Project Title: Food Ordering Web Application Using MERN - SB Foods

Team Members: PETA SRAVAN KUMAR
PREMKUMAR S
NITHISH NARAYANAN S
THARUN RAJ RS

Project Roles:

Frontend Developer : NITHISH NARAYANAN S

- Develop the user interface using **React.js**.
- Implement key features like **user authentication**, **cart management**, **product display**, and **profile management**.
- Ensure the design is user-friendly and responsive across devices.
- Work with the backend team to integrate **API endpoints** for dishes, orders, and user data.
- Share tasks such as handling the UI/UX design and testing the responsiveness of the app.
- **Skills**: **React.js**, CSS, HTML, JavaScript, REST APIs, attention to UI/UX design, responsive design principles.

Backend Developer : PETA SRAVAN KUMAR

- Develop and maintain the server-side logic using **Node.js** and **Express.js**.
- Create and manage **API endpoints** for user management, order processing, and product listings.
- Implement secure **user and admin authentication** and authorization.
- Optimize server-side code for performance and scalability.
- Work closely with the frontend team to ensure seamless integration of backend services.
- **Skills**: **Node.js**, **Express.js**, API development, security best practices, database integration.

Database Administrator : PREMKUMAR S

- Manage the **MongoDB** database, ensuring efficient handling of data for users, orders, carts, and products.
- Design and optimize the database schema for key collections like **Users**, **Orders**, **Products**, and **Cart**.
- Ensure data consistency and security, especially for user credentials and sensitive information.
- Support the backend developer by handling database-specific queries, optimizing performance, and ensuring smooth data transactions.
- Implement backup, monitoring, and security measures.
- **Skills**: **MongoDB**, database design, data optimization, data security, performance tuning**.**

Frontend Developer : THARUN RAJ RS

- Develop the user interface using **React.js**.
- Implement key features like **user authentication**, **cart management**, **product display**, and **profile management**.
- Ensure the design is user-friendly and responsive across devices.
- Work with the backend team to integrate **API endpoints** for dishes, orders, and user data.
- Share tasks such as handling the UI/UX design and testing the responsiveness of the app.
- **Skills**: **React.js**, CSS, HTML, JavaScript, REST APIs, attention to UI/UX design, responsive design principles

Project Overview

Purpose:

The **Food Ordering Web Application Using MERN - SB Foods** is designed to revolutionize the way customers and restaurants interact in the digital age. By providing a seamless platform for browsing menus, placing orders, and processing secure payments, it enhances convenience and efficiency for both users and business owners. Customers can easily search for their favorite dishes, view detailed descriptions and customer reviews, customize their orders, and track delivery progress in real-time, giving them full control over their dining experience.

For restaurants, the platform simplifies order management by integrating key features like **inventory tracking**, **order processing**, and **customer communication** tools. Restaurant owners can efficiently monitor their stock levels, receive instant notifications of new orders, and communicate with customers to resolve issues quickly, leading to higher customer satisfaction and loyalty.

Built using the **MERN stack** (MongoDB, Express.js, React, and Node.js), SB Foods ensures **scalability**, **high performance**, and **security**. The platform is capable of handling large volumes of orders and data, making it suitable for restaurants of all sizes, from local eateries to large chains. Furthermore, its mobile-responsive design allows customers to access the service from any device, making it easier than ever to order food on the go.

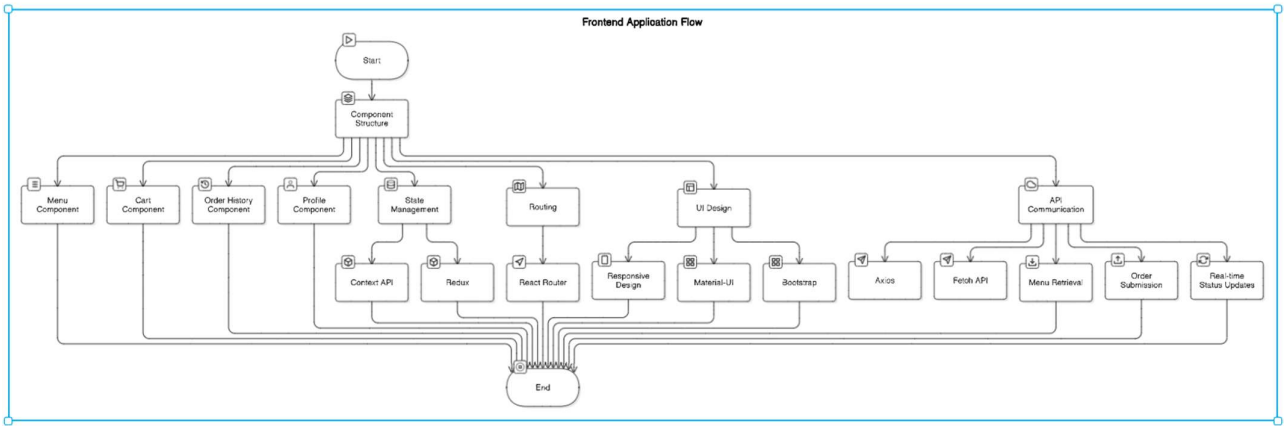
With its modern approach, **SB Foods** empowers restaurants to meet the growing demand for fast, reliable online food services, helping them expand their reach, enhance customer engagement, and optimize daily operations. This solution is particularly beneficial for businesses looking to scale without compromising on service quality, providing a competitive edge in today’s market.

Features

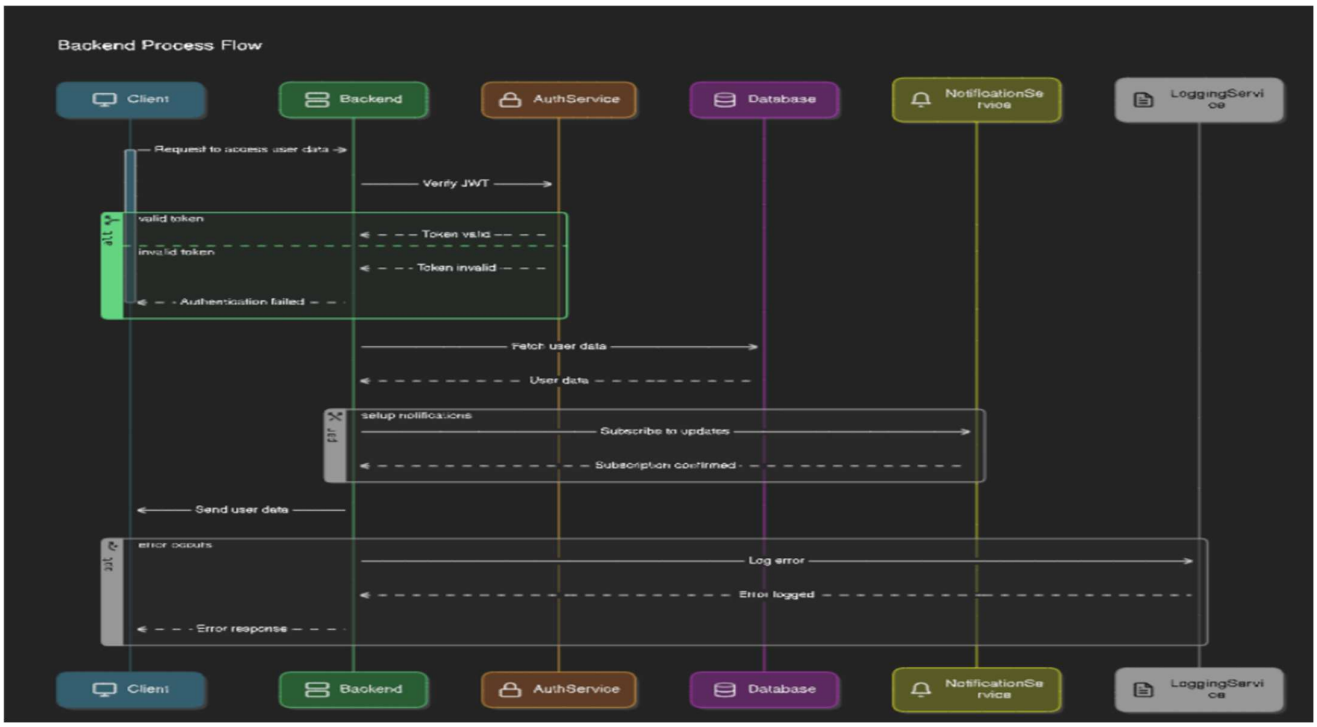
- **User Registration and Login**: Secure signup and login features ensure users can create accounts, log in, and keep their data private, using token-based authentication.
- **Detailed Menu Browsing**: Users can view menu items organized by categories, with item descriptions, images, and prices, making it easy to select what they want.
- **Order Customization**: Users can customize items (e.g., adding extras, choosing portions), creating a personalized order experience.
- **Cart System for Order Placement**: A cart system allows users to add multiple items, view total costs, and place orders with ease.
- **Secure Payment Integration**: Integrated payment gateway supports credit cards, digital wallets, and other payment methods, ensuring secure transactions.
- **Real-Time Order Tracking**: Customers can view the progress of their orders, from preparation to delivery, for increased transparency.
- **Admin Dashboard**: A backend interface for restaurant owners allows them to manage the menu, handle orders, monitor inventory, and view sales data.
- **Notifications**: Real-time notifications inform customers of their order's status, keeping them updated at every stage.
- **Rating and Review System**: Customers can rate and review menu items, providing valuable feedback for the restaurant to improve quality and service.

Architecture Diagram :

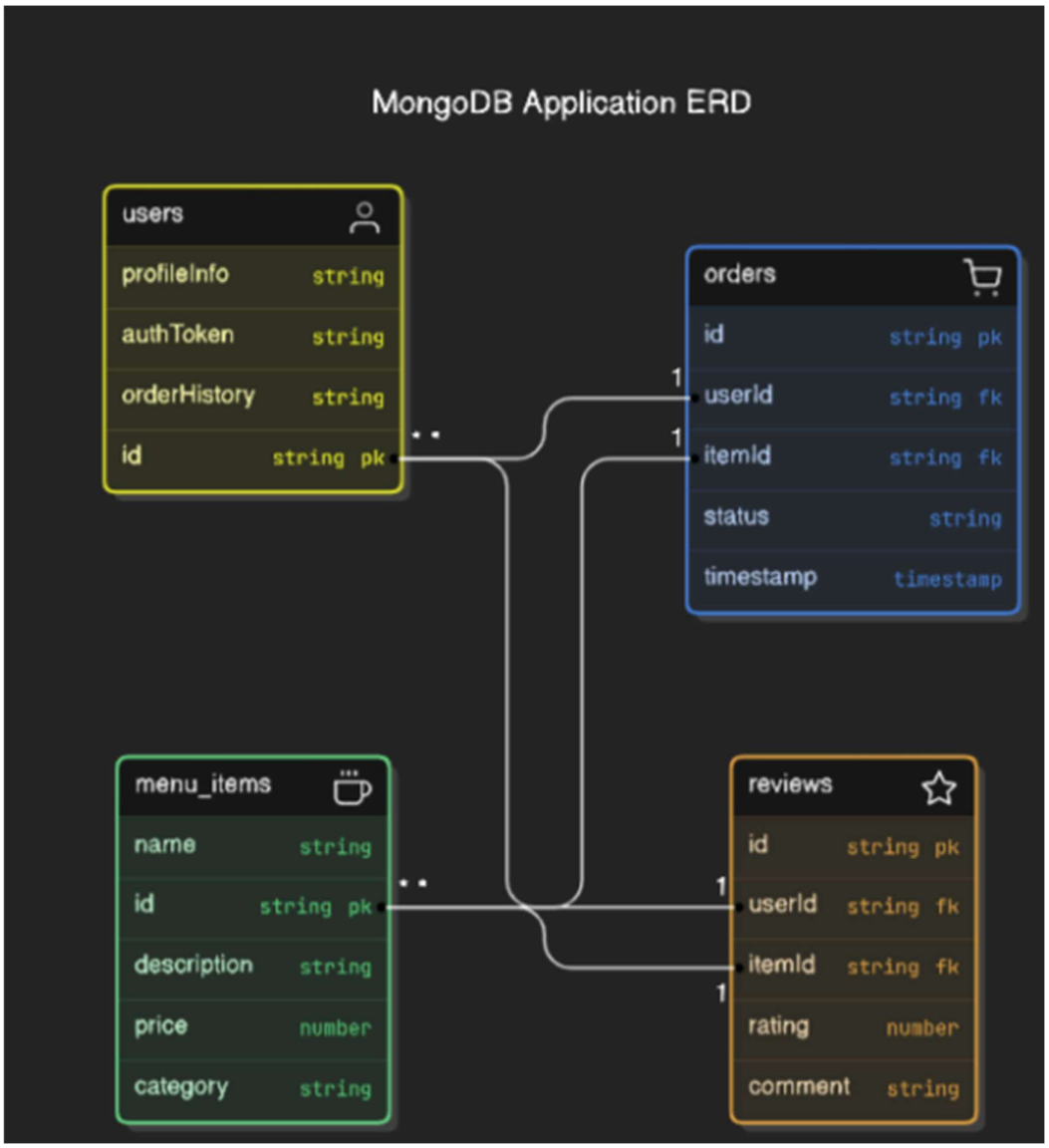
Frontend:



Backend Architecture :



Database Schema and Interactions Architecture:



Setup Instructions

Prerequisites
Before setting up the project, ensure that you have the necessary software dependencies installed on your system. These tools and technologies are essential to run the **MERN stack** application (MongoDB, Express.js, React, Node.js).

Software Dependencies:

1. **Node.js**:
    - **Purpose**: Node.js is a JavaScript runtime that allows you to run JavaScript code on the server. It's essential for running the backend of the SB Foods application.
    - **Installation**: You can download and install Node.js from the official website. Ensure that both **Node.js** and its package manager, **npm (Node Package Manager)**, are installed. Run the following commands in your terminal to verify:
    - The commands that are used :

      node -v
      npm -v
2. **MongoDB**:
    - **Purpose**: MongoDB is a NoSQL database used for storing user data, orders, products, etc. It's the database management system for the SB Foods application.
    - **Installation**: Download and install MongoDB from the official website. After installation, make sure the MongoDB service is running. You can verify it by running the following commands:
    - The commands that are used :

      mongo --version
3. **Git**:
    - **Purpose**: Git is used for version control and allows you to clone the project repository from GitHub.
    - **Installation**: You can download Git from the official website. After installation, verify by running:
    - The commands that are used :

      git --version
4. **Text Editor** (Optional but recommended):
    - It's recommended to use a code editor like **Visual Studio Code** for writing and editing the code.

Installation Steps

Now that you've installed the prerequisites, you can set up the SB Foods project by following these steps:

Step-by-Step Guide:

1. **Clone the Repository**:
    - First, navigate to the directory where you want to store the project in your terminal.
    - Clone the project from the GitHub repository using the following command:
    - The commands are used :
      git clone https://github.com/your-username/sb-foods.git
2. **Navigate into the Project Directory**:
    - Once cloned, change into the project directory using:
    - The directry are used
      cd sb-food
3. **Install Dependencies**:
    - The project contains both **frontend** and **backend** folders. Each folder has its own set of dependencies that need to be installed.
    - First, install backend dependencies by running the following commands:
    - The commands are used :
      cd backend
      npm install

This command will install all the necessary packages required to run the Node.js backend, as listed in the package.json file.

- Then, install frontend dependencies by navigating to the frontend folder and running:
- The commands are used :
  cd ../frontend
  npm install

This will install all React.js-related dependencies for the frontend.

4. **Set Up Environment Variables**:
    - You need to set up environment variables for the project to work properly. These variables usually include sensitive information like database connection URLs, API keys, and port numbers. The project likely includes a .env.example file, which serves as a template for your environment configuration.
    - In the **backend** folder, create a .env file by copying the example:

- The commands are used :

  cp .env.example .env
- Open the .env file and configure the necessary variables. Here's a sample configuration:
- The commands are used :

  PORT=5000
  MONGODB_URI=mongodb://localhost:27017/sbfoods
  JWT_SECRET=your_jwt_secret_key
  - **MONGODB_URI**: Set this to the MongoDB connection string. If you're running MongoDB locally, use mongodb://localhost:27017/sbfoods (or your custom database name).
  - **JWT_SECRET**: This is a secret key for signing JWT tokens used for user authentication. You can generate any random string for this.
- If the **frontend** needs environment variables, set them up similarly in the **frontend** directory.

5. **Run the Application**:
   - After setting up the environment variables and installing dependencies, you're ready to start the application.
   - First, start the backend server by navigating to the backend folder:
   - The commands are used :

     cd ../backend
     npm start

This will start the Node.js backend server, which should be running on the port specified in the .env file (e.g., http://localhost:5000).

- Then, start the frontend server by navigating to the frontend folder
- The commands are used :

  cd ../frontend
  npm start

This will start the React development server, usually on a different port like http://localhost:3000.

6. **Access the Application**:
   - Once both servers are running, you can access the SB Foods web application by visiting http://localhost:3000 (or the port where the frontend is running) in your browser.
   - The frontend will communicate with the backend, allowing you to browse the restaurant's menu, place orders, and manage customer interactions.

Folder Structure:

The project is divided into two main sections: the **Client** (React frontend) and the **Server** (Node.js backend). Each section has a well-defined folder structure that supports efficient development, scalability, and maintainability

Client Folder (React Frontend)

The client folder contains all files related to the user interface built with React. Each subfolder organizes specific types of code, promoting modularity and reusability.

- **src/**
  - The main source folder for the React application, containing all core logic and components.
  - **components/**
    - This folder houses reusable UI components, such as buttons, modals, and navigation bars, that can be used across multiple pages.
    - **Examples**: Menu.js (for displaying food items), Cart.js (for managing items in the cart), OrderStatus.js (for displaying the status of an order).
    - These components are designed to be modular, allowing for easy updates and consistency in UI elements throughout the app.
  - **pages/**
    - This folder holds the main pages of the application, each of which represents a specific section, like the homepage, checkout page, or order history page.
    - **Examples**: HomePage.js (displays restaurant menu and featured items), CheckoutPage.js (handles payment and order confirmation), OrderHistoryPage.js (displays a list of past orders for the user).
    - Each page is structured to fetch and display relevant data, interact with components, and manage user navigation.
  - **services/**

        - Contains functions for making API requests to the backend server. This folder isolates API call logic from the components, allowing components to focus solely on UI rendering.
        - **Examples**: api.js (handles general API calls), authService.js (handles authentication-related requests).
        - Centralizing API calls here simplifies error handling, retry logic, and request consistency.
    - **context/**
        - This folder manages global application state using the React Context API, making it easier to pass data across components without prop drilling.
        - **Examples**: AuthContext.js (handles user authentication state), CartContext.js (manages the items in the shopping cart).
        - By providing a centralized state management solution, this setup improves the app's performance and maintains consistency across components.
    - **utils/**
        - Contains helper functions for tasks like formatting data, validating inputs, and other operations that are used across the app.
        - **Examples**: formatCurrency.js (formats prices), validateInput.js (validates form fields for correct input).
        - This folder ensures that repetitive utility functions are stored in one place, improving code readability and reusability.
    - **App.js**
        - This is the main application component that manages routing and renders the primary components and pages of the app.
        - It acts as the app's core, using React Router to switch between different pages based on the URL.
    - **index.js**
        - The entry point of the React application, responsible for rendering <App /> and mounting the app to the root DOM element.
        - It includes initial setup code and configuration, like importing global styles and setting up React's strict mode.

- **styles/** (optional)
  - Contains CSS or styling files, ensuring that the app has a consistent look and feel.
  - **Examples**: App.css (for global styling), Menu.css (specific styles for the menu component).
  - This folder centralizes the app's styling, making it easy to apply updates or theme changes.

Server Folder (Node.js Backend)

The server folder houses the backend code, built with Node.js and Express.js. This structure separates the core backend components, improving code organization and making it easier to manage and extend the API.

- **src/**
  - The source folder for the backend application, containing the core logic and configurations.
  - **controllers/**
    - Contains functions that handle incoming requests, perform necessary business logic, and send responses back to the client.
    - **Examples**: userController.js (manages user registration, login, and profile updates), orderController.js (handles order creation, status updates, and retrieval), menuController.js (manages menu data retrieval and updates).
    - Controllers ensure that logic is separated from route definitions, making code easier to maintain and update.
  - **routes/**
    - Defines all API routes organized by feature or resource, linking each route to its respective controller function.
    - **Examples**: userRoutes.js (handles routes like /register and /login), orderRoutes.js (manages /orders and related endpoints), menuRoutes.js (for menu-related endpoints).
    - This modular routing structure helps keep the codebase organized and makes it easy to add or modify routes as the application evolves.

- **models/**
  - Contains schema definitions and models for MongoDB using Mongoose. Each model defines the structure and data types for a specific entity.
  - **Examples**: User.js (schema for user profiles and authentication data), Order.js (schema for customer orders, including items, status, and timestamps), MenuItem.js (schema for menu items with properties like name, description, and price).
  - Models enforce data consistency and provide a structured way to interact with the database.
- **middleware/**
  - This folder includes middleware functions that are used to handle tasks like authentication, error handling, and request validation.
  - **Examples**: authMiddleware.js (checks for user authentication on protected routes), errorHandler.js (handles errors gracefully and logs them).
  - Middleware functions streamline request processing and help secure the application by performing checks before reaching the route logic.
- **config/**
  - Stores configuration files, such as database connection setup and environment variables.
  - **Examples**: db.js (configures and establishes the MongoDB connection), config.js (stores global settings and configuration values).
  - Centralizing configuration files allows easy adjustments and makes it simple to change environments (e.g., development, production).
- **utils/**
  - Contains utility functions for backend tasks, such as generating tokens, hashing passwords, or validating request data.
  - **Examples**: generateToken.js (generates JWTs for user authentication), validateRequest.js (validates data formats and values in API requests).

        - Utility functions help maintain cleaner code by avoiding repetitive logic across controllers.
    - **server.js**
        - The main entry point of the backend application, setting up the Express server, applying middleware, and defining API routes.
        - It initializes the app, connects to the database, and listens on the specified port for incoming requests.
    - **.env**
        - Environment file that stores sensitive information, such as database URIs, API keys, and JWT secrets.
        - This file is essential for maintaining security and making the app easily configurable for different environments.

Running the Application :

The *Food Ordering Web Application Using MERN - SB Foods* requires both the frontend and backend servers to be running simultaneously. Each server is run from its respective directory using npm commands. Follow these detailed instructions to get the application running locally.

1. Starting the Frontend Server (React)

The frontend, built with React, serves as the user interface of the application. Here's how to get it running:

1. **Navigate to the Client Directory**
    - Open a terminal window, and navigate to the client folder where all frontend To locate the files are present :

      cd client
2. **Install Dependencies**
    - Install all the necessary dependencies specified in the package.json file. This step is crucial to ensure that all React libraries and modules are available:
    - This command is used for the necessary libraries :

      npm install
3. **Check for Environment Variables (if any)**
    - Some React applications require environment variables for API URLs or configuration settings. If needed, create a .env file in the client directory with values like:
    - This is the command for the Port Number :

      REACT_APP_API_URL=http://localhost:5000
    - Ensure that any necessary variables are correctly set for smooth frontend-backend communication.
4. **Start the Frontend Server**
    - Start the development server by running:
    - This is the command for to run the frontend part :

      npm start
    - By default, the React server runs on http://localhost:3000.
5. **Access and Test the Frontend**
    - Open a web browser and navigate to http://localhost:3000 to interact with the frontend.
    - Check key pages (e.g., homepage, menu, cart, order tracking) to ensure the frontend is displaying correctly and able to communicate with the backend.

6. **Hot Reloading**
    - React's development server includes hot reloading, so any changes made to the frontend code will be reflected automatically in the browser without restarting the server.

2 Starting the Backend Server (Node.js and Express):

The backend, built with Node.js and Express, handles data processing, API routing, and database interactions. Follow these steps to get it up and running:

1. **Navigate to the Server Directory**
    - Open a new terminal window, and change the directory to the server folder:

      cd server
2. **Install Dependencies**
    - Install all necessary dependencies listed in the backend package.json file. This includes Express, Mongoose, and other backend libraries:

      npm install
3. **Set Up the .env File**
    - Create a .env file in the server directory to securely store configuration values:

      DB_URI=mongodb://localhost:27017/foodOrderingApp

      JWT_SECRET=your_jwt_secret_key

      PORT=5000
    - Ensure values like DB_URI (MongoDB URI) and JWT_SECRET (JWT signing secret) are set correctly for database and user authentication.
4. **Verify Database Connection**
    - Make sure MongoDB is running locally, or verify the connection to MongoDB Atlas if using a cloud database. A successful connection is crucial for handling user data, orders, and menu items.
5. **Start the Backend Server**
    - Run the following command to start the backend server:

      npm start

- By default, this server listens on http://localhost:5000. You can confirm it's running by visiting this URL or testing specific API endpoints (e.g., /api/menu, /api/orders).

6. **Test API Endpoints**
   - To ensure backend functionality, test key API routes manually using tools like Postman or Insomnia.
   - Confirm that routes like /api/auth/login, /api/orders, and /api/menu return expected results.
7. **Automatic Backend Restarting (Optional)**
   - For development, you can use nodemon (if installed) to automatically restart the server on code changes:
   - This command used for to run the server continuously :
     nodemon server.js

3. Verifying the Full Application

After both servers are running, verify that the frontend and backend can communicate effectively.

1. **Accessing the Application**
   - With the frontend running at http://localhost:3000 and the backend at http://localhost:5000, navigate to the frontend in a browser.
2. **Testing Key Functionalities**
   - Test main features of the application to confirm they work end-to-end:
     - **User Authentication**: Sign up and log in to verify JWT-based authentication.
     - **Menu Browsing**: Check if menu items load from the backend.
     - **Order Placement**: Add items to the cart, place an order, and check backend for new orders.
     - **Order Tracking**: Verify real-time order updates on the frontend.
3. **Troubleshooting Common Issues**
   - **CORS Errors**: Ensure CORS is configured on the backend to allow requests from the frontend origin (http://localhost:3000).

- **Environment Variable Issues**: Double-check .env files in both client and server directories to ensure all variables are correctly defined.

4. **Stopping the Servers**
   - To stop the servers, press Ctrl + C in each terminal window.

API Documentation

The backend of the *Food Ordering Web Application Using MERN - SB Foods* exposes a set of RESTful API endpoints. These endpoints allow the frontend to interact with the backend, handling user authentication, order processing, menu management, and more. Here's a detailed list of API endpoints, their request methods, parameters, and example responses.

1. Authentication API

Register User

- **Endpoint**: /api/auth/register
- **Method**: POST
- **Parameters**:
  - **Body**:
    - username (string): Required, the username of the new user.
    - email (string): Required, the user's email address.
    - password (string): Required, the user's password.
- **Example Request**:

json

```
{
  "username": "johndoe",
  "email": "johndoe@example.com",
  "password": "password123"
}
```

- **Example Response**:

Json

```
{
  "message": "User registered successfully",
  "user": {
    "id": "617c1f71b6b7b16c1c4e91e7",
    "username": "johndoe",
    "email": "johndoe@example.com"
  }
}
```


Login User

- **Endpoint**: /api/auth/login
- **Method**: POST
- **Parameters**:
  - **Body**:
    - email (string): Required, the user's email address.
    - password (string): Required, the user's password.
- **Example Request**:

Json


```
{
  "email": "johndoe@example.com",
  "password": "password123"
}
```


- **Example Response**:

Json


```
{
  "message": "Login successful",
  "token": "jwt-token",
  "user": {
    "id": "617c1f71b6b7b16c1c4e91e7",
    "username": "johndoe"
  }
```

}

2. Menu API

Get All Menu Items

- **Endpoint**: /api/menu
- **Method**: GET
- **Parameters**: None
- **Example Response**:

Json


```
[
  {
    "id": "617c2f71b6b7b16c1c4e91e8",
    "name": "Pizza Margherita",
    "description": "Classic pizza with tomatoes, mozzarella, and basil",
    "price": 8.99,
    "category": "Pizza",
    "image": "url_to_image"
  },
  {
    "id": "617c2f71b6b7b16c1c4e91e9",
    "name": "Spaghetti Carbonara",
    "description": "Pasta with egg, cheese, pancetta, and pepper",
    "price": 12.5,
    "category": "Pasta",
    "image": "url_to_image"
  }
]
```


Add New Menu Item

- **Endpoint**: /api/menu
- **Method**: POST

- **Parameters**:
  - **Headers**:
    - Authorization (string): Required, Bearer token for admin access.
  - **Body**:
    - name (string): Required, the name of the menu item.
    - description (string): Optional, description of the item.
    - price (number): Required, price of the item.
    - category (string): Optional, category of the item (e.g., "Pizza", "Pasta").
    - image (string): Optional, URL of the item's image.
- **Example Request**:

Json

```
{
  "name": "Tiramisu",
  "description": "Italian dessert made with coffee-soaked ladyfingers and mascarpone",
  "price": 6.5,
  "category": "Desserts",
  "image": "url_to_image"
}
```

- **Example Response**:

Json

```
{
  "message": "Menu item added successfully",
  "menuItem": {
    "id": "617c3f71b6b7b16c1c4e91ea",
    "name": "Tiramisu",
    "price": 6.5,
    "category": "Desserts"
  }
}
```

3. Order API

Place an Order

- **Endpoint**: /api/orders
- **Method**: POST
- **Parameters**:
    - **Headers**:
        - Authorization (string): Required, Bearer token for user authentication.
    - **Body**:
        - items (array of objects): Required, list of items being ordered, each with:
            - menuItemId (string): Required, the ID of the menu item.
            - quantity (number): Required, the quantity of the item.
        - totalPrice (number): Required, the total price for the order.
- **Example Request**:

Json

```
{
  "items": [
    {
      "menuItemId": "617c2f71b6b7b16c1c4e91e8",
      "quantity": 2
    },
    {
      "menuItemId": "617c2f71b6b7b16c1c4e91e9",
      "quantity": 1
    }
  ],
  "totalPrice": 30.48
}
```

- **Example Response**:

Json

```
{
  "message": "Order placed successfully",
```

```
  "order": {
    "id": "617c4f71b6b7b16c1c4e91eb",
    "status": "Pending",
    "items": [
      {
        "menuItemId": "617c2f71b6b7b16c1c4e91e8",
        "quantity": 2
      },
      {
        "menuItemId": "617c2f71b6b7b16c1c4e91e9",
        "quantity": 1
      }
    ],
    "totalPrice": 30.48
  }
}
```


Get User Orders

- **Endpoint**: /api/orders
- **Method**: GET
- **Parameters**:
    - **Headers**:
        - Authorization (string): Required, Bearer token for user authentication.
- **Example Response**:

Json


```
[
  {
    "id": "617c4f71b6b7b16c1c4e91eb",
    "status": "Pending",
    "items": [
      {
```

```
      "menuItemId": "617c2f71b6b7b16c1c4e91e8",
      "quantity": 2
    },
    {
      "menuItemId": "617c2f71b6b7b16c1c4e91e9",
      "quantity": 1
    }
  ],
  "totalPrice": 30.48,
  "date": "2023-12-01T10:20:30.000Z"
 }
]
```


Update Order Status

- **Endpoint**: /api/orders/:orderId/status
- **Method**: PATCH
- **Parameters**:
  - **Headers**:
    - Authorization (string): Required, Bearer token for admin access.
  - **URL Parameter**:
    - orderId (string): Required, the ID of the order to update.
  - **Body**:
    - status (string): Required, the new status of the order (e.g., "In Progress", "Completed").
- **Example Request**:

Json


```
{
 "status": "Completed"
}
```


- **Example Response**:

Json

```
{
  "message": "Order status updated successfully",
  "order": {
    "id": "617c4f71b6b7b16c1c4e91eb",
    "status": "Completed"
  }
}
```


Authentication and Authorization :

In the *Food Ordering Web Application Using MERN - SB Foods*, authentication and authorization are implemented to securely manage user access and protect sensitive data. The application uses **JSON Web Tokens (JWT)** to handle both authentication and authorization, ensuring that only authenticated users can access certain features and data, and that different levels of access are assigned to users and administrators.

1. Authentication with JSON Web Tokens (JWT)

- **JWT-Based Authentication**:
  - When a user logs in or registers, the backend generates a JSON Web Token (JWT), a secure token containing encoded information about the user. This token is signed with a secret key stored on the server and is unique to each user session.
  - The JWT is sent to the client (frontend) and stored locally, usually in browser storage such as **localStorage** or **sessionStorage**. The client includes this token in the header of each API request that requires user authentication, allowing the server to identify and validate the user.
- **Registration and Login**:
  - Upon registration, a new user account is created in the database, and a JWT token is generated and sent to the user.

- During login, the provided credentials (email and password) are validated. If valid, a JWT is generated and sent in the response, allowing the user to access protected routes and resources.

- **Token Validation**:
  - Each time an authenticated request is made, the token is sent in the HTTP Authorization header as a Bearer token (e.g., Authorization: Bearer <token>).
  - The backend verifies the token using the secret key. If the token is valid, the user is granted access to the requested resource; if it is invalid or expired, the user is denied access and may need to log in again.

2. Authorization and Role-Based Access Control

- **User Roles**:
  - The application assigns roles to users, typically either **standard users** (for regular customers) or **admin users** (for restaurant owners or management). User roles are stored in the database and included in the payload of the JWT.
- **Role-Based Access Control**:
  - **Standard Users**: Regular users can access endpoints related to menu browsing, order creation, and order history.
  - **Admin Users**: Admins have additional access to endpoints for managing menu items, updating order statuses, and viewing all orders.
  - Middleware checks the role of the authenticated user to ensure they have permission to perform the requested operation. For instance, only users with an "admin" role can access endpoints for adding or updating menu items.

3. Middleware for Authentication and Authorization

- **Authentication Middleware**:
  - The authentication middleware verifies the presence and validity of a JWT in each protected request. If the token is missing or invalid, the middleware returns an error response, preventing unauthorized access.

- After validating the token, the middleware decodes it to retrieve the user's information (like user ID and role) and attaches it to the request object, making it available to subsequent middleware or route handlers.

- **Authorization Middleware**:
  - For endpoints that require a specific role (e.g., admin access), the authorization middleware checks the user role from the decoded JWT. If the user role doesn't match the required permissions, the middleware denies access with an error message.

4. Token Expiration and Refresh

- **Token Expiration**:
  - To enhance security, each JWT token is set to expire after a specific period (e.g., 1 hour). After expiration, the user must log in again to obtain a new token, reducing the risk of unauthorized access if a token is somehow compromised.
- **Token Refreshing** (Optional):
  - In more complex applications, a **refresh token** system can be implemented to renew expired tokens without requiring a full re-login. Here, a longer-lived refresh token is used to request a new access token, allowing continuous access while limiting security risks.

5. Storing Tokens on the Client Side

- **Local Storage or Session Storage**:
  - The frontend stores the JWT securely, typically in localStorage or sessionStorage. This token is then included in the Authorization header for each request to authenticated endpoints.
  - **Security Consideration**: For added security, the token could be stored in an HTTP-only cookie to prevent access via JavaScript, mitigating certain security risks like cross-site scripting (XSS).

6. Example Workflow of Authentication and Authorization

1. **Registration/Login**: User registers or logs in, receiving a JWT from the backend.

2. **Token Storage**: The JWT is stored locally on the client side.
3. **Authenticated Request**: The token is attached to the Authorization header of requests to protected routes.
4. **Token Verification**: The backend verifies the token, extracts the user info, and processes the request if valid.
5. **Role-Based Access Control**: For role-specific routes, the backend checks the user's role and grants or denies access accordingly.

User Interface

To demonstrate the functionality and user experience of the *Food Ordering Web Application Using MERN - SB Foods*, here are descriptions of key UI screens with suggested screenshots or GIFs to showcase each feature.

Testing with Selenium

In the *Food Ordering Web Application Using MERN - SB Foods*, Selenium is used as a powerful tool for automated end-to-end testing. Selenium allows for in-depth, automated testing of user workflows by simulating user interactions with the web application within a real

browser. This approach ensures the UI behaves as expected under various scenarios, providing a realistic user experience and identifying issues in complex interactions.

1. Selenium Overview

Selenium is an open-source tool used primarily for testing web applications. It interacts with elements on a webpage just like a human would, enabling automated testing of full workflows. Selenium supports multiple programming languages, including JavaScript, Python, Java, and C#, allowing flexibility in how tests are written.

- **Selenium WebDriver**: The core component of Selenium used to automate browser actions (clicking, typing, scrolling).
- **Browser Compatibility**: Supports multiple browsers (Chrome, Firefox, Safari), allowing cross-browser testing.

2. Testing Strategy with Selenium

For the *SB Foods* application, Selenium is used to perform comprehensive end-to-end tests, focusing on full user workflows and complex interactions. Here's how Selenium fits into the testing strategy:

- **End-to-End Testing**:
    - Selenium tests the entire user experience, from user registration/login to ordering and tracking, replicating real user interactions.
    - **Objective**: To ensure that each workflow, from menu browsing to order placement and tracking, functions correctly.
- **Regression Testing**:
    - Selenium tests are re-run after code changes to confirm new updates don't break existing functionality.
    - **Objective**: To verify that recent changes don't introduce bugs or affect other features.
- **Cross-Browser Testing**:
    - Selenium allows tests to run on different browsers (Chrome, Firefox, Safari) to ensure the application performs consistently.

- **Objective**: To catch browser-specific issues and ensure a uniform user experience across platform

3. Selenium Test Scenarios

Common end-to-end scenarios tested with Selenium in the *SB Foods* application:

1. **User Registration and Login**
   - **Test Objective**: Verify that new users can register and existing users can log in.
   - **Actions**: Fill out registration fields, submit the form, check for confirmation. For login, enter credentials, submit, and confirm successful login.
   - **Expected Outcome**: User is successfully registered or logged in and redirected to the homepage.
2. **Menu Browsing and Item Selection**
   - **Test Objective**: Ensure users can browse the menu, view item details, and add items to the cart.
   - **Actions**: Click on menu categories, select an item, open its details, and add it to the cart.
   - **Expected Outcome**: Items are displayed correctly, and selected items appear in the cart.
3. **Cart and Checkout Process**
   - **Test Objective**: Verify that users can view their cart, modify quantities, and proceed to checkout.
   - **Actions**: Access the cart, adjust item quantities, proceed to checkout, enter payment details, and confirm the order.
   - **Expected Outcome**: The order is confirmed, and the user sees an order confirmation page.
4. **Order Tracking**
   - **Test Objective**: Ensure that users can view real-time order status updates.
   - **Actions**: Place an order, navigate to the order tracking page, and verify the order status updates.
   - **Expected Outcome**: Order status updates accurately as it progresses from preparation to delivery.
5. **Admin Dashboard (for Admin Users)**

- **Test Objective**: Ensure that admin users can access the dashboard to view and manage orders.
- **Actions**: Login as an admin, navigate to the dashboard, view incoming orders, and update order statuses.
- **Expected Outcome**: Admin functionalities work as expected, and order status updates reflect on the user side.

4. Setting Up Selenium for Testing

To implement Selenium testing, you need the following setup:

1. **Install Selenium WebDriver**:
   - Install the Selenium package compatible with your chosen programming language. For Python:

pip install selenium

2. **Download Browser Drivers**:
   - Selenium requires a specific driver for each browser (e.g., chromedriver for Chrome). Download and configure the driver, ensuring it matches the browser version used for testing.
3. **Configure Tests**:
   - Write Selenium test scripts to execute the test scenarios. These scripts simulate user actions like clicking, typing, and navigating between pages.
4. **Sample Test Script**: Here's an example of a Selenium test in Python that logs in a user:

java

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys

# Setup WebDriver for Chrome
driver = webdriver.Chrome()

# Open the Login Page
```

```
driver.get("http://localhost:3000/login")

# Enter Email and Password
driver.find_element(By.ID, "email").send_keys("johndoe@example.com")
driver.find_element(By.ID, "password").send_keys("password123")

# Submit the Login Form
driver.find_element(By.ID, "loginButton").click()

# Verify login by checking for user profile element
assert "Profile" in driver.page_source

# Close the browser
driver.quit()
```

5. Running and Managing Selenium Tests

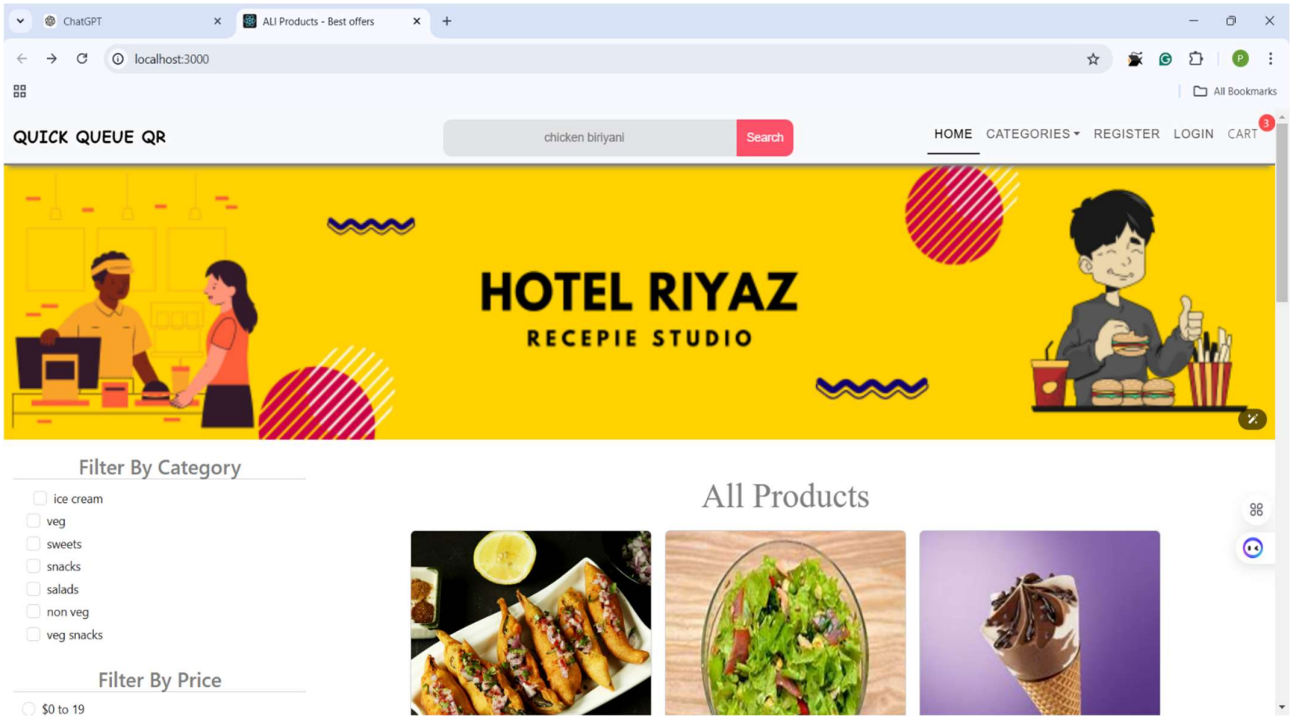
- **Test Execution**:
  - Run the Selenium tests as part of the testing pipeline, either locally or on a CI/CD platform (e.g., GitHub Actions or Jenkins).
- **Reporting and Logging**:
  - Set up reporting (e.g., using Allure or TestNG for Java) to capture test results, screenshots on failure, and logs for detailed debugging.
- **Continuous Integration (CI)**:
  - Integrate Selenium tests with a CI/CD pipeline to automate testing on each code push, providing continuous feedback on application stability.

6. Benefits of Using Selenium

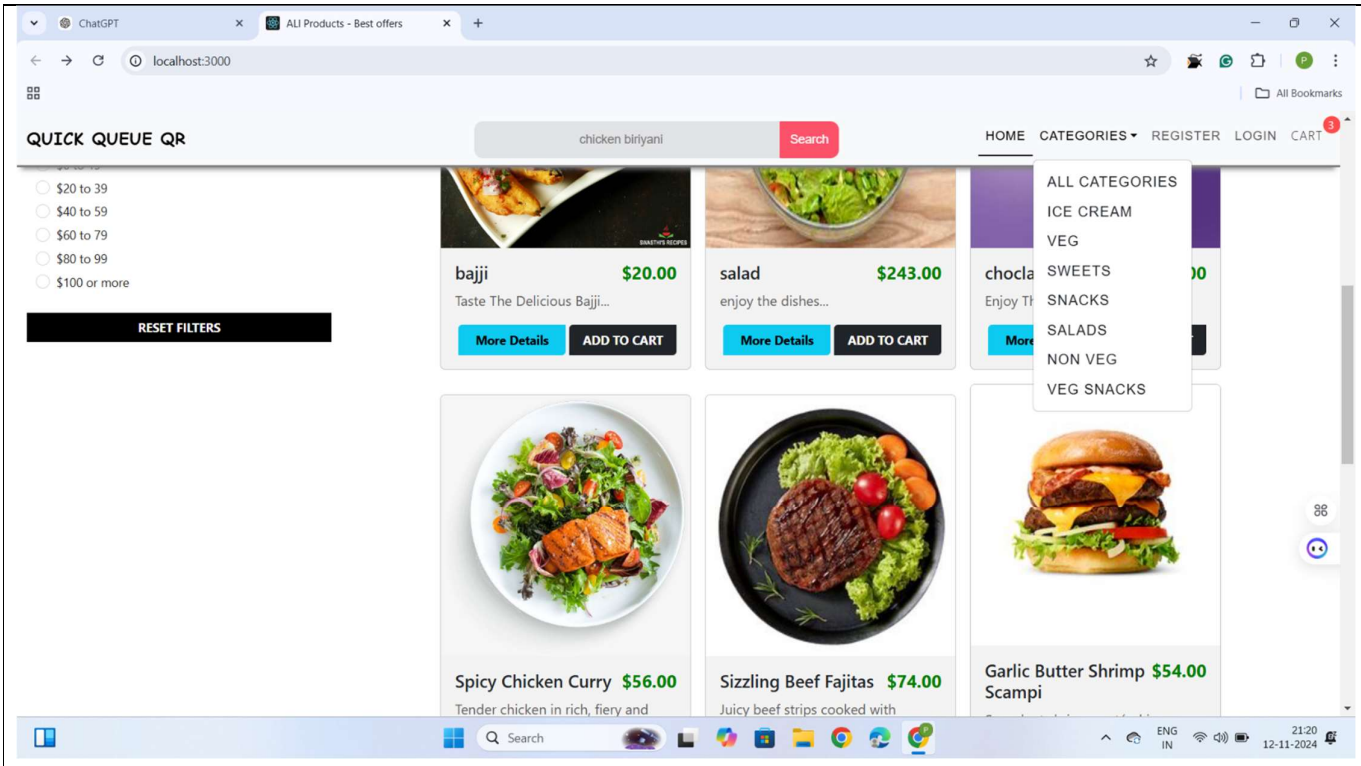
- **Realistic User Simulation**: Selenium replicates real user actions, providing a reliable measure of how the application will perform under real conditions.
- **Cross-Browser Testing**: Tests can run on multiple browsers, ensuring compatibility and consistent behavior across platforms.

- **Automated Workflow Validation**: By testing entire workflows, Selenium helps catch issues in complex interactions, enhancing the end-user experience.
- **Scalable and Flexible**: Selenium supports parallel testing across multiple environments, optimizing testing speed and efficiency.

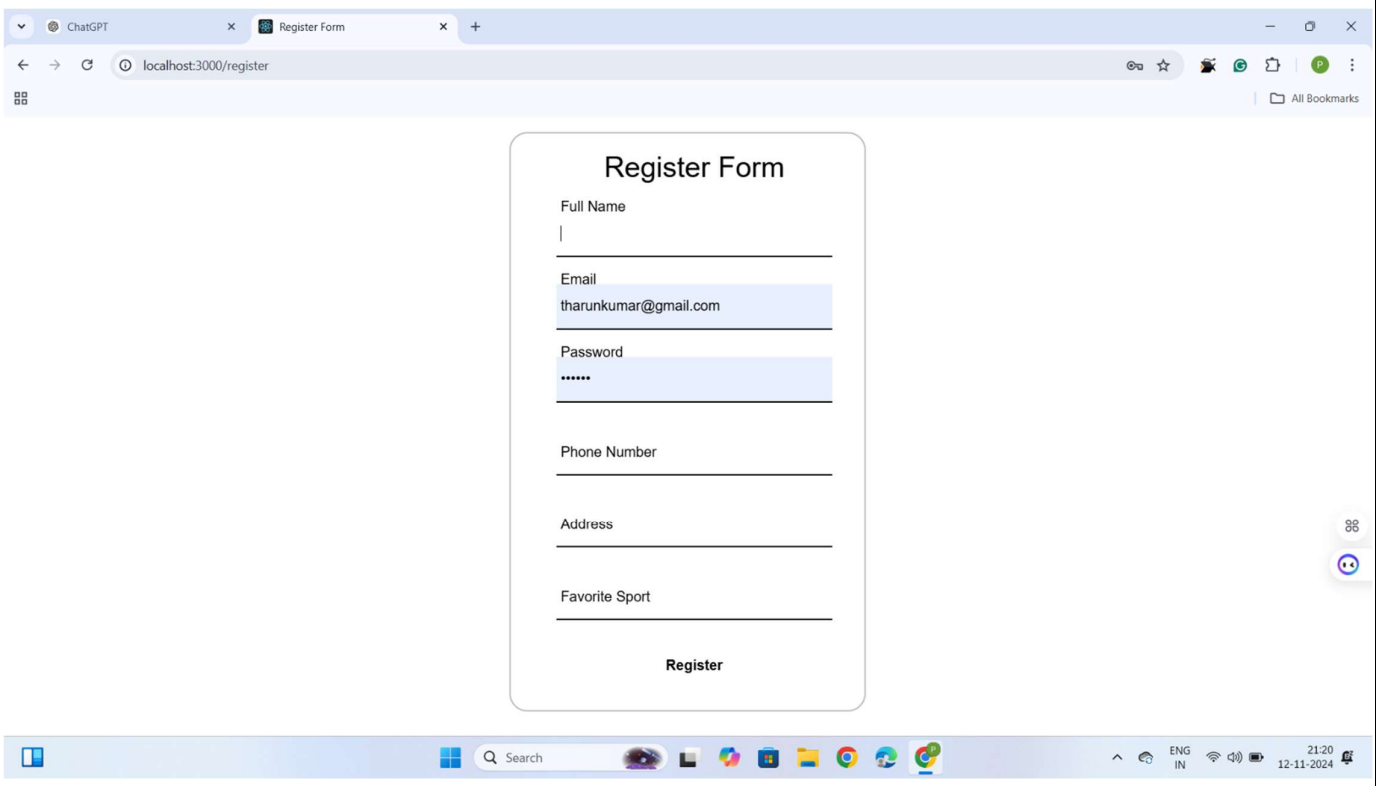
Screenshots or Demo // Out-Put:



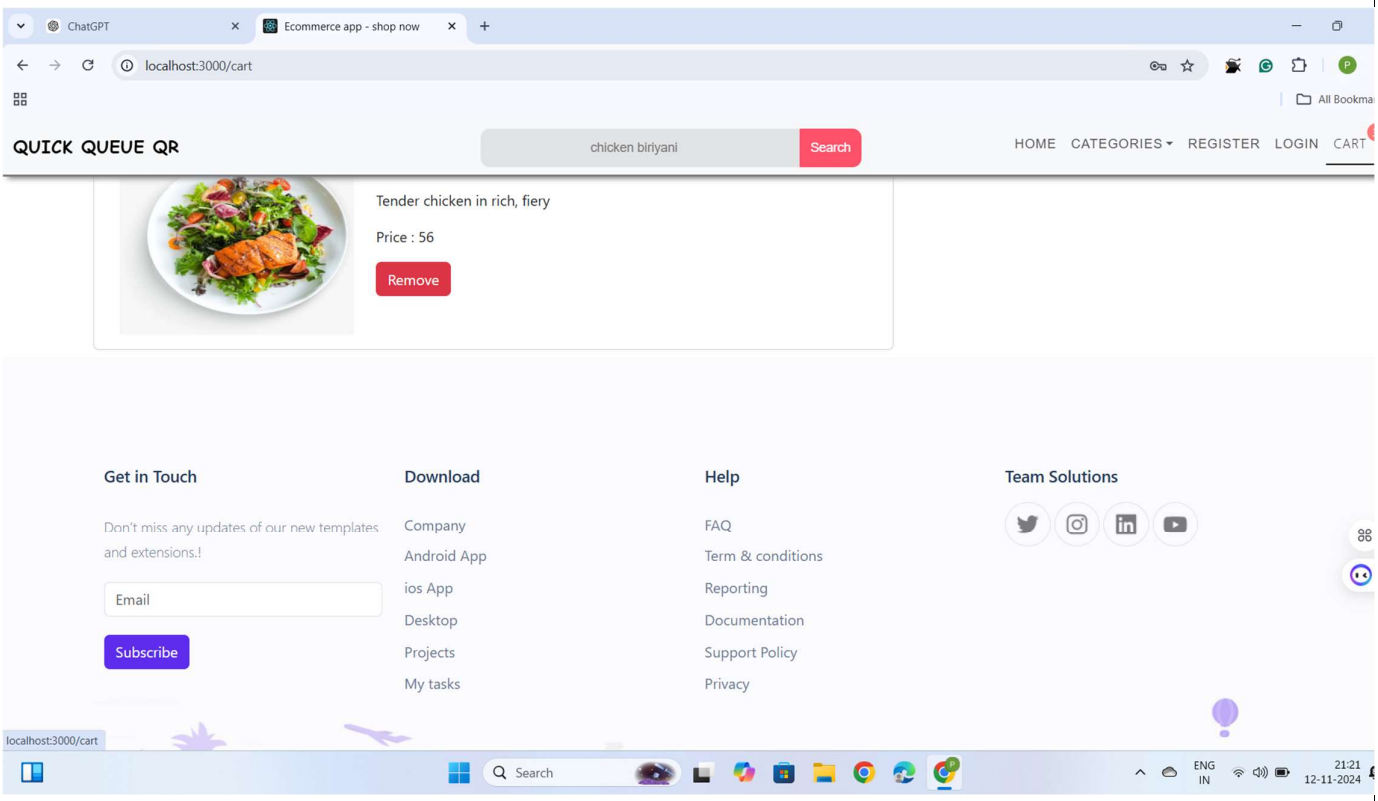
Second Picture :



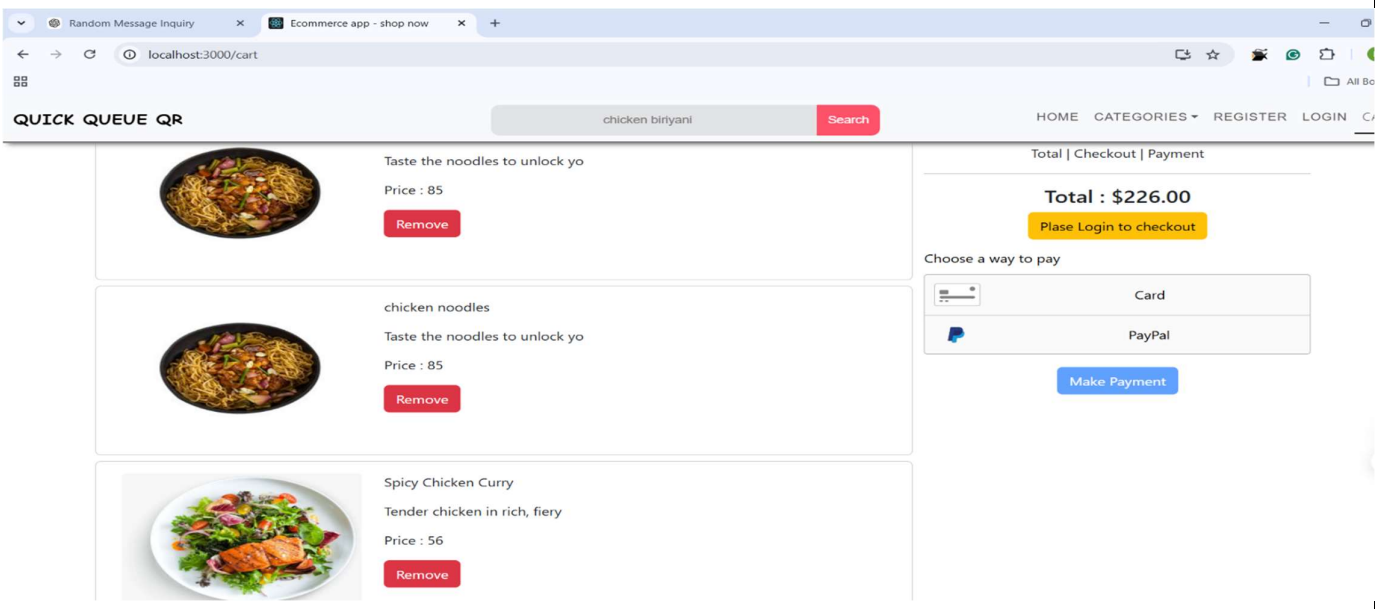
Third Picture Registration Form :



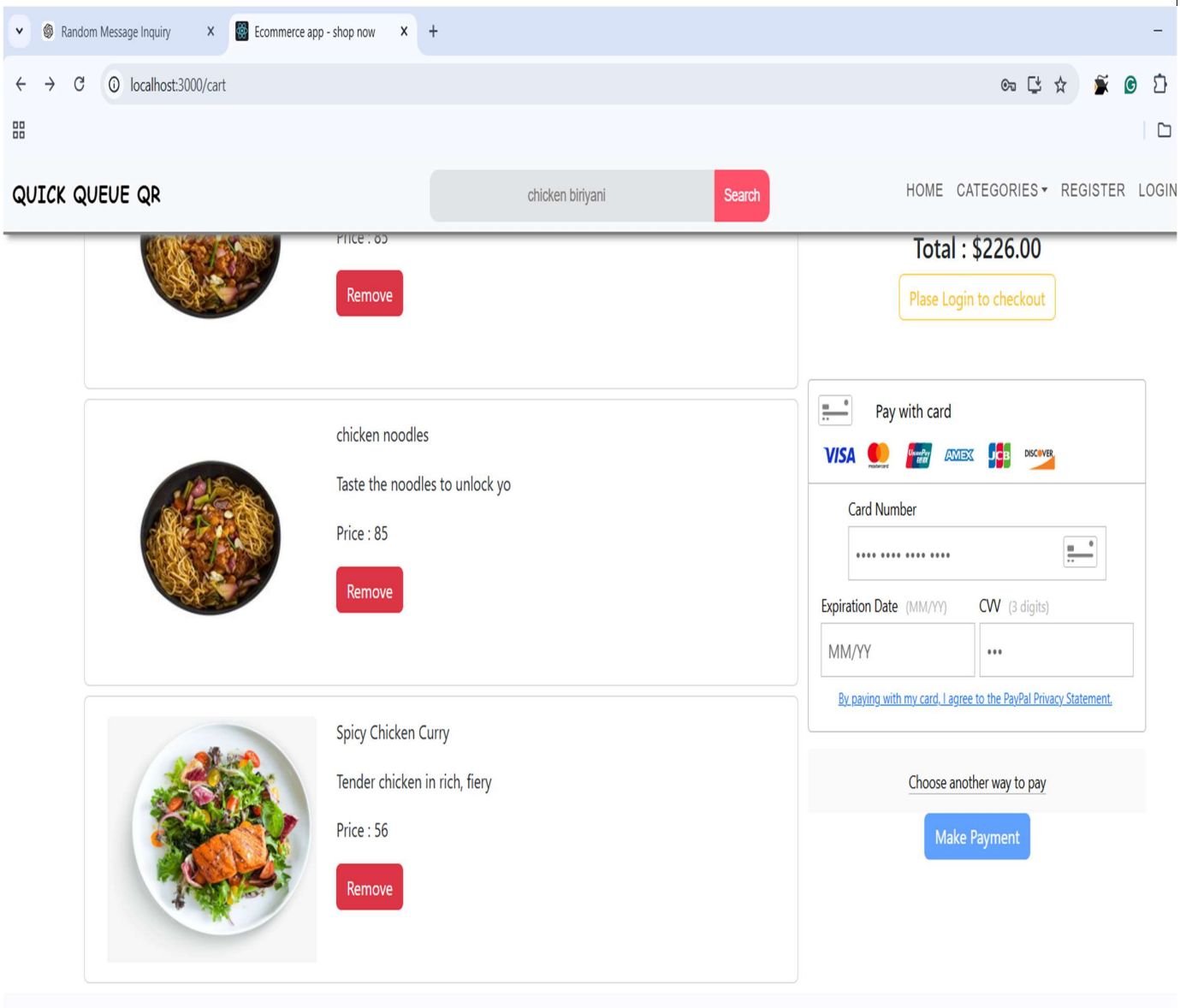
Forth Picture :



Fifth Picture Cart:



Sixth Picture Payments :



The demo link GitHub link https://github.com/PetaSravankumar/nmFoodOrderingApp/blob/main/demo%20video%20(1).mp4

The Google Driver link :https://drive.google.com/file/d/1WDh3QGFMEu7qGUYzaA85fh7nGXFGLNZh/view?usp=sharing

Known Issues:

Here are some known issues and bugs that users and developers should be aware of while using or developing the **SB Foods** web application:

1. **Image Upload Failure in Admin Dashboard**:
    - **Issue**: Occasionally, when restaurant owners try to upload images of dishes through the admin dashboard, the image upload fails without displaying an error message.
    - **Cause**: This might be due to a file size limitation on the backend or improper file handling.
    - **Workaround**: Until fixed, restaurant owners should try uploading smaller image files (under 2 MB). Developers can address this by adding proper error handling and validation for image uploads.
2. **Slow Performance During High Traffic**:
    - **Issue**: Under heavy traffic (e.g., many users placing orders simultaneously), the application can experience slow response times or delayed order processing.
    - **Cause**: The current server infrastructure may not be optimized for scaling, especially with the MongoDB queries taking longer under load.
    - **Workaround**: Implementing caching mechanisms (e.g., Redis) and optimizing database queries can mitigate this issue. Developers should also consider load balancing for better handling of traffic spikes.
3. **Cart Not Updating in Real-Time**:
    - **Issue**: Changes to the cart (such as adding or removing items) do not always reflect immediately on the frontend.
    - **Cause**: This may be due to a lack of real-time synchronization between the frontend and backend or issues with state management.
    - **Workaround**: Users can refresh the page to see the latest updates to the cart. Developers should investigate using WebSockets or polling to ensure real-time updates.
4. **Mobile Responsiveness on Checkout Page**:

- **Issue**: On smaller mobile screens, some elements of the checkout page (like the "Place Order" button) may overlap or become inaccessible.
- **Cause**: CSS styling may not be fully optimized for mobile responsiveness in some components.
- **Workaround**: Developers should test the layout on various mobile devices and adjust the CSS for better responsiveness, particularly for smaller screen sizes.

5. **Database Timeout on Large Orders**:
   - **Issue**: When placing very large orders with many items, the system occasionally times out, causing the order to fail.
   - **Cause**: The backend may not be efficiently handling large data submissions.
   - **Workaround**: Developers should optimize the order processing logic to handle bulk items more efficiently and increase the timeout limits where necessary.
6. **Delayed Email Notifications**:
   - **Issue**: Users and restaurant owners sometimes experience delays in receiving email notifications for order confirmations and updates.
   - **Cause**: The current email-sending process might be queued or facing latency issues.
   - **Workaround**: Developers should explore integrating a more robust email service (e.g., SendGrid or Amazon SES) with proper error handling and retry mechanisms.
7. **Logout Button Unresponsive After Session Timeout**:
   - **Issue**: After a session timeout, the logout button occasionally becomes unresponsive, and the user must manually refresh the page to log out.
   - **Cause**: This could be due to a frontend state management issue where the session expiration is not properly handled.
   - **Workaround**: Developers should ensure the session expiration logic triggers a forced logout and displays a proper message to the user.
8. **Real-Time Order Tracking Glitches**:
   - **Issue**: The real-time order tracking feature sometimes shows incorrect or outdated status updates for users.

- **Cause**: There might be synchronization issues between the backend order status and the frontend tracking system.
- **Workaround**: Developers should check the WebSocket implementation and ensure accurate status updates are pushed in real-time.

Future Enhancements:

1. AI-Powered Dish Recommendation System

- **Description**: An AI-based recommendation engine that suggests dishes based on users' order history, preferences, and reviews. This offers personalized meal suggestions to users.
- **Benefit**: Increases user engagement and order frequency by offering tailored options, improving the overall user experience.

2. Multilingual Support

- **Description**: Support for multiple languages, allowing users to navigate the app in their preferred language.
- **Benefit**: Expands the app's reach to non-English-speaking regions, attracting a broader user base and improving accessibility.

3. Real-Time Delivery Tracking with GPS Integration

- **Description**: GPS-based real-time tracking of delivery drivers, providing users with accurate delivery time estimates and live updates on their orders.
- **Benefit**: Enhances user satisfaction through transparency and reliable delivery updates, building trust in the service.

4. Loyalty Program and Discount Features

- **Description**: A rewards system where customers earn points for each order, which can be redeemed for discounts or special promotions.
- **Benefit**: Encourages repeat business, improves customer retention, and increases customer lifetime value.

5. Advanced Analytics for Restaurant Owners

- **Description**: Detailed analytics tools for restaurant owners, including sales data, popular dishes, and customer trends, presented via a user-friendly dashboard.
- **Benefit**: Enables restaurant owners to make data-driven decisions, optimize operations, and improve inventory management.

6. Mobile App Development

- **Description**: A native mobile app for iOS and Android that complements the web version, allowing users to browse, order, and track deliveries from their smartphones.
- **Benefit**: Increases accessibility and convenience, leading to higher customer engagement and retention.

7. Customizable Dietary Preferences

- **Description**: A feature that allows users to filter menu options based on dietary preferences (e.g., vegan, gluten-free, low-carb).
- **Benefit**: Improves the user experience by providing personalized food options, making the app more inclusive for users with dietary restrictions.

THANK YOU