. Within **src**, further segmentation is made to ensure clarity and maintainability:

- **components/**: Contains reusable UI components like buttons, form fields, navigation bars, and other widgets that are used across multiple pages of the app. These components are typically small, isolated pieces of functionality designed to be reused.
- **pages/**: The **pages** folder contains the React components that represent different views or pages in the application. For example, you might have a HomePage.js for the home screen, MenuPage.js for the food menu, or LoginPage.js for user authentication.

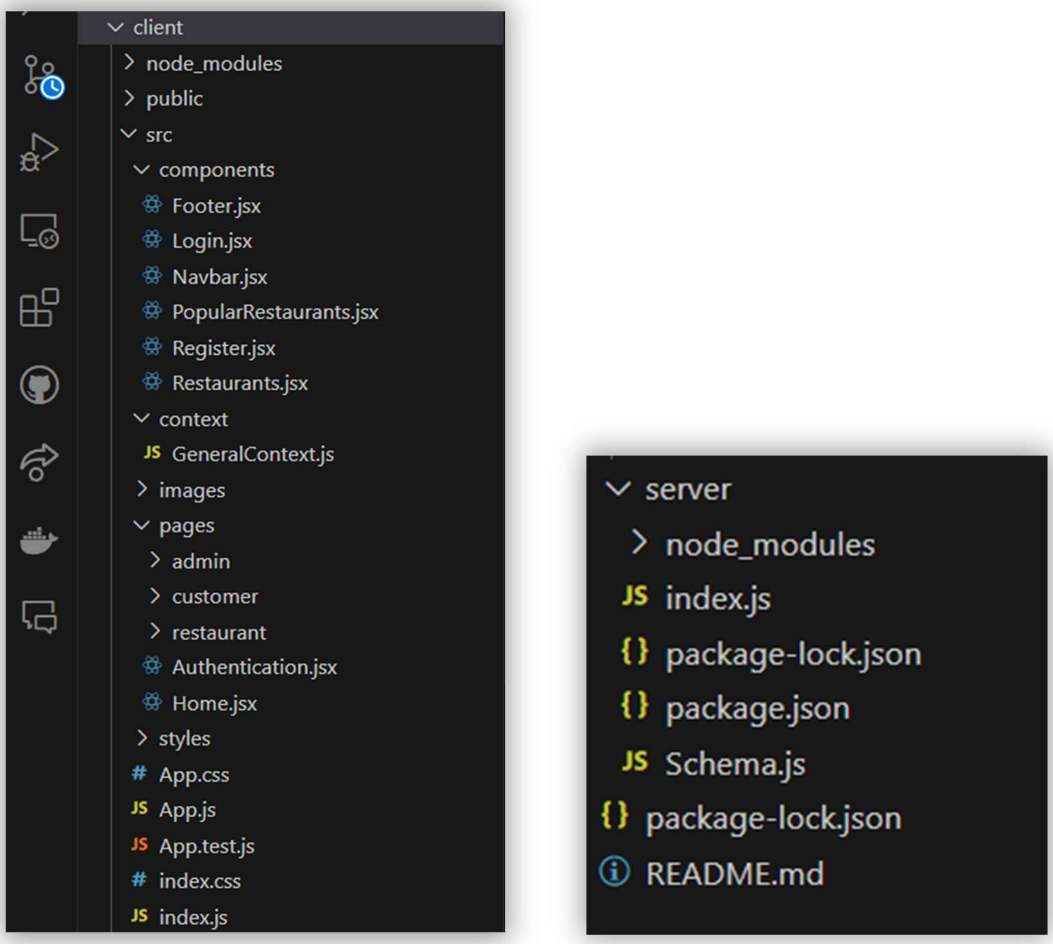
- **context/**: This folder is used if you're utilizing the **Context API** to manage global state across your application. For instance, it might store authentication data or the shopping cart state, making it accessible throughout the app.
- **services/**: Contains functions that interact with external resources, such as API calls to the backend. The **api.js** file, for example, could include methods like fetchMenu() or placeOrder() to make requests to the server-side API.

App.js

The **App.js** file is the root component where routing and top-level components are defined. It is where you establish routing between different pages of the application using **React Router**.

index.js

The **index.js** file is the entry point for the React application. It renders the **App.js** component into the DOM and initiates the application.



```
client
  node_modules
  public
  src
    components
      Footer.jsx
      Login.jsx
      Navbar.jsx
      PopularRestaurants.jsx
      Register.jsx
      Restaurants.jsx
    context
      GeneralContext.js
    images
    pages
      admin
      customer
      restaurant
      Authentication.jsx
      Home.jsx
    styles
    App.css
    App.js
    App.test.js
    index.css
    index.js
```

```
server
  node_modules
  index.js
  package-lock.json
  package.json
  Schema.js
package-lock.json
README.md
```

Fig. 4 – client and server folder structure

Server:

The **backend** of the application is built using **Node.js** with the **Express.js** framework. The backend is responsible for handling API requests, processing business logic, and interacting with the **MongoDB** database. The server folder structure follows common practices for maintaining clean, modular, and scalable server-side applications.

Server Folder Structure:

- **controllers/**
  Controllers handle the business logic of your application. They are responsible for processing incoming requests and sending appropriate responses. For example, a userController.js file might handle user authentication, registration, and profile management, while an orderController.js would handle placing and tracking food orders. Keeping the controller logic separated ensures that your business logic is well-organized and easier to test.

- **models/**
  The **models** directory contains the **Mongoose** schemas and models used to interact with the MongoDB database. These models define the structure of the data stored in the database and provide methods to query and update that data. For example, a userModel.js file might define the schema for user data (name, email, password) and include validation methods.

- **routes/**
  The **routes** directory is where you define the API routes for different endpoints. Routes are responsible for mapping incoming HTTP requests (GET, POST, PUT, DELETE) to the appropriate controller functions. For instance, userRoutes.js would define the routes for user-related actions like login, registration, and fetching user data.

- **middleware/**
  The **middleware** directory contains functions that run before a request reaches the final route handler. Middleware can perform tasks such as validating request data, checking if a user is authenticated, or logging the request. For example, an **authMiddleware.js** file could check if a user has a valid JWT token before allowing access to a protected route.

- **index.js**
  The **index.js** file is the entry point to the Node.js application. It sets up the **Express.js** server, connects to the database, and configures middleware and routes. It is the core file that initializes and runs the backend server.

Running the Application:

To run the full application locally, you will need to start both the frontend (React) and backend (Node.js) servers. Follow the steps below to get the application up and running:

Frontend (React Application):

First, navigate to the client directory, which contains the React application:

```
cd client
```

If you haven't already installed the necessary dependencies, run the following command to install them:

```
npm install
```

After installing the dependencies, start the React development server by running:

```
npm start
```

This will launch the frontend application, and it should be accessible in your web browser at http://localhost:3000. Any changes you make to the frontend code will automatically reload in the browser.

Backend (Node.js Application):

Next, navigate to the server directory, which contains the Node.js backend:

```
cd server
```

Similarly, if you haven't installed the backend dependencies, run the following command to install them:

```
npm install
```

Once the dependencies are installed, you can start the backend server by running:

```
node index.js
```

The backend server will now be running on http://localhost:5000, or another port if specified in your configuration. This will allow the frontend to make API requests to the backend.

API Documentation:

1. POST /register

- **Description**: Registers a new user.
- **Request Body**: User information (e.g., username, password).
- **Response**: Confirmation of registration or error message.

2. POST /login

- **Description**: Logs in a user.

- **Request Body**: User credentials (e.g., username, password).
- **Response**: Authenticated user data or error message.

3. POST /update-promote-list

- **Description**: Updates the list of promoted restaurants.
- **Request Body**: Restaurant details to promote.
- **Response**: Confirmation of the update.

4. POST /approve-user

- **Description**: Approves a user.
- **Request Body**: User ID to approve.
- **Response**: Confirmation of approval.

5. POST /reject-user

- **Description**: Rejects a user.
- **Request Body**: User ID to reject.
- **Response**: Confirmation of rejection.

6. GET /fetch-user-details/

- **Description**: Fetch details of a specific user by their ID.
- **Request Params**: id (User ID).
- **Response**: User details in JSON format.

7. GET /fetch-users

- **Description**: Fetches a list of all users.
- **Response**: Array of user objects.

8. GET /fetch-restaurants

- **Description**: Fetches a list of all restaurants.
- **Response**: Array of restaurant objects.

9. GET /fetch-product-details/

- **Description**: Fetches details of a specific product by its ID.
- **Request Params**: id (Product ID).
- **Response**: Product details in JSON format.

10. GET /fetch-products

- **Description**: Fetches a list of all products.
- **Response**: Array of product objects.

11. GET /fetch-orders

- **Description**: Fetches a list of all orders.
- **Response**: Array of order objects.

12. GET /fetch-items

- **Description**: Fetches a list of all food items.
- **Response**: Array of food item objects.

13. GET /fetch-categories

- **Description**: Fetches a list of food categories.
- **Response**: Array of category objects.

14. GET /fetch-promoted-list

- **Description**: Fetches a list of promoted restaurants.
- **Response**: Array of promoted restaurant objects.

15. GET /fetch-restaurant-details/

- **Description**: Fetches details of a specific restaurant by its ID.
- **Request Params**: id (Restaurant ID).
- **Response**: Restaurant details in JSON format.

16. GET /fetch-restaurant/

- **Description**: Fetches basic details of a restaurant by its ID.
- **Request Params**: id (Restaurant ID).
- **Response**: Restaurant details in JSON format.

17. GET /fetch-item-details/

- **Description**: Fetches details of a specific food item by its ID.
- **Request Params**: id (Item ID).
- **Response**: Food item details in JSON format.

18. POST /add-new-product

- **Description**: Adds a new product to the menu.
- **Request Body**: Product details (e.g., name, description, price).

- **Response**: Confirmation of product addition.

19. PUT /update-product/

- **Description**: Updates details of a product.
- **Request Params**: id (Product ID).
- **Request Body**: Updated product details.
- **Response**: Confirmation of the update.

20. PUT /cancel-order

- **Description**: Cancels an order.
- **Request Body**: Order ID to cancel.
- **Response**: Confirmation of order cancellation.

21. PUT /update-order-status

- **Description**: Updates the status of an order.
- **Request Body**: Order ID and new status.
- **Response**: Confirmation of status update.

22. GET /fetch-cart

- **Description**: Fetches items in the user's shopping cart.
- **Response**: Array of cart items.

23. POST /add-to-cart

- **Description**: Adds an item to the cart.
- **Request Body**: Item details to add to the cart.
- **Response**: Confirmation of item addition to the cart.

24. PUT /remove-item

- **Description**: Removes an item from the cart.
- **Request Body**: Item ID to remove.
- **Response**: Confirmation of item removal.

25. POST /place-cart-order

- **Description**: Places an order from the cart.
- **Request Body**: Cart items and user information (e.g., delivery address).
- **Response**: Confirmation of the order.

Authentication and Authorization:

Authentication in this project is responsible for verifying the identity of users by checking their credentials, such as username and password, when they attempt to log in. If the credentials match the information stored in the database, the user is authenticated. Once authenticated, the system uses authorization to determine what actions the user is allowed to perform based on their role, such as "customer," "admin," or "restaurant." This role-based access ensures that each user only has access to the parts of the application relevant to their role. For example, a customer can browse and order food, an admin can manage users and restaurants, and a restaurant owner can update their menu and process orders. The combination of authentication and authorization ensures that the application is secure, and only authorized users can access certain features or data.

Authentication:

Authentication is the process of verifying the identity of a user. In your project, the authentication process is implemented using a combination of **username/password** authentication.

- **User Registration:**
    - When a user registers (via the /register API), the server checks if the username or email already exists. If not, the password is hashed using bcrypt before being saved to the database, ensuring secure storage.
    - After successful registration, the user is assigned a user type (e.g., customer, restaurant, admin). The server stores the user information in the User collection in MongoDB.
- **User Login:**
    - During login (via the /login API), the user's username and password are sent to the server. The server first checks if the username exists in the database.
    - If the username exists, the password entered by the user is compared with the stored hashed password using bcrypt.compare().
    - If the password matches, the user is authenticated, and relevant user data (e.g., userId, username, usertype, email) is returned in the response. If authentication fails, an error message is sent.
- **Session Management (Client-side):**
    - Once the user is authenticated, the application stores the user data in the localStorage (e.g., userId, username, usertype, email). This allows the client to maintain the user session between page reloads.
    - On subsequent visits, the application checks for the presence of this data in localStorage to authenticate the user without requiring them to log in again.

- In the case of logout, the localStorage is cleared to invalidate the session.

Authorization:

Authorization determines what an authenticated user is allowed to do based on their roles or permissions.

- **Role-based Access Control (RBAC):**
    - The project has three main user types: customer, admin, and restaurant. Each user type has different permissions and access levels.
    - When a user logs in, their usertype is saved in localStorage. Based on this, the application redirects users to different routes:
        - **Customers** are redirected to the homepage (/).
        - **Admins** are redirected to the admin dashboard (/admin).
        - **Restaurant owners** are redirected to their restaurant management page (/restaurant).
- **Frontend Authorization:**
    - On the frontend, certain routes or pages may be restricted based on the user's role. For instance, only admin users can access the admin dashboard, and restaurant users can access their restaurant-specific pages.
    - These restrictions are enforced using React Router, where the app checks the user's role before allowing access to protected routes.
- **Backend Authorization:**
    - On the backend, you can implement additional security by validating the user's role before processing requests. For example, routes that require admin access (e.g., approve-user, reject-user, etc.) can check if the logged-in user has the admin role before proceeding with the action.
    - This is typically done by checking the usertype stored in the session or request header before processing sensitive operations.

User Interface:

The user interface (UI) of the platform is crafted to provide a seamless and intuitive experience for all users. It features a responsive design with clear navigation menus and visually appealing layouts, ensuring accessibility across devices.

Customer Page: Customers have a streamlined experience for browsing products, managing their cart, and placing orders. Forms like registration and login are user-friendly and efficient.

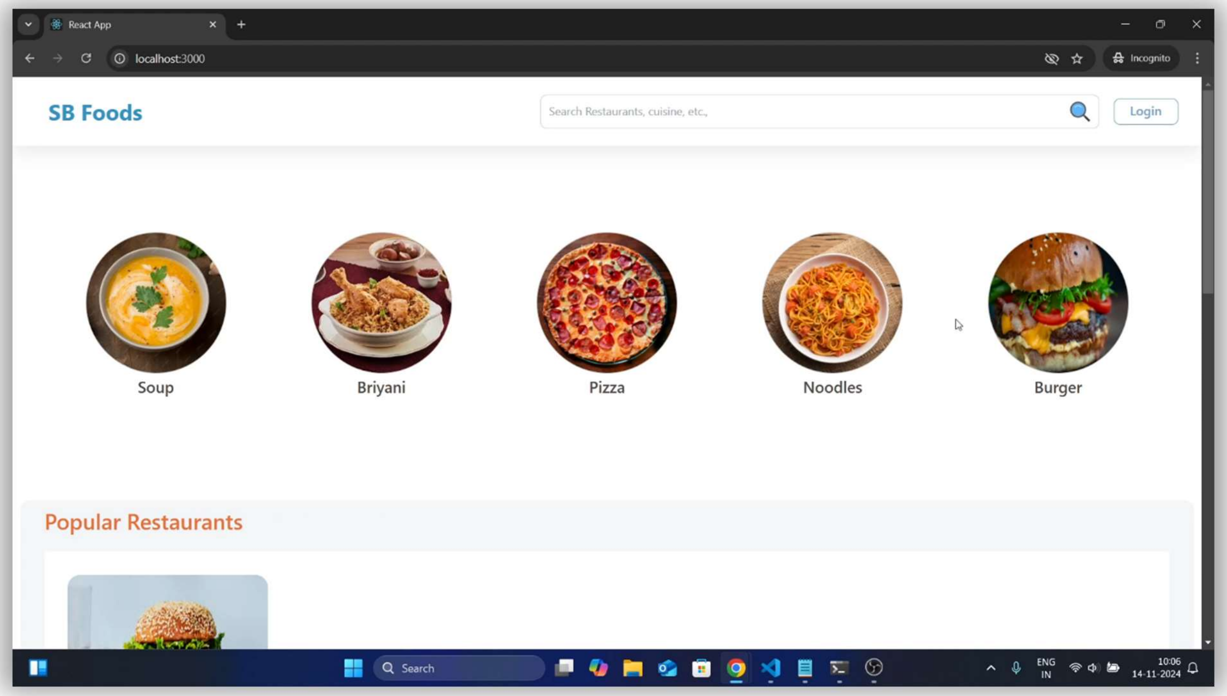


Fig. 5 – Landing page

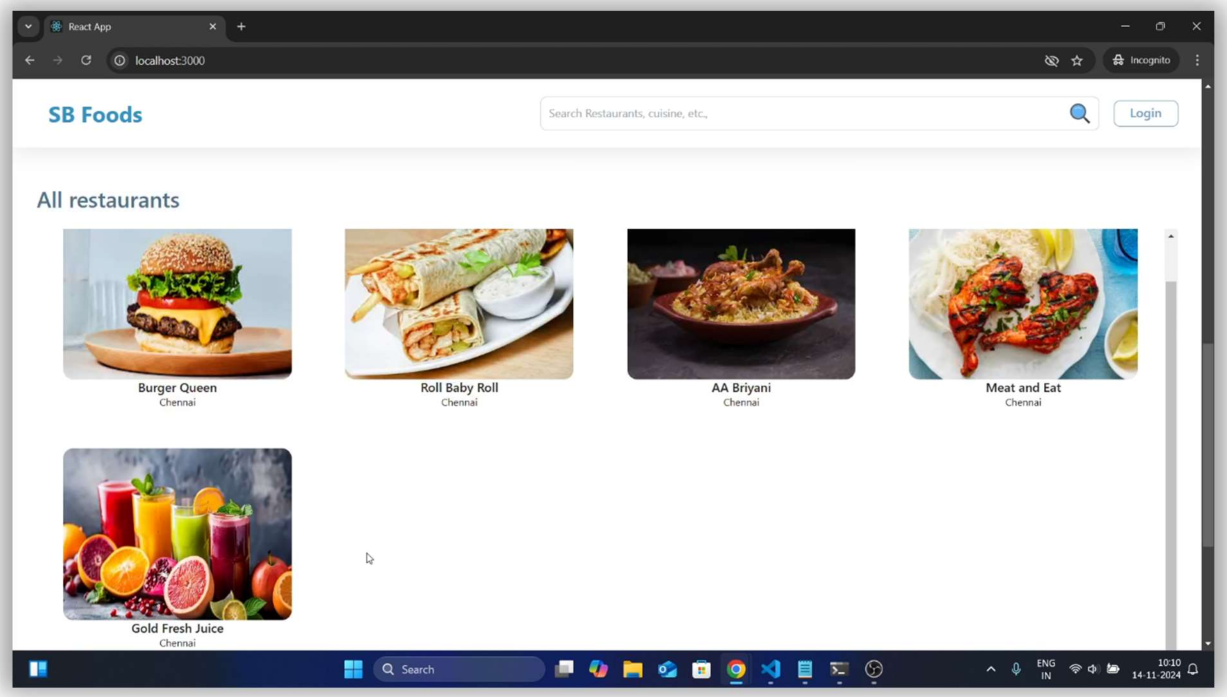
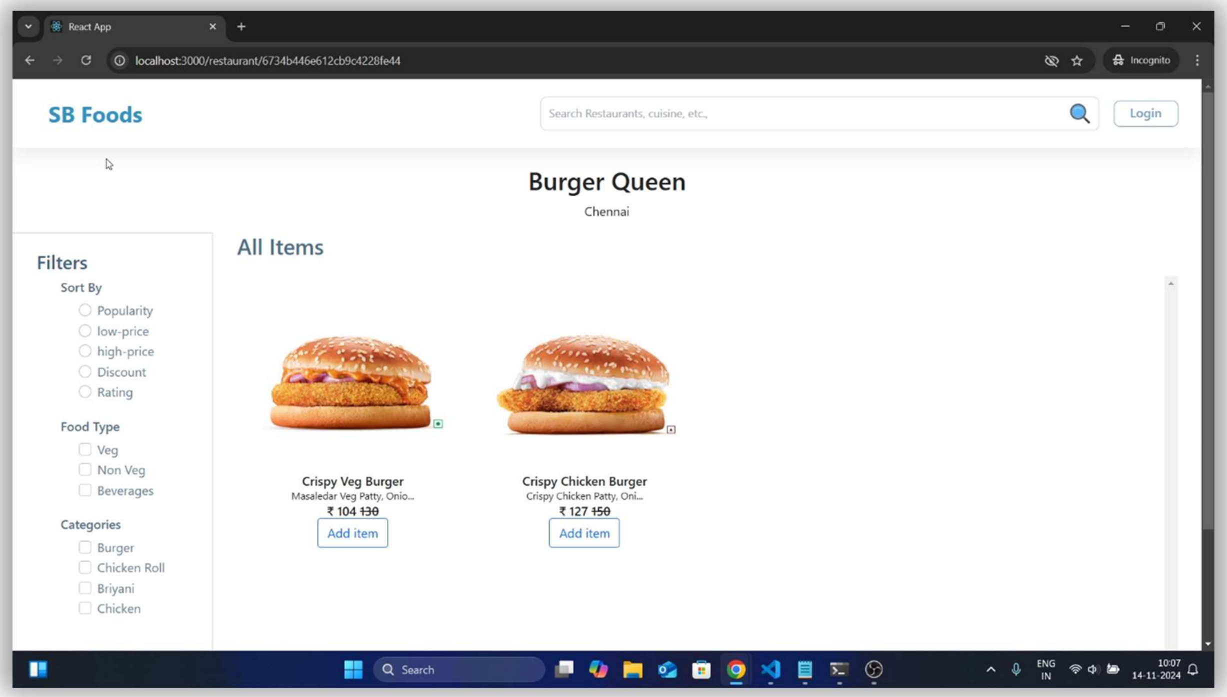


Fig. 6 – Restaurants



Burger Queen

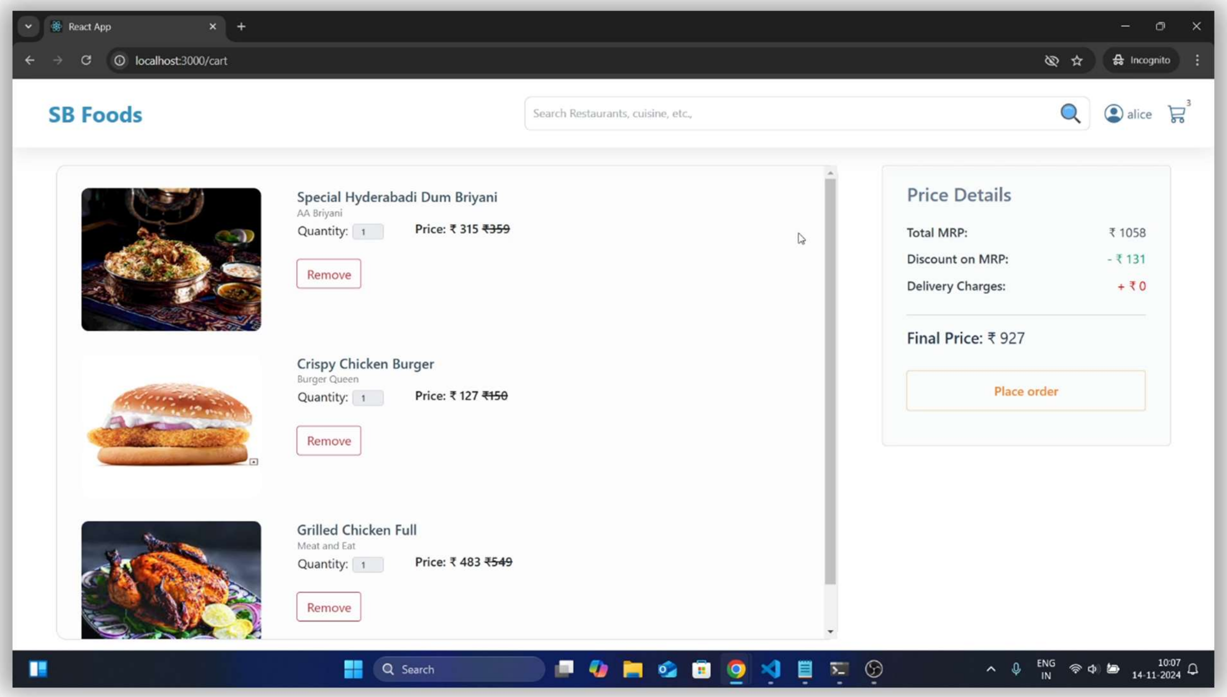
Chennai

All Items

Crispy Veg Burger
Masaledar Veg Patty, Onio...
₹ 104 ~~130~~

Crispy Chicken Burger
Crispy Chicken Patty, Oni...
₹ 127 ~~150~~

Fig. 7 – Restaurant Menu



Special Hyderabadi Dum Briyani
AA Briyani
Quantity: 1 Price: ₹ 315 ~~₹359~~

Crispy Chicken Burger
Burger Queen
Quantity: 1 Price: ₹ 127 ~~₹150~~

Grilled Chicken Full
Meat and Eat
Quantity: 1 Price: ₹ 483 ~~₹549~~

Price Details

Total MRP:	₹ 1058
Discount on MRP:	- ₹ 131
Delivery Charges:	+ ₹ 0

Final Price: ₹ 927

Fig. 8 – User Cart

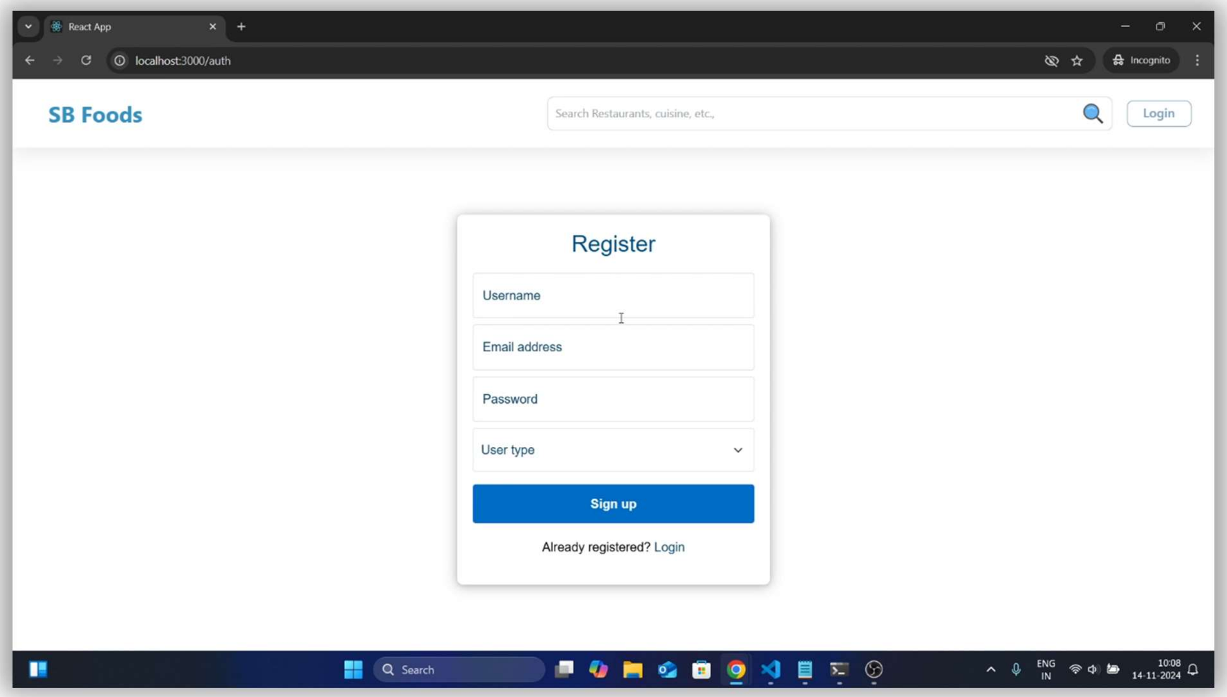
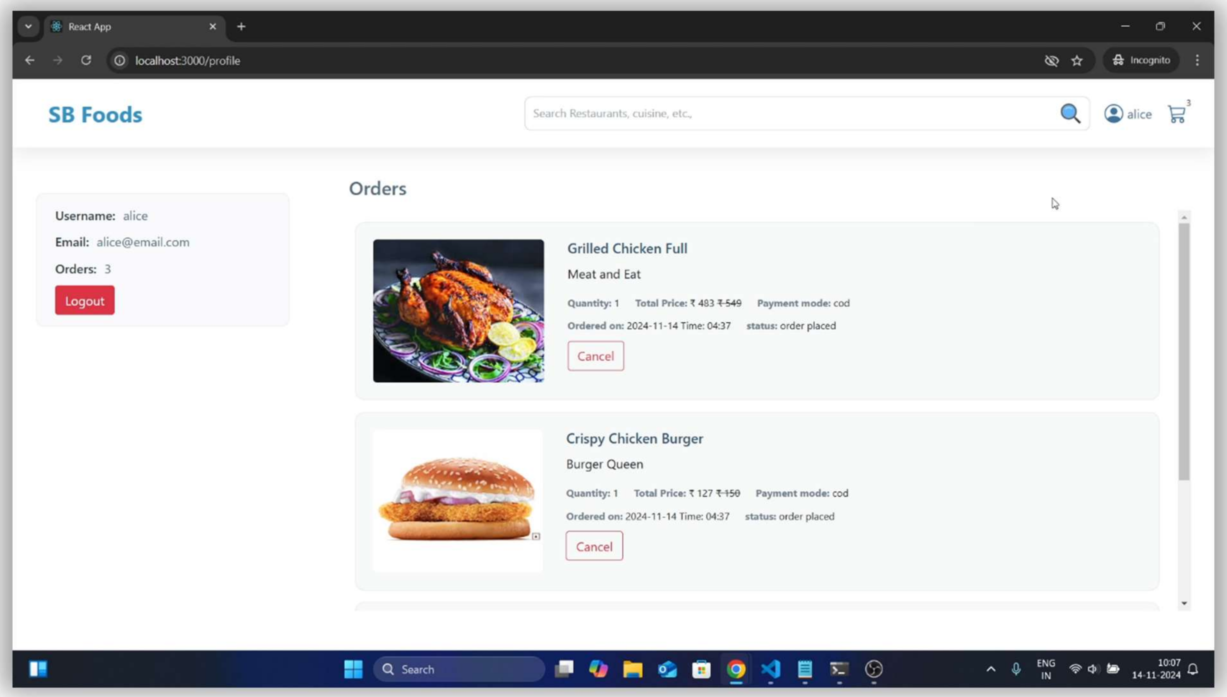


Fig. 9 – Authentication



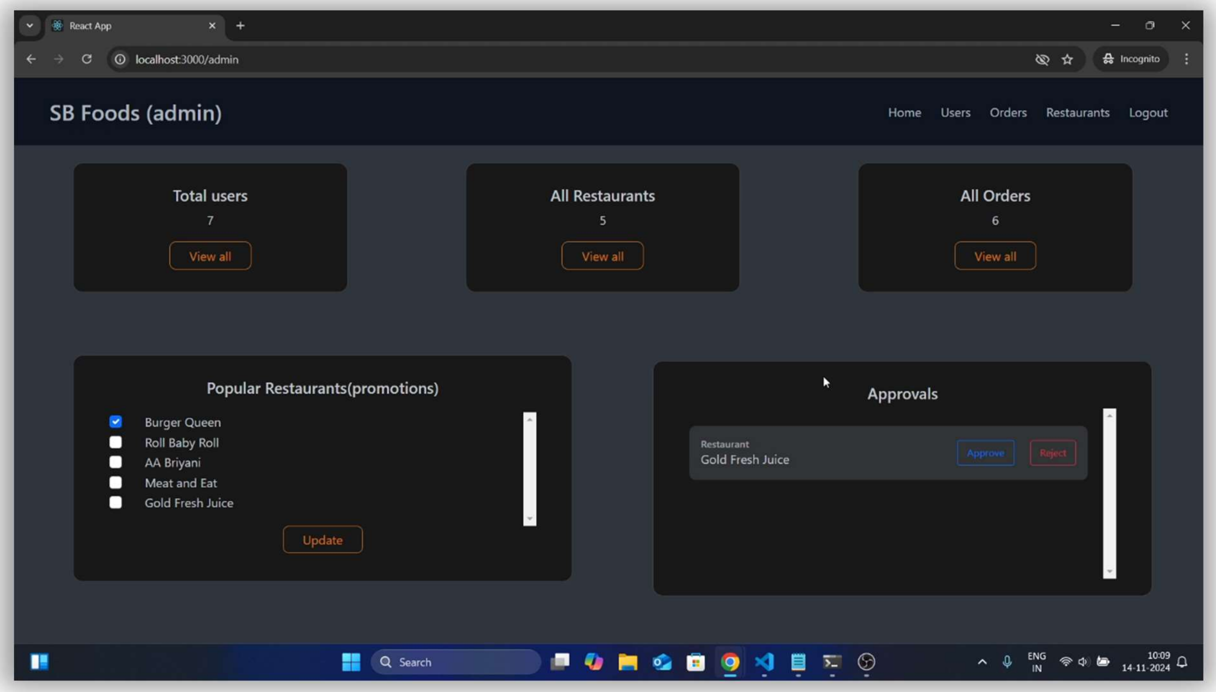
Username: alice
Email: alice@email.com
Orders: 3

Grilled Chicken Full
Meat and Eat
Quantity: 1 Total Price: ₹ 483 ~~₹ 549~~ Payment mode: cod
Ordered on: 2024-11-14 Time: 04:37 status: order placed

Crispy Chicken Burger
Burger Queen
Quantity: 1 Total Price: ₹ 127 ~~₹ 150~~ Payment mode: cod
Ordered on: 2024-11-14 Time: 04:37 status: order placed

Fig. 10 – User Profile

Admin Page: Designed for administrators to manage the platform efficiently. It includes features to approve or reject restaurant registrations, monitor users, manage orders, and oversee promotions. The interface is structured to allow quick and effective decision-making.



Total users
7

All Restaurants
5

All Orders
6

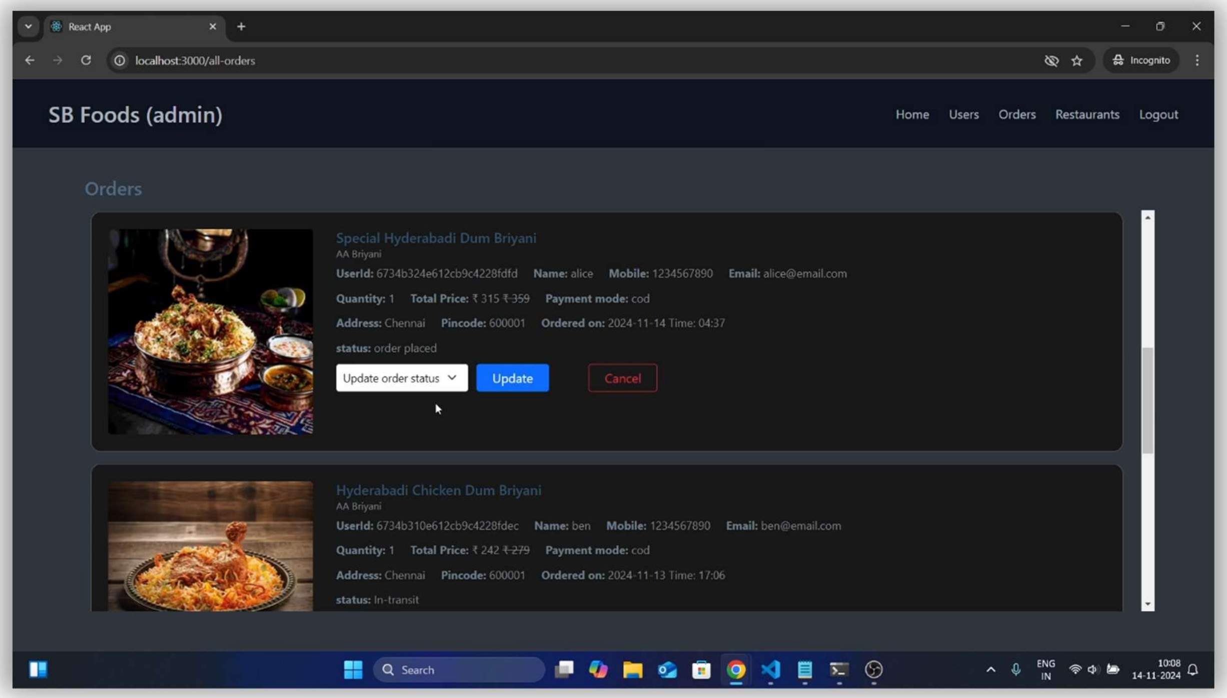
Popular Restaurants(promotions)

[✓ Burger Queen]
Roll Baby Roll
AA Briyani
Meat and Eat
Gold Fresh Juice

Approvals

Restaurant
Gold Fresh Juice

Fig. 11 – Restaurant Approval



Orders

Special Hyderabadi Dum Briyani
AA Briyani
UserId: 6734b324e612cb9c4228fdfd **Name:** alice **Mobile:** 1234567890 **Email:** alice@email.com
Quantity: 1 **Total Price:** ₹ 315 ~~₹ 359~~ **Payment mode:** cod
Address: Chennai **Pincode:** 600001 **Ordered on:** 2024-11-14 Time: 04:37
status: order placed

Hyderabadi Chicken Dum Briyani
AA Briyani
UserId: 6734b310e612cb9c4228fdec **Name:** ben **Mobile:** 1234567890 **Email:** ben@email.com
Quantity: 1 **Total Price:** ₹ 242 ~~₹ 279~~ **Payment mode:** cod
Address: Chennai **Pincode:** 600001 **Ordered on:** 2024-11-13 Time: 17:06
status: In-transit

Fig. 12 – All orders

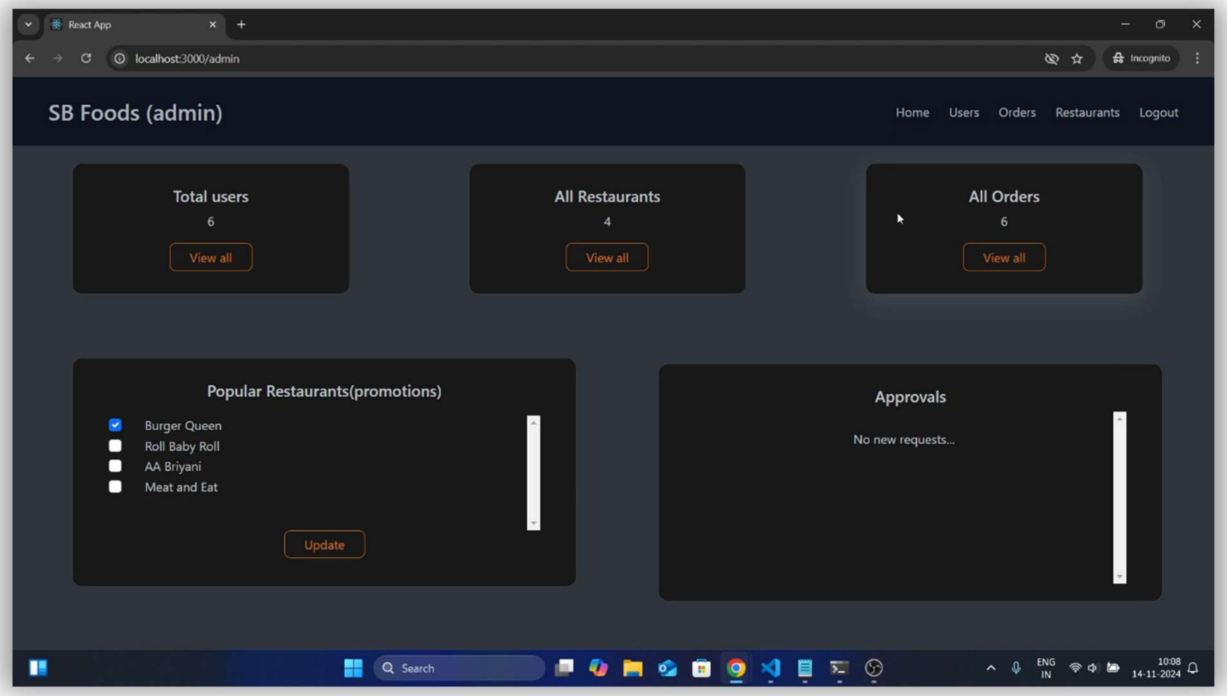


Fig. 13 – Admin Dashboard

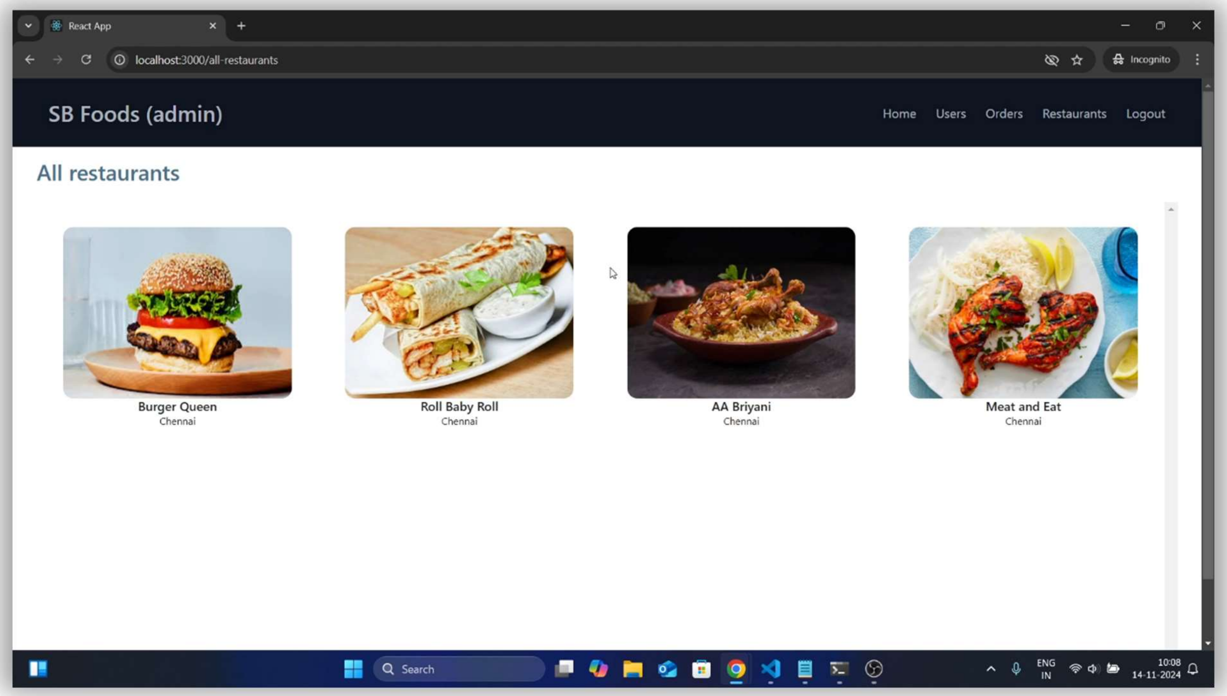
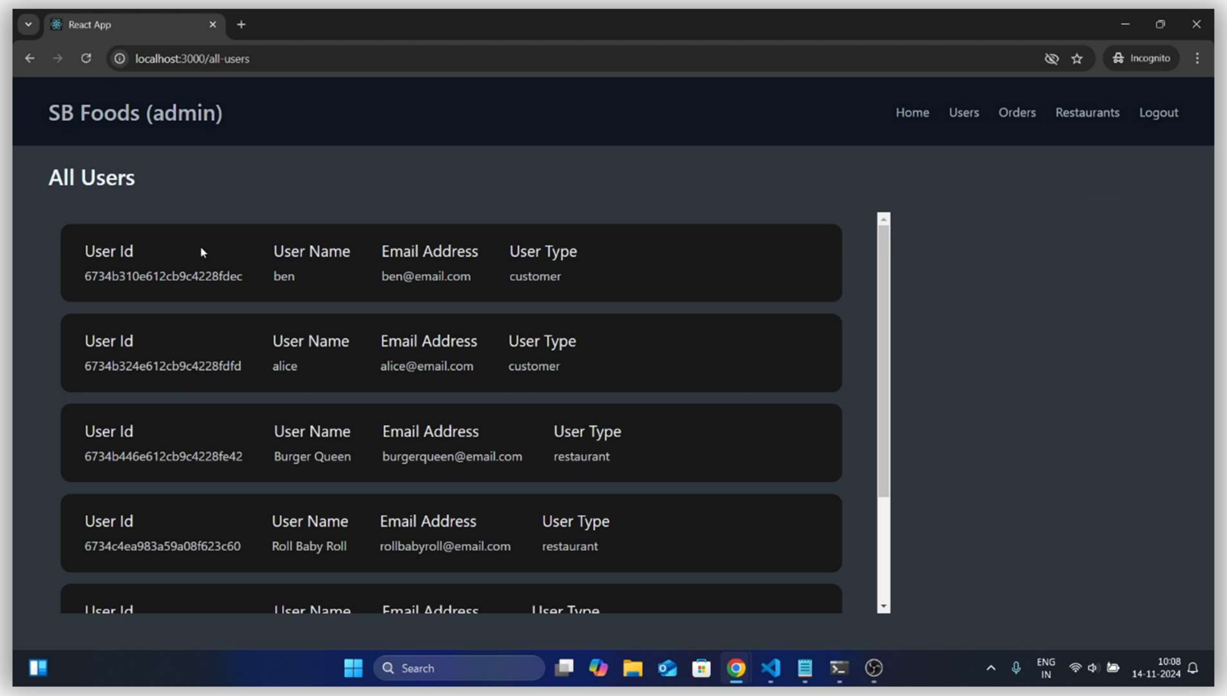


Fig. 14 – All Restaurants



React App

localhost:3000/all-users

SB Foods (admin)

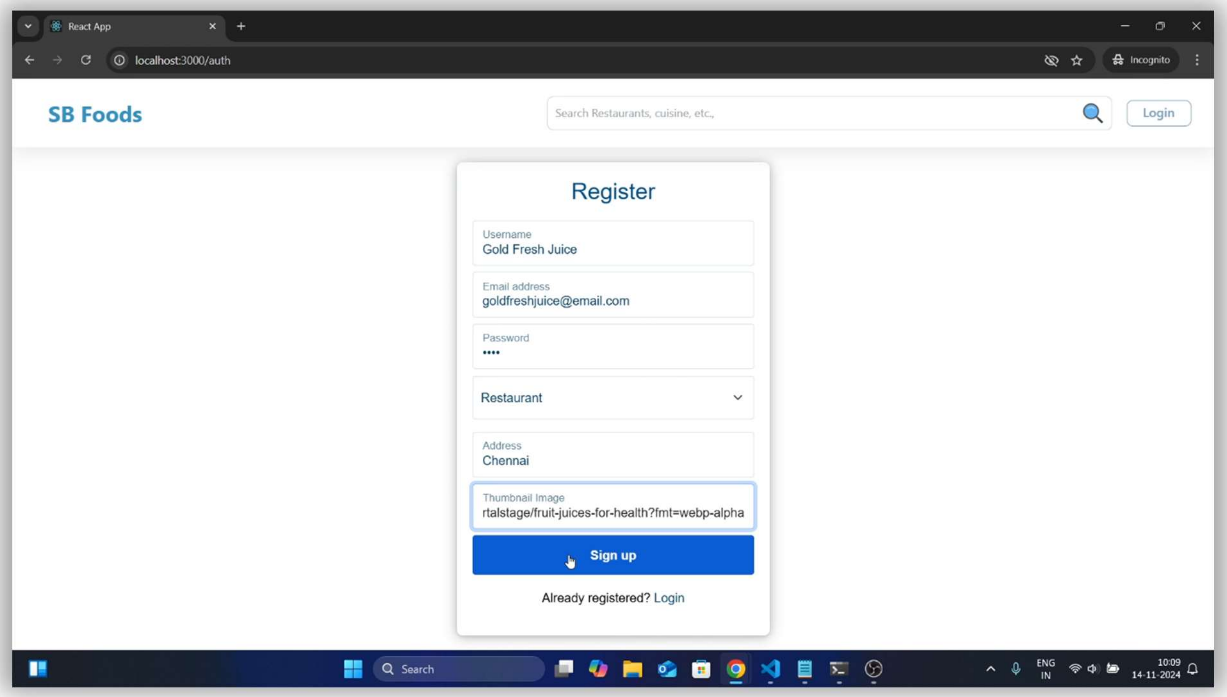
Home Users Orders Restaurants Logout

All Users

User Id	User Name	Email Address	User Type
6734b310e612cb9c4228fdec	ben	ben@email.com	customer
6734b324e612cb9c4228fdfd	alice	alice@email.com	customer
6734b446e612cb9c4228fe42	Burger Queen	burgerqueen@email.com	restaurant
6734c4ea983a59a08f623c60	Roll Baby Roll	rollbabyroll@email.com	restaurant

Fig. 15 – All Users

Restaurant Page: Tailored for restaurant owners, this page allows them to manage their profiles, add or update menu items, and track orders. It provides a dashboard for insights into their business performance.



React App

localhost:3000/auth

SB Foods

Search Restaurants, cuisine, etc.,

Login

Register

Username: Gold Fresh Juice

Email address: goldfreshjuice@email.com

Password: ••••

Restaurant

Address: Chennai

Thumbnail Image: rtalstage/fruit-juices-for-health?fmt=webp-alpha

Sign up

Already registered? Login

Fig. 16 –Registration of Restaurant

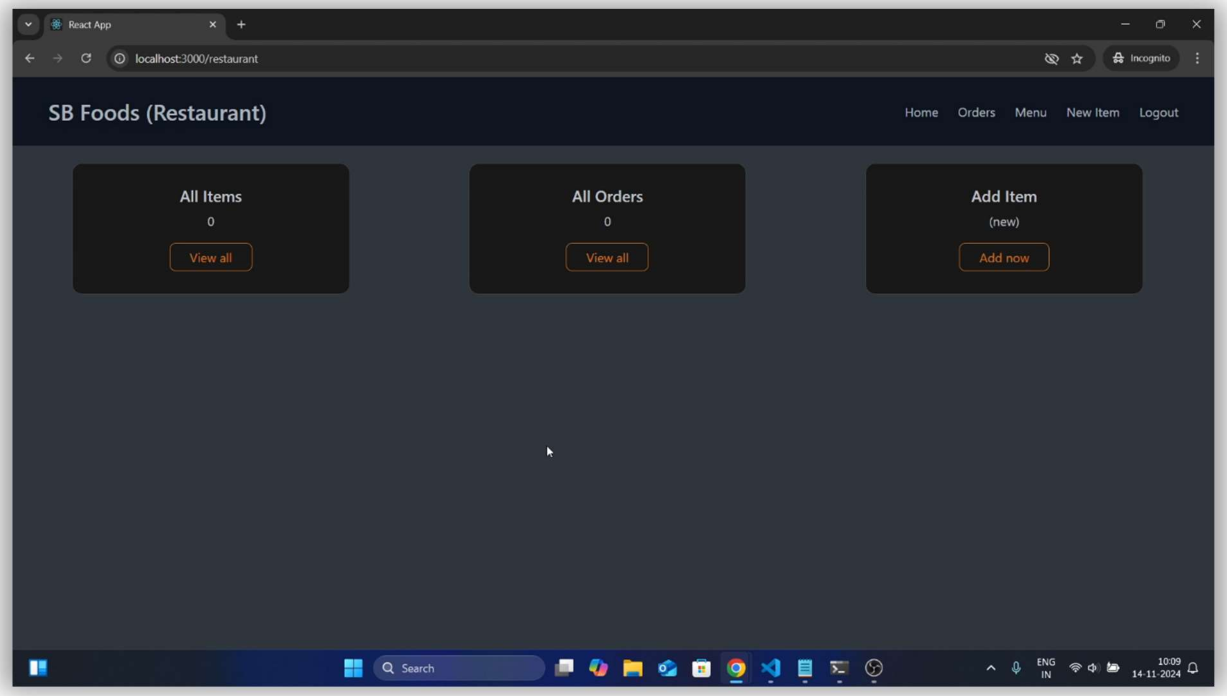


Fig. 17 – Restaurant Dashboard

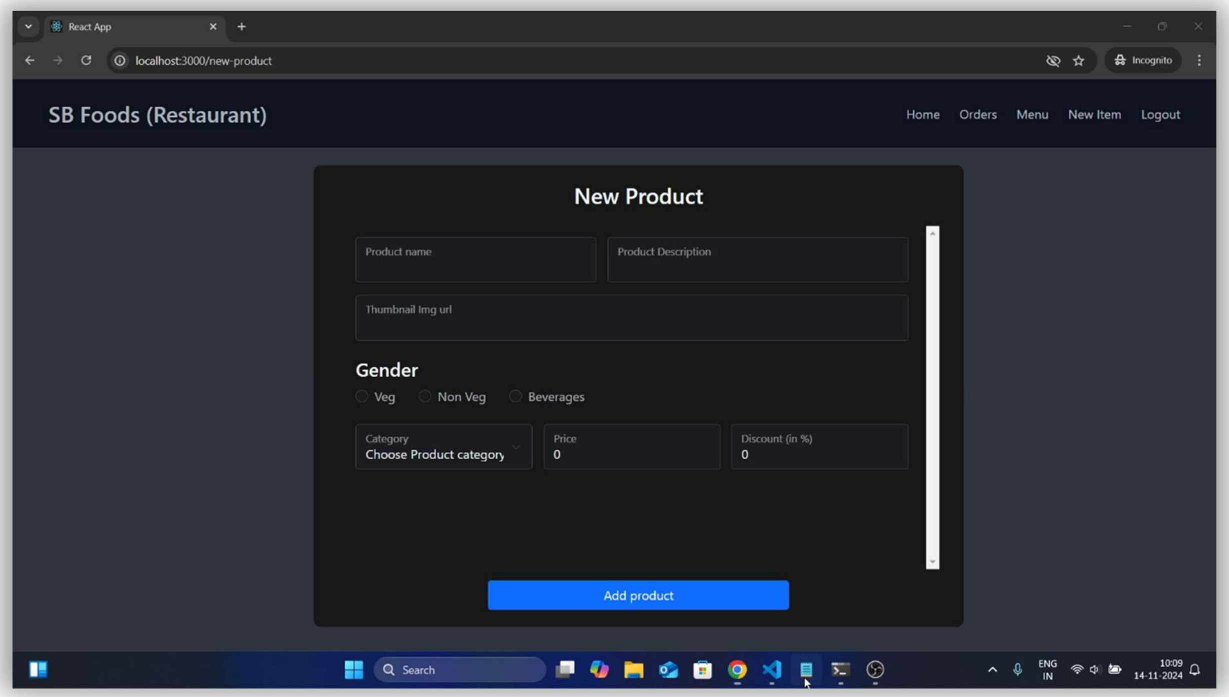


Fig. 18 – New Item

Testing:

The testing strategy for this project is designed to ensure functionality, reliability, and performance at every level of development. The following methodologies are applied:

Unit Testing

- **Objective**: Verify the correctness of individual components and functions in isolation.
- **Scope**:
  - Test React components to ensure they render as expected.
  - Validate utility functions and helper methods.
- **Implementation**:
  - The App.test.js file uses **React Testing Library** and **Jest** to check if the application renders specific elements correctly (e.g., checking for text "learn react").
  - Custom matchers from **jest-dom** (e.g., toBeInTheDocument) simplify DOM testing.

Integration Testing

- **Objective**: Test the interaction between components and API calls.
- **Scope**:
  - Validate the front-end integration with back-end services.
  - Test data flow across application layers.
- **Implementation**:
  - Mock APIs to simulate responses and validate the behavior of components making API calls.

Performance Monitoring

- **Objective**: Measure the performance of the application to improve user experience.
- **Scope**:
  - Capture metrics like Cumulative Layout Shift (CLS), First Input Delay (FID), and Largest Conceitful Paint (LCP).
- **Implementation**:
  - The reportWebVitals.js file integrates **Web Vitals** for real-time performance monitoring.

Manual Testing

- **Objective**: Ensure the application works as intended through exploratory testing.
- **Scope**:
  - Test user flows, UI responsiveness, and edge cases.
  - Verify layout consistency across different devices and browsers.

Tools Used:

Jest

A JavaScript testing framework used for running unit and integration tests and it provides test execution, mocking, and assertion utilities.

React Testing Library

Simplifies testing of React components by focusing on user interactions and outputs.

jest-dom

Extends Jest with custom matchers for testing DOM elements.

Web Vitals

Used in reportWebVitals.js to measure and track critical performance metrics.

Browser DevTools

A manual testing tool to debug front-end behavior and analyze performance.

Demo:

To get a better understanding of the platform's functionality and user experience, check out our demo video. It provides a walkthrough of key features such as user registration, login, browsing products, managing orders, and admin functionalities.

Link: https://drive.google.com/file/d/1gD4g7Pg-4ZH0trV9lhlPAqRwgUpIp7qt/view

Known Issues:

- **Cart Synchronization Delay**: Changes made to the cart, such as adding or removing items, may take a few seconds to reflect due to API response times.
- **Responsive Design Limitations**: Some elements of the admin and restaurant pages may not render correctly on smaller screen devices, requiring improvements in responsiveness.

- **Error Handling**: Certain API errors are not displayed with user-friendly messages, making it harder for users to understand the problem.
- **Login Validation**: The login system currently provides limited feedback when credentials are incorrect, which can confuse users.
- **Data Filtering**: Search and filtering features in the product and restaurant lists could use optimization for faster and more accurate results.
- **Session Timeout**: User sessions are not automatically invalidated after prolonged inactivity, potentially causing security concerns.

Future Enhancements:

Recommendation System:

- Integrate machine learning to suggest food items or restaurants based on user behaviour, preferences, and past orders.

Real-Time Notifications:

- Implement push notifications for order status updates, promotional offers, and discounts.
- Add email or SMS alerts for cart reminders and order delivery notifications.

Mobile Application:

- Develop dedicated mobile apps for Android and iOS to make the platform more accessible.

Payment Integration:

- Add a secure online payment system to allow customers to pay via credit/debit cards, UPI, or wallets.
- Introduce payment tracking for admin and restaurants.

Restaurant Features:

- Allow restaurants to manage their menu, track orders, and analyse sales within their dashboard.
- Introduce inventory tracking to assist restaurants in managing their stock.

Multi-Language Support:

- Add language options to cater to a diverse audience.

Feedback System:

- Implement a detailed review and rating system for users to share feedback on food and service quality.
- Analyse feedback to improve the platform and restaurant services.

Improved Security Measures:

- Enhance authentication methods by introducing two-factor authentication.
- Strengthen data protection and encryption mechanisms to ensure user safety.

AI-Powered Chatbot:

- Introduce a chatbot to assist users with queries, order placement, and issue resolution.

Order Tracking:

- Provide real-time tracking of orders with estimated delivery times and driver information.

Scalability:

- Optimize the system for scalability to handle increased user traffic and additional restaurants as the platform grows.

Conclusion:

The development of this project represents a concerted effort to create a robust, user-friendly application that addresses the needs of various stakeholders, including customers, restaurant owners, and administrators. By leveraging modern technologies such as React.js for the frontend and Node.js with Express for the backend, the project achieves a balance between performance, scalability, and maintainability.

Key features such as dynamic routing, modular architecture, and responsive design ensure a seamless user experience across devices, while secure authentication and well-defined APIs guarantee a reliable and efficient backend system. The project has been structured with future enhancements in mind, allowing for easy integration of new features and improvements.

This platform serves as a foundation for delivering practical solutions in the food service domain, showcasing the potential of modern web technologies. While there is room for growth and innovation, the current implementation establishes a solid baseline for further advancements and contributions.