**FULL STACK DEVELOPMENT WITH MERN**

**MERN STACK POWERED BY MONGODB**

**PROJECT:**

**QUICKBITE**

**FOOD ORDERING APPLICATION**

**Submitted by:**

**ABHINAYA B**

**RAMYA SIVAKUMAR**

**REETA A V**

**SHAKTHI KEERTHANA M**

**1. INTRODUCTION**

* **Project Title:** Food Ordering Application - QuickBite
* **Team Members:**

Abhinaya B (311521104002)

Ramya Sivakumar (311521104036)

Reeta A V (311521104038)

Shakthi Keerthana M (311521104702)

**2. PROJECT OVERVIEW**

**Objective:**The Food Ordering Application - QuickBite is a full-stack web platform designed to provide a seamless experience for customers to browse restaurant menus, place orders, and receive real-time updates on their order status. Built using the MERN stack (MongoDB, Express.js, React.js, and Node.js), the application focuses on creating an intuitive and responsive interface with robust backend functionality and secure data management.

**Technology Stack:**

Frontend: React.js for building a modern, responsive user interface that ensures smooth navigation and interaction across various devices.

Backend: Node.js and Express.js, used to develop a RESTful API that supports functionalities like user authentication, order processing, and payment handling.

Database: MongoDB, chosen for its NoSQL structure and flexibility, ideal for managing dynamic data such as menus, user profiles, and order histories.

**Key Features:**

User-Friendly Interface: An easy-to-navigate UI where customers can view restaurant menus, filter items, add items to their cart, and proceed to checkout.

Real-Time Updates: Integrated with Socket.IO for real-time notifications, allowing customers to receive live updates on their order status.

Secure Authentication: JWT (JSON Web Tokens) and bcrypt hashing ensure secure user login and data protection.

**Challenges & Solutions:**

Adaptable Data Management: MongoDB’s document-based schema was used to handle complex data structures efficiently.

Security: JWT and bcrypt provided a secure authentication flow, essential for user trust and data integrity.

Real-Time Functionality: Socket.IO was implemented to deliver immediate order status updates to users, enhancing the user experience.

**Conclusion:**

The QuickBite Food Ordering Application successfully combines modern web technologies to deliver a scalable, secure, and interactive platform for online food ordering. The use of the MERN stack has proven to be an effective choice in building a dynamic, feature-rich application that meets the needs of both customers and restaurant owners.

**Purpose:**

The purpose of the QuickBite Food Ordering Application project is to develop a modern, efficient, and scalable platform that facilitates a seamless online food ordering experience for customers while providing restaurant owners with an easy-to-manage, integrated system. The project aims to address common challenges in online ordering, such as complex interfaces, slow response times, and integration issues, by leveraging the MERN stack (MongoDB, Express.js, React.js, and Node.js) to create a cohesive, high-performance application.

The key goals of the project include:

**Enhancing User Experience:** Delivering an intuitive, responsive, and user-friendly interface that allows customers to browse menus, filter items, add items to the cart, and place orders easily.

**Real-Time Order Updates:** Providing live order status updates through real-time notifications, allowing customers to track their orders accurately and improving overall engagement.

**Streamlining Restaurant Management:** Simplifying the management of menu items, orders, and customer data for restaurant owners through a secure and scalable backend system.

**Ensuring Data Security and Reliability:** Implementing secure user authentication, data management, and payment processing mechanisms to protect customer information and build trust.

Achieving Scalability: Utilizing a scalable architecture that supports high traffic and large data volumes, making it adaptable to the needs of various restaurant businesses.

Overall, the project aims to create a robust and adaptable food ordering application that enhances convenience for customers and efficiency for restaurant operations.

**Features:**

### 1. User Account Management

**Sign Up / Login**: Allow users to create accounts or sign in through email, phone number, or social media platforms.

**Profile Management**: Users can update their personal information, payment methods, and order history.

**Order History**: Displays past orders, allowing users to reorder easily.

### 2. Search and Discover

**Restaurant Listings:** List restaurants based on location, ratings, cuisine type, popularity, etc.

**Menu Browsing:** Detailed menu listings with item descriptions, prices, and availability.

**Filtering and Sorting:** Filter restaurants by ratings, delivery time, cuisines, and cost.

**Search Bar:** Keyword-based search to find specific dishes or restaurants.

### 3. Menu Customization and Ordering

### Customizable Options: Allow users to modify items (e.g., extra toppings, spice level).

### Add to Cart: Users can add items to a cart, modify quantities, or remove items before checkout.

### Special Instructions: Users can add special instructions for individual items.

### 4. Payment Options

**Multiple Payment Methods:** Accept credit/debit cards, digital wallets, UPI, and cash-on-delivery.

**Saved Cards:** Save card details securely for fast checkouts.

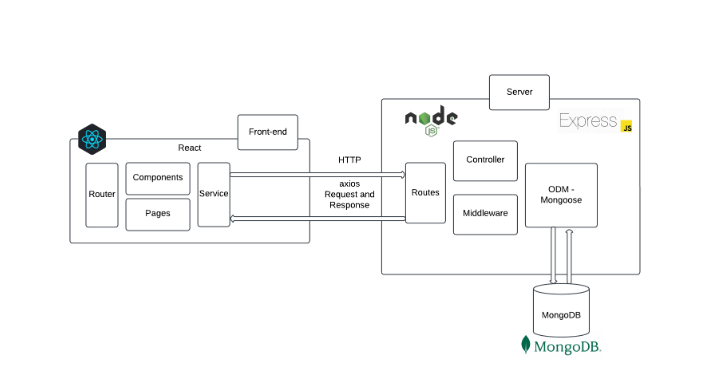
**Split Payment:** Allows group ordering and splitting of bills among friends.

### 5. Admin Panel for Restaurants

**Menu Management:** Add, remove, and update menu items and prices.

**Order Management:** View and manage incoming orders with real-time updates.

**Analytics Dashboard:** View insights on sales, popular items, customer feedback, etc.

**3. ARCHITECTURE:**

### Frontend:

The frontend remains a single-page application (SPA) built with React, emphasizing interactivity and smooth transitions without page reloads.

#### Component Structure:

* **Functional Components**: Continue using functional components for all UI elements (e.g., RestaurantList, Menu, Cart, OrderHistory, OrderTracking).
* **Component Breakdown**:
  + **RestaurantList**: Displays all available restaurants based on filters.
  + **Menu**: Displays the restaurant’s menu, supports adding/removing items to the cart.
  + **Cart**: Shows selected items, allows modifications, and triggers order confirmation.
  + **OrderHistory**: Lists past orders with the option to reorder.
  + **OrderTracking**: Real-time updates on order status, from preparation to delivery.

#### State Management:

* **Context API**: Context API with custom hooks and reducers handles global state for user sessions, cart items, and real-time order tracking data.
* **Session Storage for Authentication**: Store JWT tokens and user data in session storage to persist login status across tabs. Token verification occurs before each request to refresh tokens and maintain a secure session.
* **Order Status Updates**: Use context to manage real-time updates for the user's order status, making it easy to notify users of each stage (preparing, out for delivery, delivered).

#### Routing:

* **React Router**: Utilize react-router-dom for:
  + **Public Routes**: Home page, restaurant browsing, menu viewing.
  + **Protected Routes**: User-specific (OrderHistory, Profile, Cart) and admin-specific pages (AdminDashboard for order and menu management).
* **Dynamic Routing**: Use dynamic routing for menu pages based on restaurant selection, enhancing the seamless flow between browsing and ordering.

#### API Communication:

* **Axios**: Use Axios to communicate with the backend, attaching JWT tokens in the header for secure access to protected endpoints.
* **Centralized Error Handling**: Implement user-friendly error feedback for various scenarios like login failure, unavailable items, or payment issues.

#### Styling and UI:

* **Styled Components**: Continue with styled components or inline styles to maintain modular and scalable CSS.
* **Responsive Design**: CSS media queries for responsive layouts, especially for key components like restaurant and menu listings, cart, and order tracking.

### Backend:

The backend is structured with Node.js and Express.js to provide a RESTful API handling user authentication, restaurant/menu data, order processing, and delivery tracking.

#### Server Structure:

* **Express Server**: The application runs on Express, initializing middleware, routes, and database connections.
* **Environment Configuration**: dotenv is used to manage environment variables like database URI, JWT secret, and API keys.

#### API Design:

* **RESTful Endpoints**:
  + **Authentication**: Endpoints for registration, login, and token refresh.
  + **Restaurant Management**: CRUD operations for restaurant listings.
  + **Menu Management**: CRUD for menu items within each restaurant.
  + **Order Management**: Endpoints for creating, updating, tracking, and retrieving user orders.
  + **Cart Management**: Adding, removing, and updating items in the user’s cart.
* **Route Modules**: Modularized routing with specific files for each entity (authRoutes, restaurantRoutes, menuRoutes, orderRoutes, cartRoutes).

#### Middlewares:

* **JWT Middleware**: Validates tokens on protected routes, ensuring only authenticated users can access restricted endpoints.
* **Centralized Error Handling**: Handles all server errors consistently, logging them for troubleshooting.
* **express-validator**: Used to validate input for user registration, login, and order creation (e.g., required fields, email format, valid product IDs).

#### Authentication and Authorization:

* **JSON Web Tokens (JWT)**: Tokens are generated on login, stored on the client side, and verified on each request for secure access to protected routes.
* **bcrypt**: Passwords are hashed with bcrypt for secure storage in the database.
* **Role-Based Access Control**:
  + **Customer Role**: Access to browse restaurants, view menu items, place orders, and track order status.
  + **Admin Role**: Additional access to manage restaurants, menus, and view all customer orders.

#### Session and Token Handling:

* **Token Expiry**: Tokens are configured with an expiry time, requiring users to log in again after expiration.
* **Session Storage in Frontend**: The frontend stores JWT tokens, user ID, and user type in session storage, maintaining the user’s session across tabs.
* **Refresh Tokens**: Implement refresh tokens to avoid re-authentication after token expiration, allowing users to maintain longer sessions.

**Database:**

The database uses MongoDB with a schema defined through Mongoose. Collections are created to manage restaurants, menu items, orders, and user-specific elements like cart and order history.

**Database Structure:**

### Database Structure:

#### i) User Schema:

* **Fields**:
  + username: Unique identifier for each user.
  + password: Hashed password using bcrypt.
  + email: Unique email address for contact and verification.
  + userType: Specifies user role (e.g., "customer", "admin").
  + address: Stores saved addresses for easier checkout (could be an array to store multiple addresses).
  + phone: Contact number for order updates.
* **Indexes**: Unique indexes on username and email to prevent duplicates.
* **Relationships**: User IDs are referenced in the Order, Cart, and Favorites collections.

#### ii) Restaurant Schema:

* **Fields**:
  + name: Name of the restaurant.
  + location: Physical location and geolocation (latitude and longitude) for accurate delivery zones.
  + cuisineType: Type of cuisine offered (e.g., Italian, Chinese, Indian).
  + rating: Average user rating for the restaurant.
  + openHours: Operating hours for each day.
  + menu: Reference to an array of menu items related to the restaurant.
* **Indexes**: Indexed by name, location, and cuisineType for efficient searching and filtering.

#### iii) Menu Item Schema:

* **Fields**:
  + restaurantId: References the Restaurant collection.
  + name: Name of the menu item.
  + description: Description of the dish.
  + price: Price of the item.
  + category: Category within the menu (e.g., Appetizers, Main Course, Dessert).
  + image: Image URL for the item.
  + availability: Boolean to indicate if the item is currently available.
  + tags: Array of tags (e.g., vegetarian, spicy) for filtering.
* **Indexes**: Indexed by name and category to allow fast searching and filtering within a restaurant’s menu.
* **Relationships**: Each menu item references a restaurantId to link it back to a specific restaurant.

#### iv) Order Schema:

* **Fields**:
  + userId: References the User collection.
  + restaurantId: References the Restaurant collection for order origin.
  + items: Array of items ordered, each containing:
    - menuItemId: Reference to a Menu Item.
    - quantity: Quantity ordered for the item.
    - price: Price at the time of order to avoid discrepancies if menu prices change.
  + totalPrice: Total price of the order.
  + address: Delivery address (pulled from user’s saved addresses).
  + orderDate: Timestamp of when the order was placed.
  + estimatedDeliveryTime: Predicted time of delivery based on restaurant and location.
  + deliveryStatus: Tracks order progress (e.g., "pending", "preparing", "out for delivery", "delivered").
  + paymentMethod: Specifies method used (e.g., credit card, cash on delivery).
* **Relationships**: userId references the User collection, restaurantId references Restaurant, and each item references a Menu Item.

#### v) Cart Schema:

* **Fields**:
  + userId: References the User collection.
  + restaurantId: Ensures that the cart is consistent within a single restaurant’s menu at a time.
  + items: Array of items in the cart, each containing:
    - menuItemId: Reference to a Menu Item.
    - quantity: Quantity of each item.
    - price: Price per item for accurate total calculation.
* **Relationships**: Each cart entry references userId and restaurantId, linking it to the user and specific restaurant.
* **Constraints**: Enforce uniqueness on userId and restaurantId combination to avoid conflicts when a user selects items from a different restaurant.

#### vi) Favorites Schema:

* **Fields**:
  + userId: References the User collection.
  + menuItemId: References a Menu Item for quick reordering.
* **Relationships**: Both userId and menuItemId serve as foreign keys referencing the User and Menu Item collections.
* **Constraints**: Enforce uniqueness on userId and menuItemId combinations to avoid duplicate favorites.

**Database Operations:**

The database layer of the QuickBite Food ordering application is powered by MongoDB and is designed to perform efficient data operations specifically tailored for users, restaurants, menu items, orders, carts, and favorites. Typical operations within the database include the standard CRUD functions—Create, Read, Update, and Delete—along with specialized actions that address the unique requirements of food ordering systems.

**4. SETUP INSTRUCTIONS**

To set up the Food Ordering application, follow these steps:

**1. Install Prerequisites:**

* **Node.js (Version 20.x):** Download and install from Node.js official website, which includes npm (Node Package Manager).
* **MongoDB:** Set up MongoDB using MongoDB Compass for a local instance or create a MongoDB Atlas account for a cloud-based setup.
* **Git:** Install Git from Git downloads for version control.
* **npm -** Node Package Manager (comes with Node.js).
* **Code Editor -** Visual Studio Code or another preferred IDE.

**2. Download Project Files:**

Place all project files in a dedicated project directory on the local machine.

**3. Install Dependencies:**

Open a terminal, navigate to the project directory, and install the required dependencies:

*npm install*

**4. Set Up Environment Variables:**

In the project root, create a .env file to store environment-specific variables such as database connection strings and JWT secrets. Contents in the .env file is as follows:

PORT=3001

MONGO\_URI=<mongodb-connection-string>

JWT\_SECRET=<jwt-secret>

**5. Run the Application:**

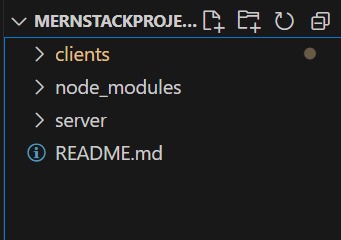
Start the application in development mode:

*npm start*

Now the application can be accessed locally at http://localhost:3000.

**5. FOLDER STRUCTURE**

The QuickBite Food Ordering application follows a well-organized folder structure for both the **Frontend** (React) and **Backend** (Node.js) to ensure clarity, scalability, and maintainability of the project.



*Figure .Overall Folder Structure*

**Client: React Frontend Structure**

The frontend of QuickBite is built using **React**, and its folder structure is organized as follows:

* **public/**:
  + Contains static assets and other public files that are served directly to the browser.
* **src/**:
  + This is where all the source code of the React application resides.
  + **components/**: Contains reusable components. These components are used throughout the application.
  + **pages/**: Contains React components that represent different pages in the application
  + **context/**: Handles the global state management for the app using React Context API. This includes user authentication and cart state management.
  + **styles/**: Contains custom CSS files to style the components and pages, ensuring the application's design is modular and maintainable.
  + **images/**: Stores image assets like food images, and icons used throughout the application, centralizing all visual elements for easy management.
  + **App.js**: The root component where routing and major layout structures are defined.
  + **index.js**: Entry point of the application where ReactDOM renders the app into the DOM.
* **package.json**:
  + This file manages project dependencies, scripts, and other configurations for the frontend application.

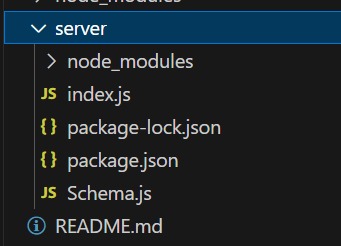


*Figure .Client Structure*

**Server: Node.js Backend Structure**

The backend of QuickBite is built using **Node.js** and follows a simple but effective folder structure for API development:

* **models/**:
  + Contains the Mongoose models for interacting with the MongoDB database. Define the schema for respective entities in the database.
* **routes/**:
  + Houses route files where the API endpoints are defined.
* **index.js**:
  + The entry point of the backend application where the Express server is set up, middleware is applied, and routes are initialized.
* **package.json**:
  + This file manages backend dependencies and includes scripts for running the server.



*Figure .Server Structure*

**6. RUNNING THE APPLICATION**

To run the Food Ordering application - QuickBite locally, the following steps are done for starting both the frontend and backend servers:

**Frontend:**

**1. Navigate to the Client Directory:**

Open a terminal window on your system.

Use the cd (change directory) command to navigate to the client folder, which contains all the frontend code for the application.

**2. Install Frontend Dependencies:**

Before starting the frontend server, ensure that all required dependencies are installed. If this is the first time running the project or if dependencies have been added or updated, run the following command to install them:

npm install

This command will install all the necessary packages listed in the package.json file, including React and other dependencies.

**3. Start the Frontend Server:**

After installing the dependencies, use the following command to start the React development server:

npm start

This will compile the React application and start a local development server.

**4. Access the Application:**

* Once the server is running, a web browser is opened and goes to http://localhost:3001.
* The frontend application will be accessible at this URL, and you should see the Food Ordering Application user interface in your browser. The frontend is now live and connected to the backend server for full functionality.

**Backend:**

**1. Navigate to the Server Directory:**

Open a terminal window.

Use the cd (change directory) command to navigate to the server folder, which contains all the backend code for the application.

**2. Install Backend Dependencies:**

If this is the first time running the backend server, ensure all required dependencies are installed by running:

npm install

This will install all the necessary backend packages, including Express, Mongoose, and other dependencies as specified in the package.json file.

**3. Start the Backend Server:**

Once the dependencies are installed, use the following command to start the backend server:

node index.js

This will initialize the backend server and connect it to the database.

**4. Access the Backend:**

* The backend server will now be running and can be accessed via http://localhost:3001.
* This URL is used to handle all API requests, and the backend will process interactions such as user authentication, product management, and order processing.

**7. API DOCUMENTATION**

The QuickBite backend exposes various endpoints to handle operations related to users, orders, menus, and authentication.

### 1. Authentication

**POST** /api/auth/register

* **Description**: Registers a new user.

**Request Body**:  
json  
Copy code  
{

"username": "john\_doe",

"email": "johndoe@gmail.com",

"password": "password123",

"usertype": "customer"

}

* **Response**:
  + **200 OK**: User registration successful.
  + **400 Bad Request**: Invalid or missing parameters.

**POST** /api/auth/login

* **Description**: Logs in a user and generates a JWT token for authorization.

**Request Body**:  
json  
Copy code  
{

"email": "johndoe@gmail.com",

"password": "password123"

}

* **Response**:
  + **200 OK**: Returns a JWT token.
  + **401 Unauthorized**: Invalid credentials.

**POST** /api/auth/forgot-password

* **Description**: Sends a password reset link to the user's email.

**Request Body**:  
json  
Copy code  
{

"email": "johndoe@gmail.com"

}

* **Response**:
  + **200 OK**: Password reset link sent.
  + **400 Bad Request**: Email not found.

### 2. User Management

**GET** /api/users

* **Description**: Retrieves a list of all users in the system.
* **Response**:
  + **200 OK**: Returns an array of all users.

json  
Copy code  
[

{

"userId": "user123",

"username": "john\_doe",

"email": "johndoe@gmail.com",

"usertype": "customer"

},

{

"userId": "user456",

"username": "jane\_doe",

"email": "janedoe@gmail.com",

"usertype": "admin"

}

]

### 3. Menu Management

**GET** /api/menu

* **Description**: Retrieves a list of available food items with optional filtering by category and price range.
* **Query Parameters**: category, priceMin, priceMax
* **Response**:
  + **200 OK**: Returns an array of food items.

json  
Copy code  
[

{

"itemId": "item123",

"name": "Margherita Pizza",

"description": "Classic cheese pizza with fresh tomato sauce",

"price": 500,

"category": "Pizza",

"image": "<img-url>"

}

]

**GET** /api/menu/{id}

* **Description**: Retrieves details of a specific menu item by its ID.
* **Path Parameter**: id
* **Response**:
  + **200 OK**: Returns detailed information of the menu item.

json  
Copy code  
{

"itemId": "item123",

"name": "Margherita Pizza",

"description": "Classic cheese pizza with fresh tomato sauce",

"price": 500,

"category": "Pizza",

"image": "<img-url>"

}

### 4. Cart Management

**POST** /api/cart

* **Description**: Adds a food item to the user's cart.

**Request Body**:  
json  
Copy code  
{

"userId": "user123",

"itemId": "item123",

"quantity": 2

}

* **Response**:
  + **200 OK**: Successfully added item to cart.

json  
Copy code  
{

"message": "Item added to cart",

"cartItem": {

"itemId": "item123",

"quantity": 2

}

}

**GET** /api/cart/{userId}

* **Description**: Retrieves all items in the user's cart.
* **Path Parameter**: userId
* **Response**:
  + **200 OK**: Returns a list of cart items.

json  
Copy code  
[

{

"itemId": "item123",

"quantity": 1,

"price": 500

}

]

### 5. Order Management

**POST** /api/orders

* **Description**: Places a new order.

**Request Body**:  
json  
Copy code  
{

"userId": "user123",

"cartItems": [

{

"itemId": "item123",

"quantity": 2

}

],

"address": "123 Food St, Food City",

"pincode": "123456",

"paymentMethod": "Credit Card"

}

* **Response**:
  + **200 OK**: Order placed successfully.

json  
Copy code  
{

"message": "Order placed successfully",

"order": {

"orderId": "order123",

"status": "Order placed"

}

}

**GET** /api/orders/{userId}

* **Description**: Retrieves all orders for a specific user.
* **Path Parameter**: userId
* **Response**:
  + **200 OK**: Returns a list of orders.

json  
Copy code  
[

{

"orderId": "order123",

"status": "Order placed",

"totalPrice": 1000

}

]

### 6. Wishlist Management

**POST** /api/wishlist

* **Description**: Adds a food item to the user's wishlist.

**Request Body**:  
json  
Copy code  
{

"userId": "user123",

"itemId": "item456"

}

* **Response**:
  + **200 OK**: Successfully added item to wishlist.

json  
Copy code  
{

"message": "Item added to wishlist",

"wishlistItem": {

"itemId": "item456"

}

}

**GET** /api/wishlist/{userId}

* **Description**: Retrieves all items in the user's wishlist.
* **Path Parameter**: userId
* **Response**:
  + **200 OK**: Returns a list of wishlist items.

json  
Copy code  
[

{

"itemId": "item456",

"name": "Garlic Bread",

"price": 150

}

]

### 7. Search

**GET** /api/search

* **Description**: Searches for food items based on a query.
* **Query Parameters**: query
* **Response**:
  + **200 OK**: Returns the search results.

json  
Copy code  
[

{

"itemId": "item789",

"name": "Spaghetti Bolognese",

"price": 400

}

]

### 8. Admin Routes

**GET** /api/admin/users

* **Description**: Retrieves all users in the system (admin-only).
* **Response**:
  + **200 OK**: Returns a list of all users.

json  
Copy code  
[

{

"userId": "user123",

"username": "john\_doe",

"email": "johndoe@gmail.com"

}

]

**POST** /api/admin/menu

* **Description**: Adds a new food item (admin-only).

**Request Body**:  
json  
Copy code  
{

"name": "Paneer Tikka",

"price": 350,

"category": "Starters",

"image": "<img-url>"

}

* **Response**:
  + **200 OK**: Food item added successfully.

json  
Copy code  
{

"message": "Food item added successfully",

"item": {

"name": "Paneer Tikka",

"price": 350

}

}

**8. AUTHENTICATION**

The authentication and authorization process in the Food Ordering application FoodieGo is designed to securely verify user identity and manage access control using JSON Web Tokens (JWT), session storage, and password hashing.

1. **User Authentication with JWT Tokens**:

* Upon successful login or registration, a JSON Web Token (JWT) is generated and returned to the client. The token is encoded with user-specific details and a secret key to ensure authenticity.
* This token is stored in the client’s session storage and is included in headers for all protected requests, allowing the server to verify user identity and permissions.

1. **Token-Based Authorization**:

* JWT tokens are validated on each request to ensure that only authenticated users can access restricted features. For instance, customers have access to profile and shopping pages, while admin users have access to the Admin Dashboard.
* The token includes encoded user information (such as user ID and role), enabling role-based authorization and ensuring access control across different parts of the application.

1. **Password Hashing**:

* Passwords are securely hashed with bcrypt before being stored in the database, adding an essential layer of security to protect user credentials.
* During login, passwords are rehashed and verified against the stored hash to authenticate users.

1. **Session Management**:

* The application uses session storage for managing active sessions across different browser tabs, allowing users to remain logged in until token expiry. User details like user ID, name, email, and user type are saved in session storage, supporting consistent user experience throughout their session.
* Token expiry is also handled to ensure that sessions automatically log out after a certain period, enhancing security by preventing unauthorized access.

**9. USER INTERFACE**

The **User Interface (UI)** of the QuickBite Food Ordering application is designed to provide an intuitive and engaging experience for both customers and administrators. Built using **React**, the UI ensures responsiveness, smooth interactions, and a user-friendly shopping environment. It is structured to guide users effortlessly from browsing foods, restaurants to completing orders while providing admins with comprehensive control over the application’s functionalities.

### 1. Customer Experience

The **QuickBite** UI prioritizes an intuitive and seamless experience for users ordering food. The homepage features a clean design with a banner highlighting popular restaurants and dishes, a search bar, and category buttons for different cuisines. Users can easily browse restaurants and dishes displayed in visually appealing cards with details such as restaurant names, ratings, estimated delivery times, and prices. Each restaurant page provides more specific menu information, including item descriptions, sizes, and customization options (such as spice level or additional ingredients). Users can add items directly to their cart or save favorites for future orders with the wishlist feature. In the cart, customers can adjust quantities, remove items, and see the estimated delivery time and total cost before proceeding to checkout. The profile page allows users to manage personal information, view past orders, reorder for a streamlined, convenient experience.

### 2. Restaurant Partner Interface

The **Restaurant Partner Dashboard** provides an organized and accessible interface for managing orders, menu items, and restaurant details. Partners can access sections to update menu items (add, edit, or remove dishes), adjust pricing, and set item availability. Order management tools allow partners to view incoming orders, change order statuses (Received, In-Process, Out for Delivery, Delivered), and monitor customer preferences through analytics on popular items. The dashboard also supports inventory tracking, helping partners ensure sufficient stock for high-demand items. A detailed form lets partners add new menu items, with fields for dish names, descriptions, prices, and images, ensuring they have full control over their restaurant's presentation on the platform.

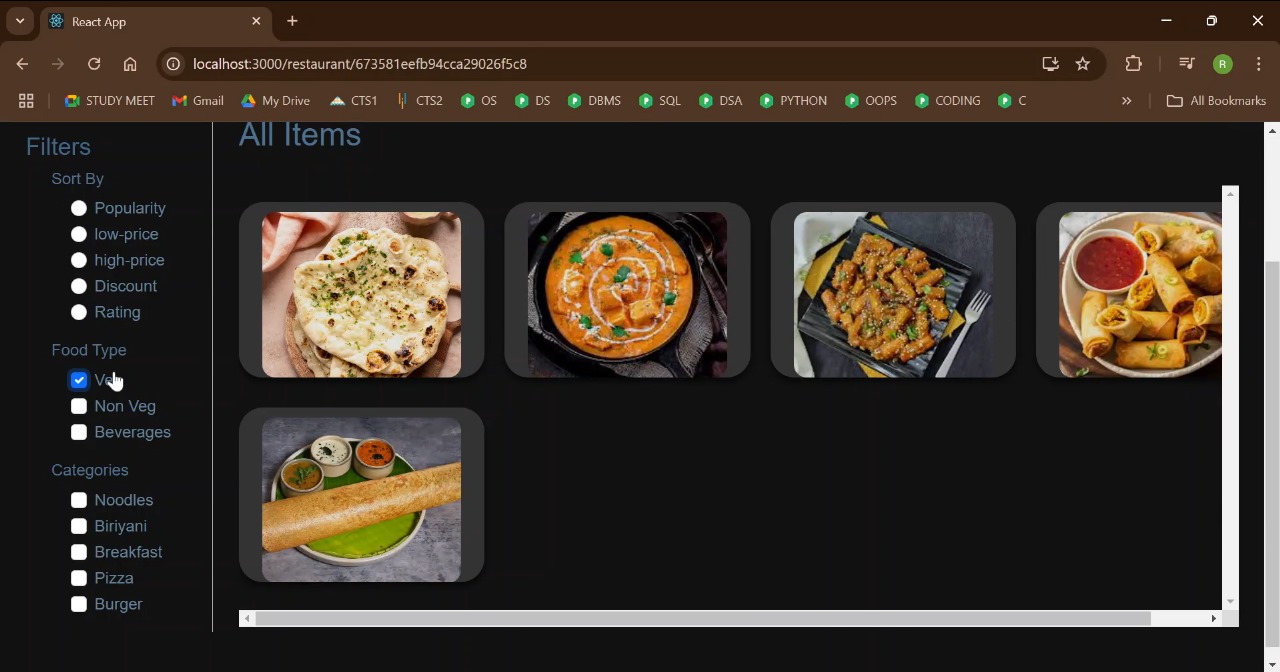
### 3. Admin Interface

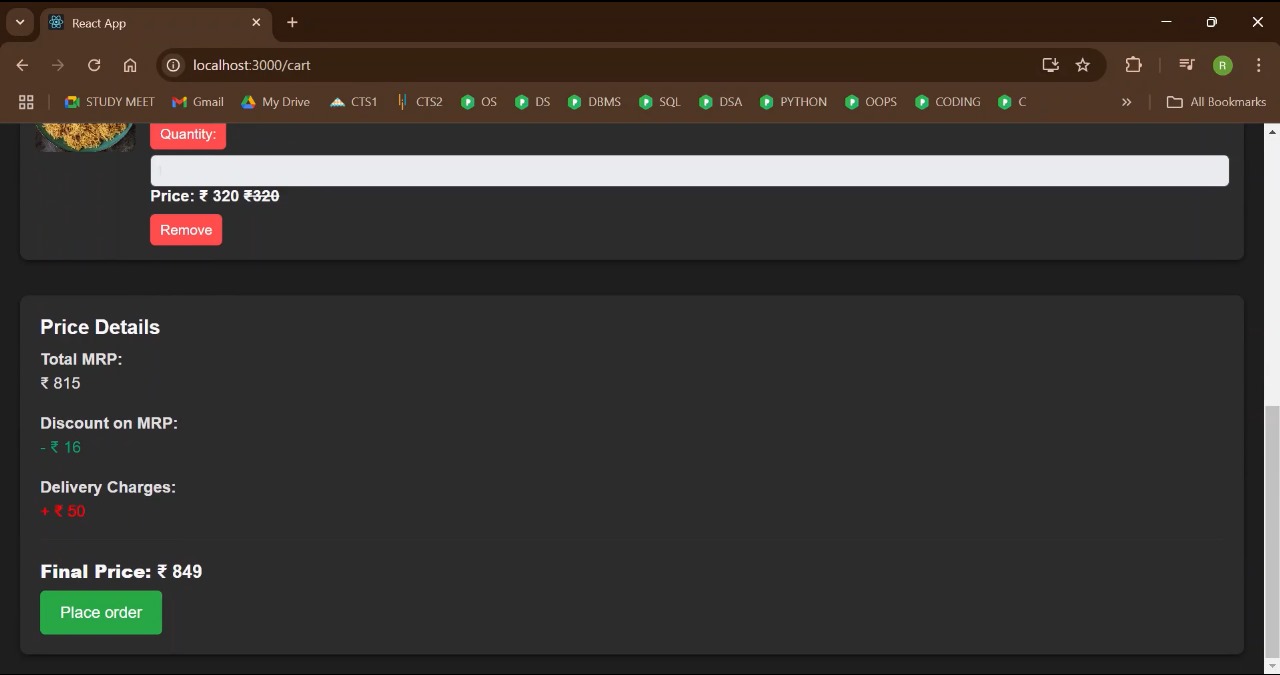
The **Admin Dashboard** offers comprehensive control over the platform’s operations. Admins have access to manage users, restaurants, and promotional content, and they can view and oversee order processes across the platform. The product management section allows admins to verify restaurant menu items and make adjustments if necessary. Additionally, admins can handle customer and restaurant partner inquiries, monitor and resolve reported issues, and manage order statuses across multiple restaurants. The dashboard provides insights into overall performance, allowing admins to track metrics such as user engagement, order volumes, and top-performing restaurants, helping them make data-driven decisions.

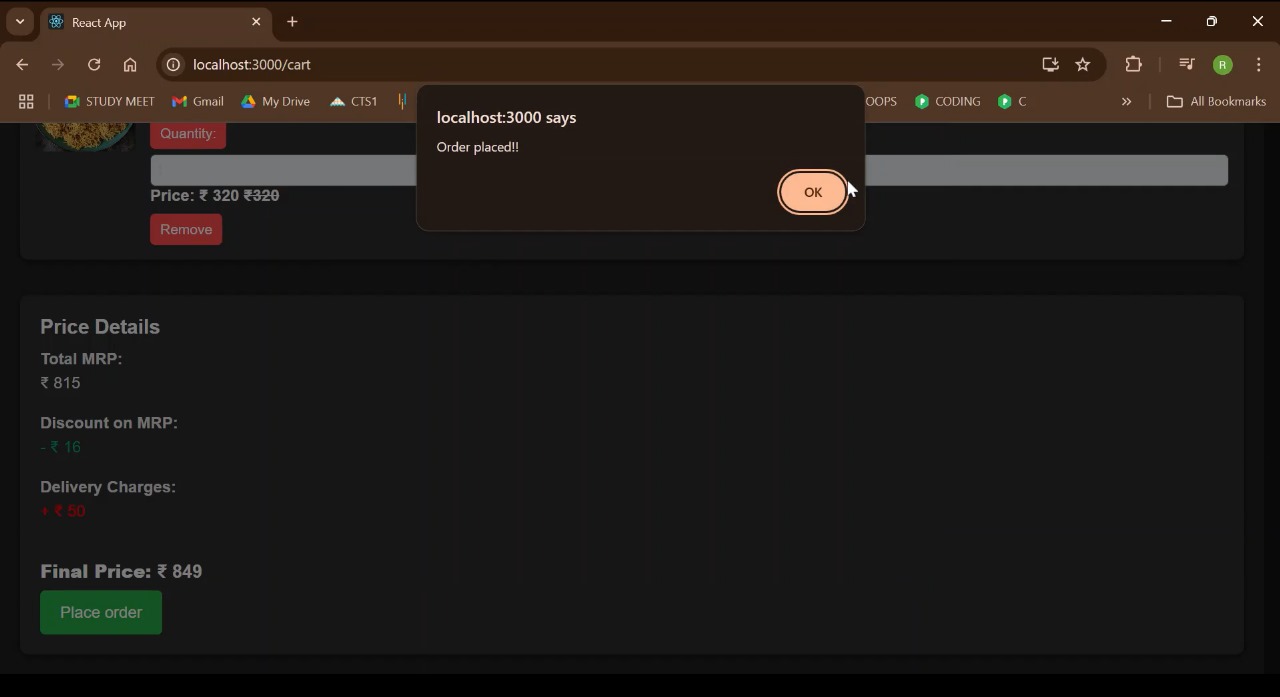
### 4. Responsiveness and Aesthetics

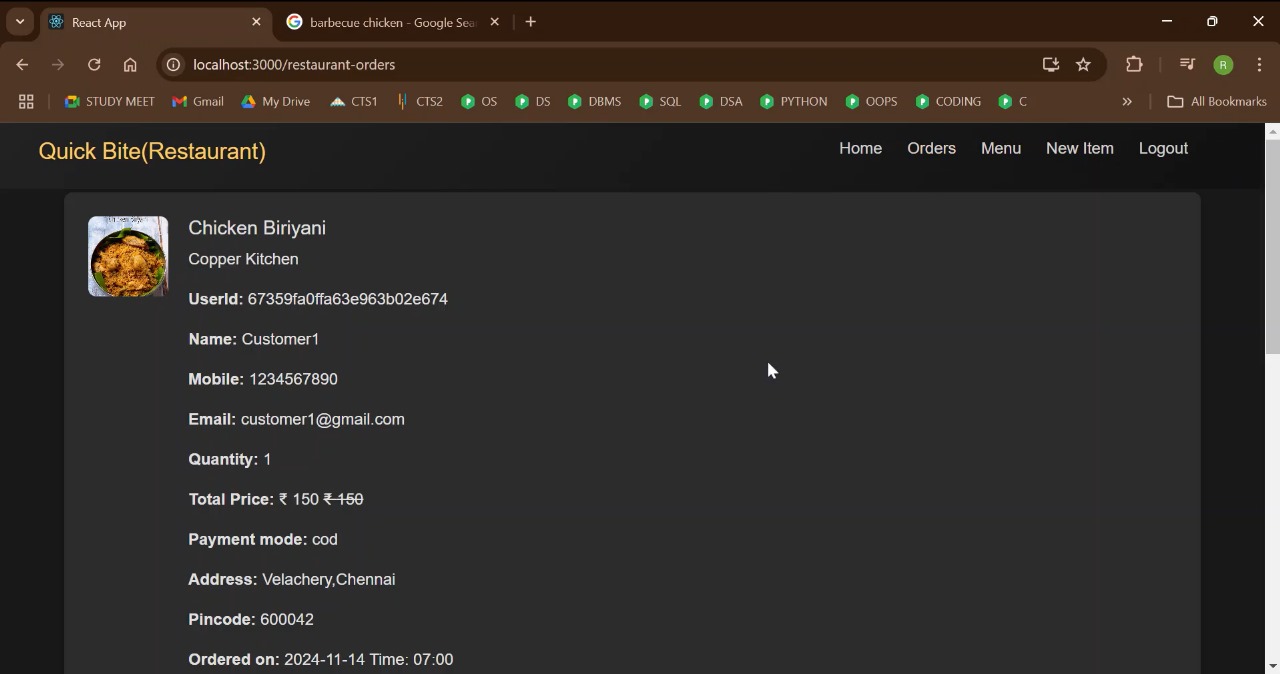
The UI is fully responsive, adapting smoothly across all devices, from mobile phones to large desktop screens. Built with Bootstrap and custom CSS, the design maintains consistency in both appearance and functionality across different screen sizes, ensuring users can easily place orders on any device. The aesthetic focus is on a clean, modern look, with vibrant visuals of food items and a user-friendly color palette, making the browsing and ordering experience both appealing and functional.

**Screenshots:**









**10. TESTING**

The testing strategy for the FoodieGo e-commerce application focuses on ensuring the functionality, security, and performance of the application through API testing using Postman. Postman is utilized for testing the backend API endpoints and verifying the responses to ensure they meet expected behaviors.

### API Testing with Postman

Postman is utilized to test all API endpoints within the **QuickBite** food ordering application, including user authentication, restaurant and menu management, order processing, and cart functionalities. Each endpoint undergoes testing with a variety of requests (GET, POST, PUT, DELETE) and different parameters to ensure accurate and reliable responses.

* **Authentication Endpoints**: Endpoints for user login and registration are tested by sending login credentials, verifying correct token generation, and validating token-based access control for restricted routes.
* **Restaurant and Menu Management Endpoints**: API tests include adding, updating, and retrieving restaurant information and menu items. This ensures smooth data flow, correct structure, and reliable error handling when changes are made to restaurant details or menu options.
* **Order and Cart Management Endpoints**: These endpoints are tested to confirm actions like adding items to the cart, placing orders, and updating order statuses (Received, In-Process, Out for Delivery, Delivered) work as expected. Testing includes adding/removing items from the cart, calculating totals, and checking estimated delivery times.

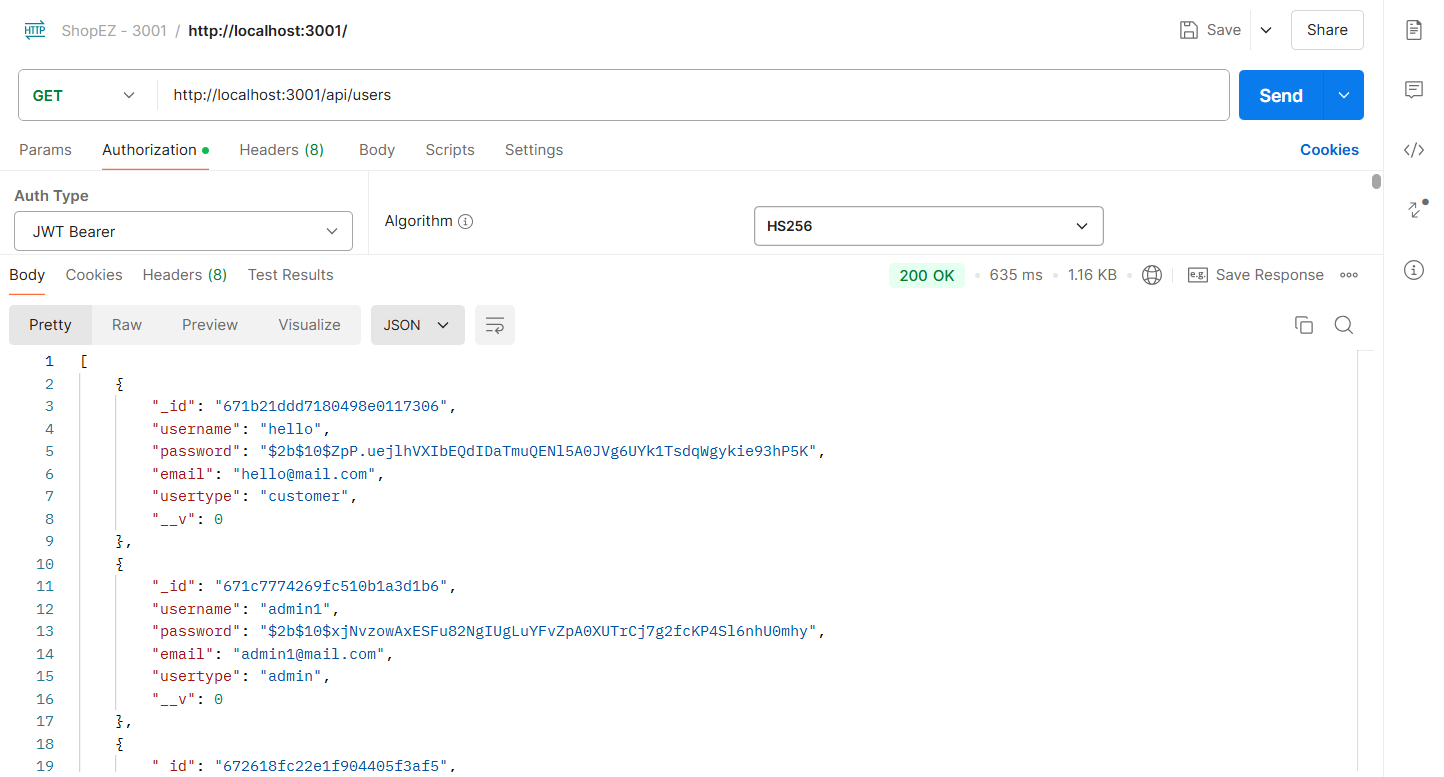
### Error Handling and Validation

Responses are checked for proper status codes (200, 400, 401, 404, etc.), ensuring that the system accurately handles errors and displays meaningful messages when invalid data is sent or when users attempt to access restricted features without authorization.

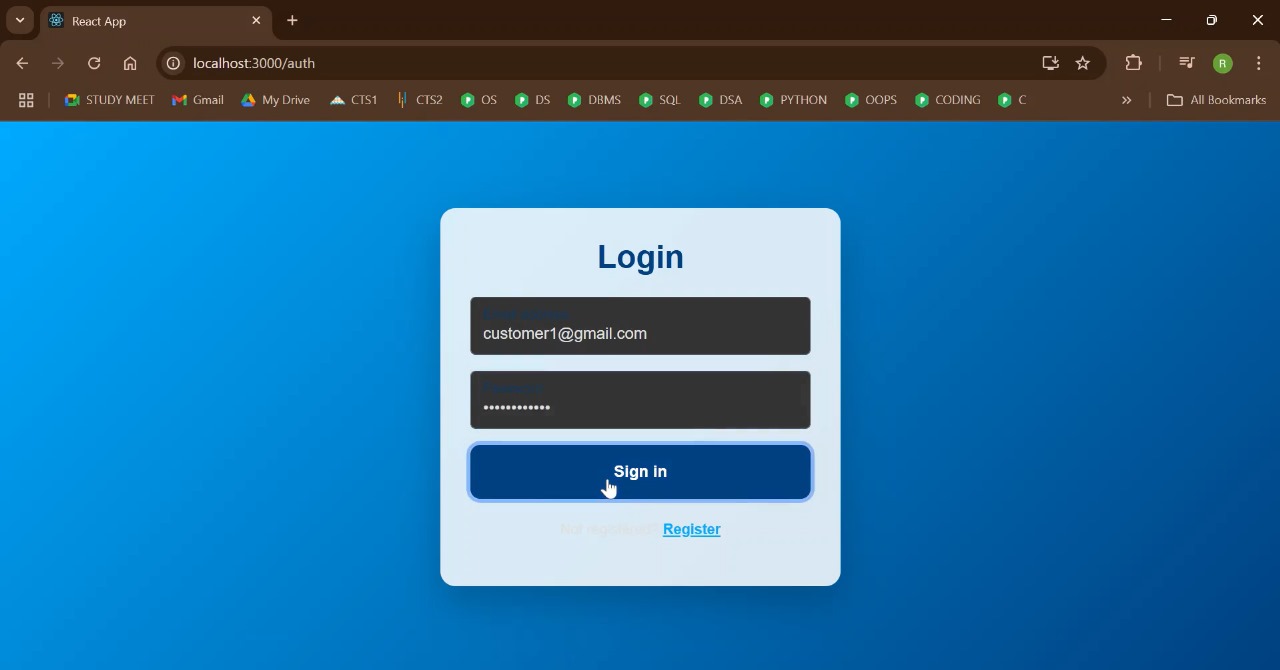
* **Positive and Negative Test Cases**: Test cases are created for both valid and edge scenarios, covering cases like incorrect login credentials, missing parameters, unauthorized actions, and invalid product IDs. These tests validate robust error handling, ensuring the system responds appropriately across different scenarios.

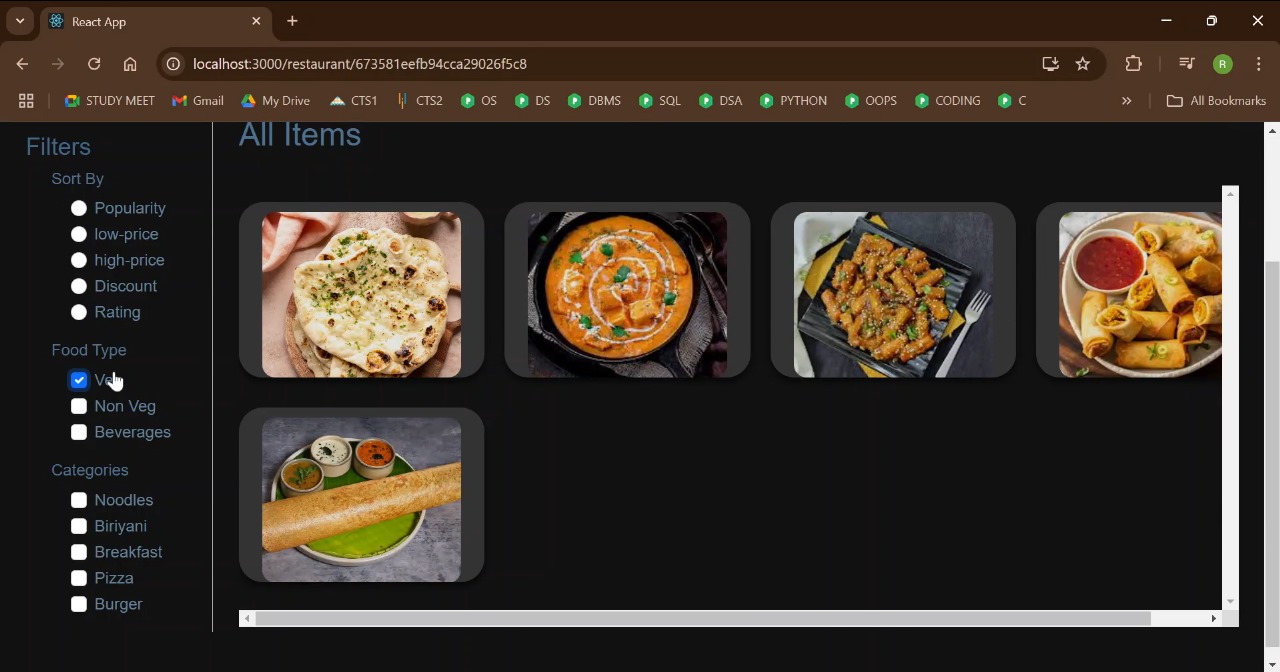
### Test Automation and Repeatability

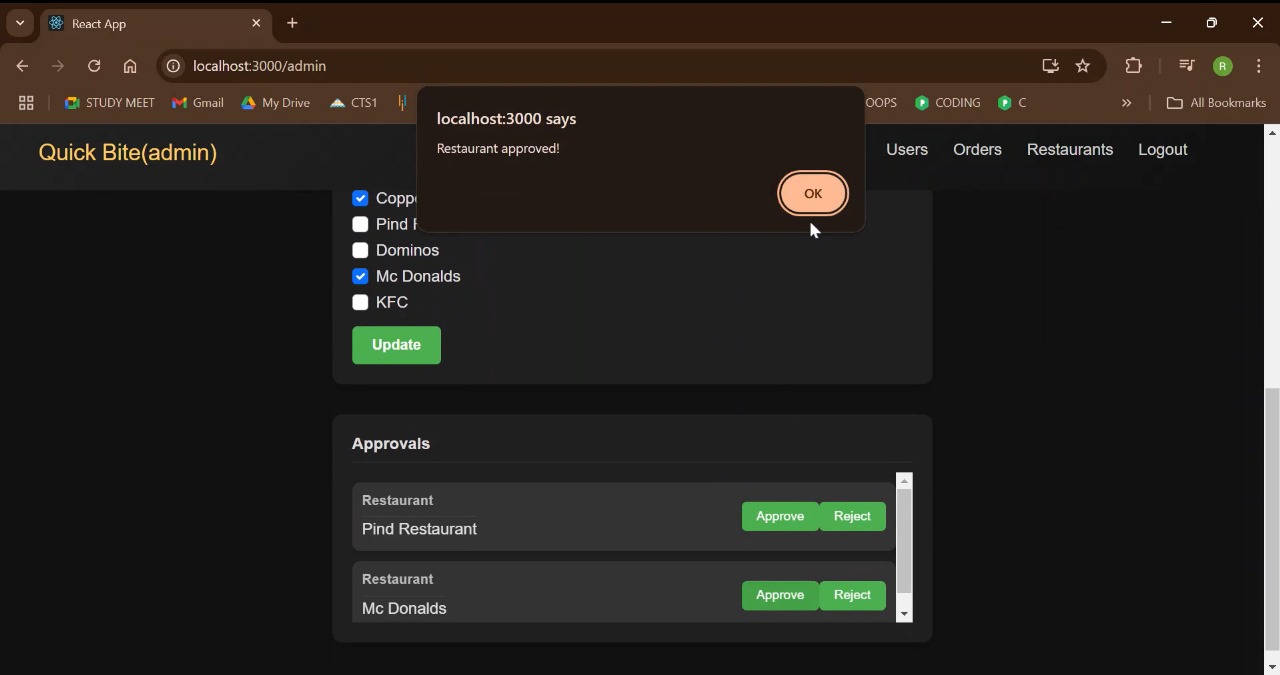
Postman collections group and automate tests, enabling the efficient repeatability of test cases for all API endpoints. These collections support testing consistency when changes or updates are applied to the application, making it easier to validate endpoint behavior and ensure reliable performance across the platform.

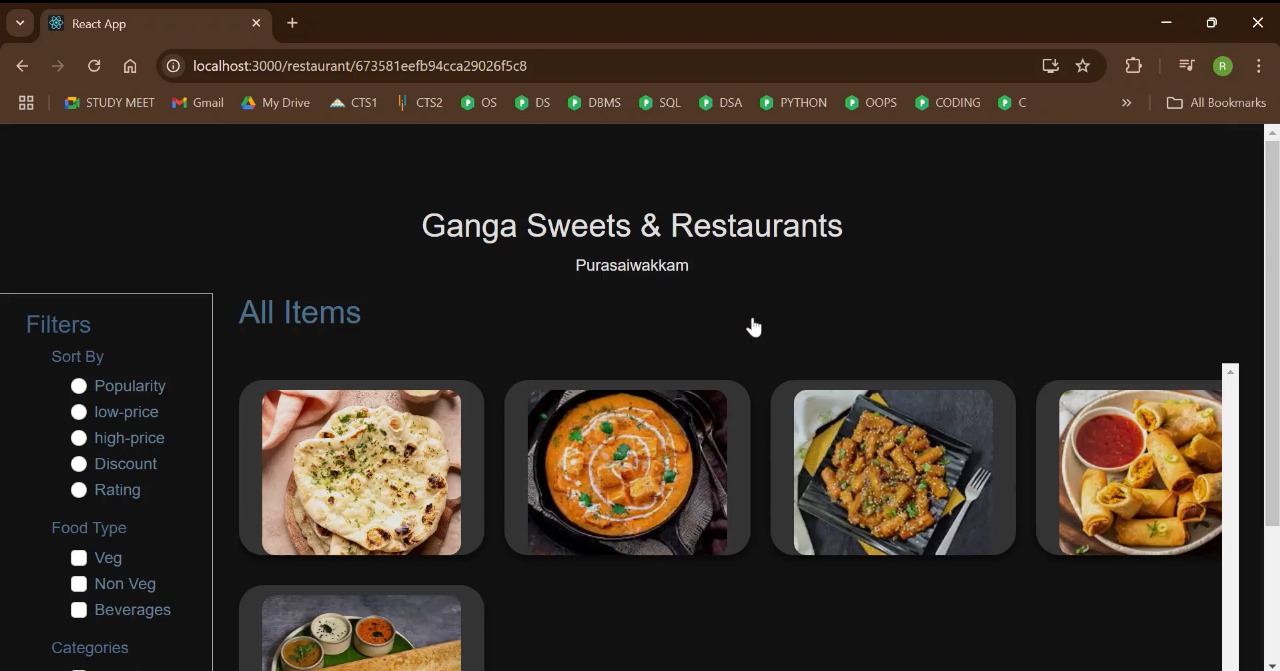


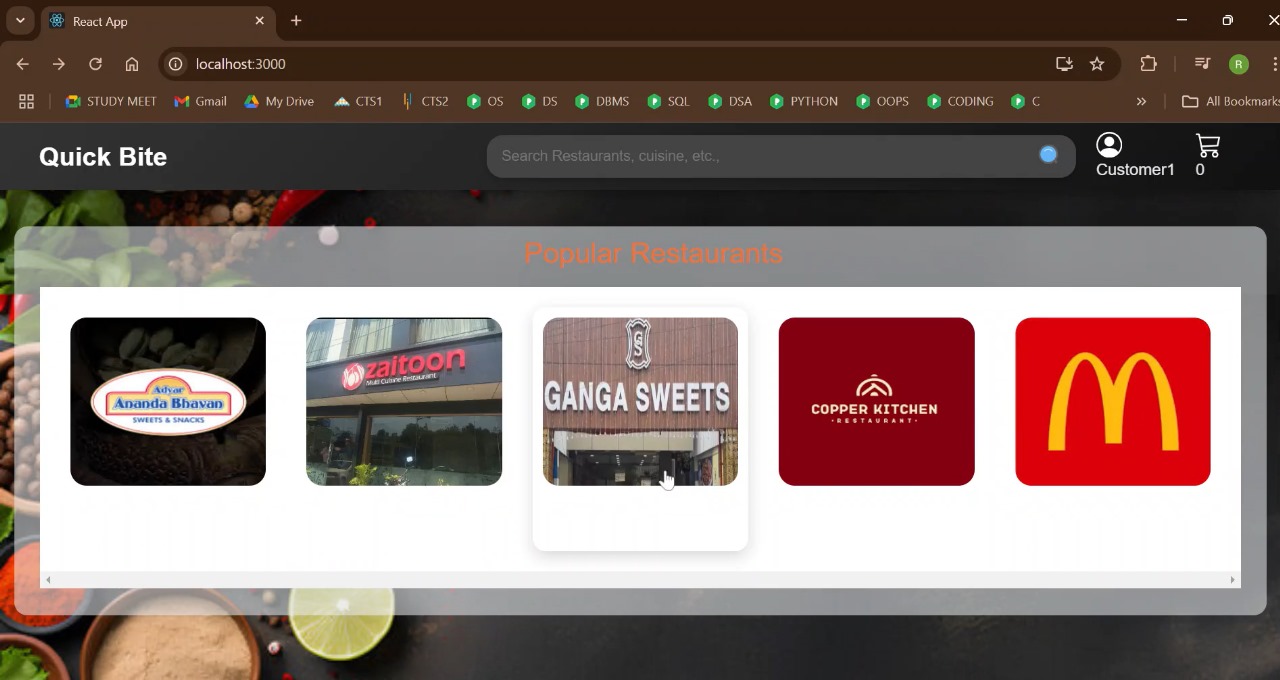
**11. SCREENSHOTS OR DEMO**

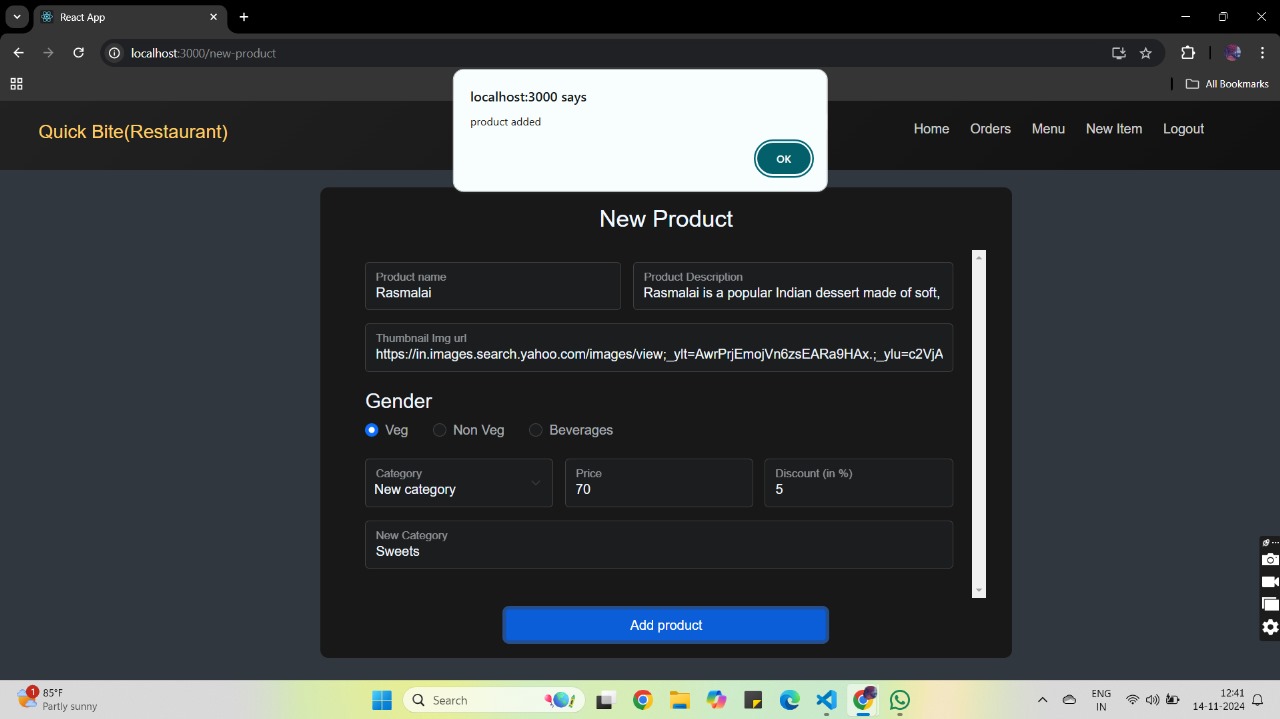


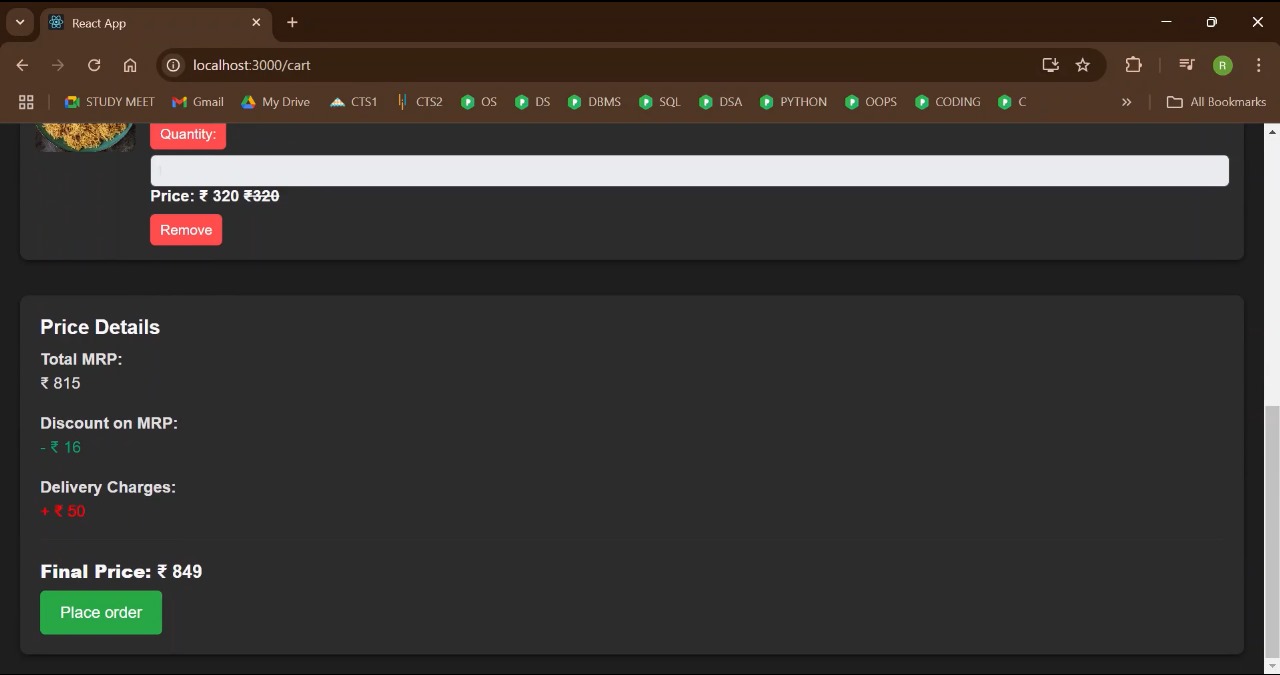


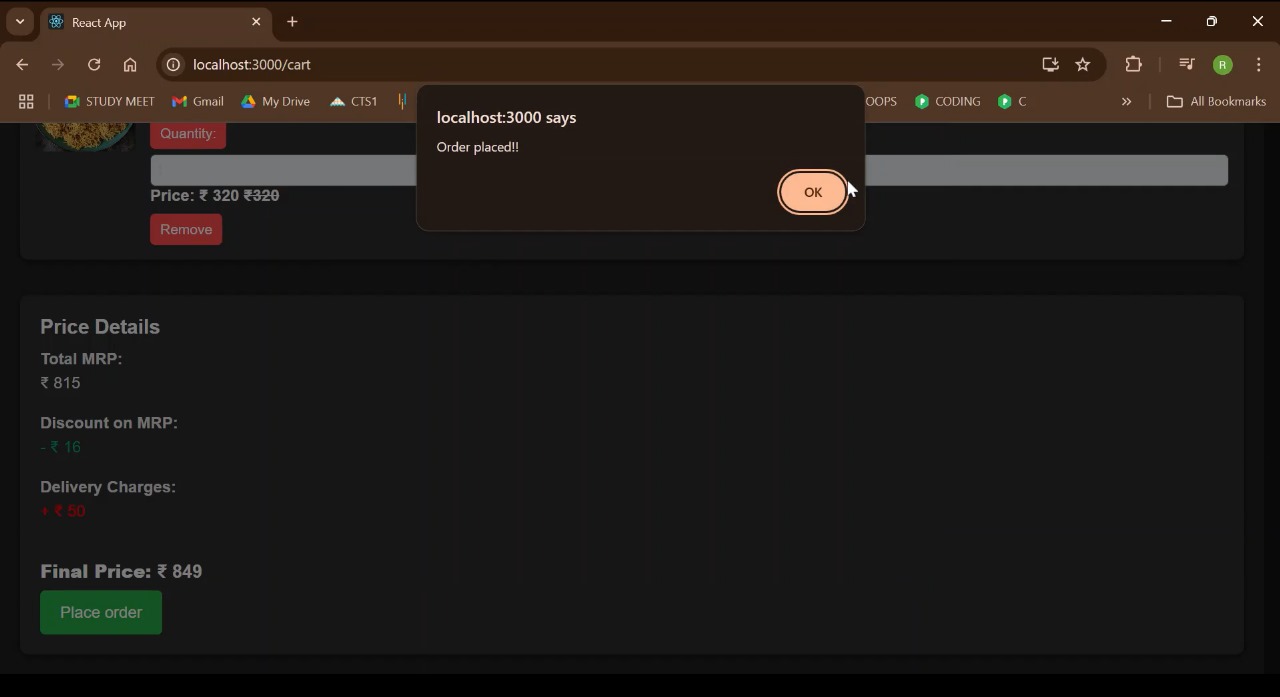


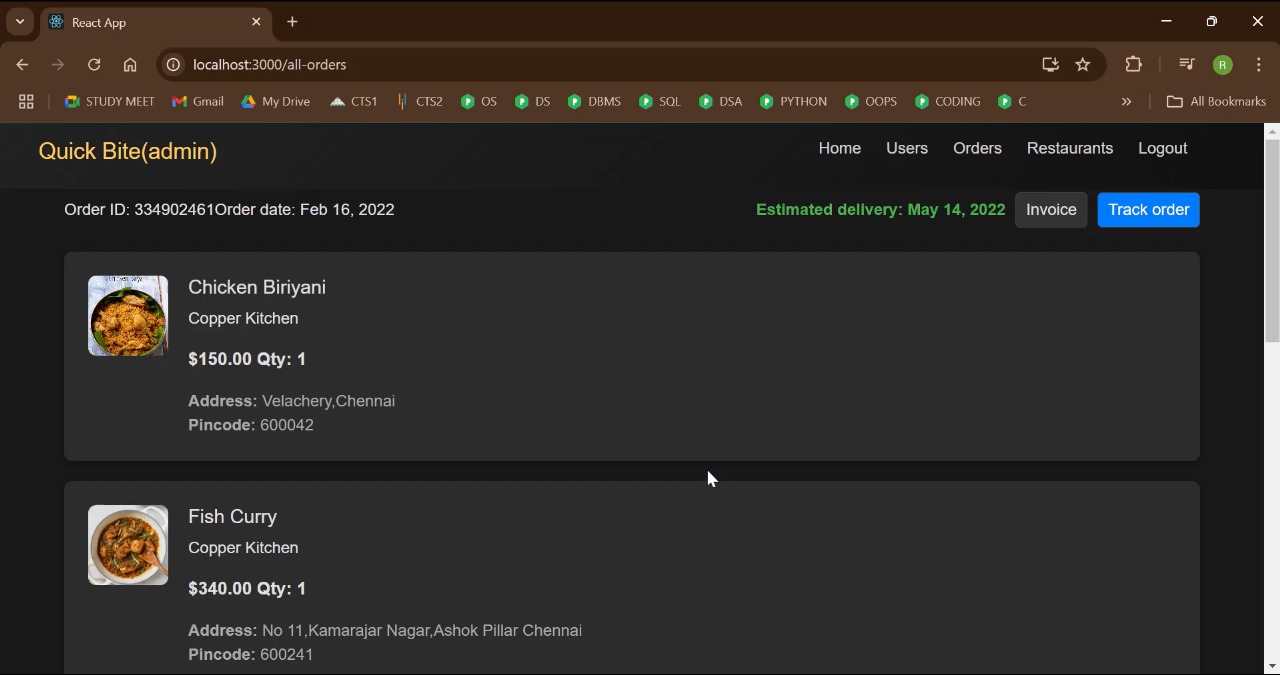


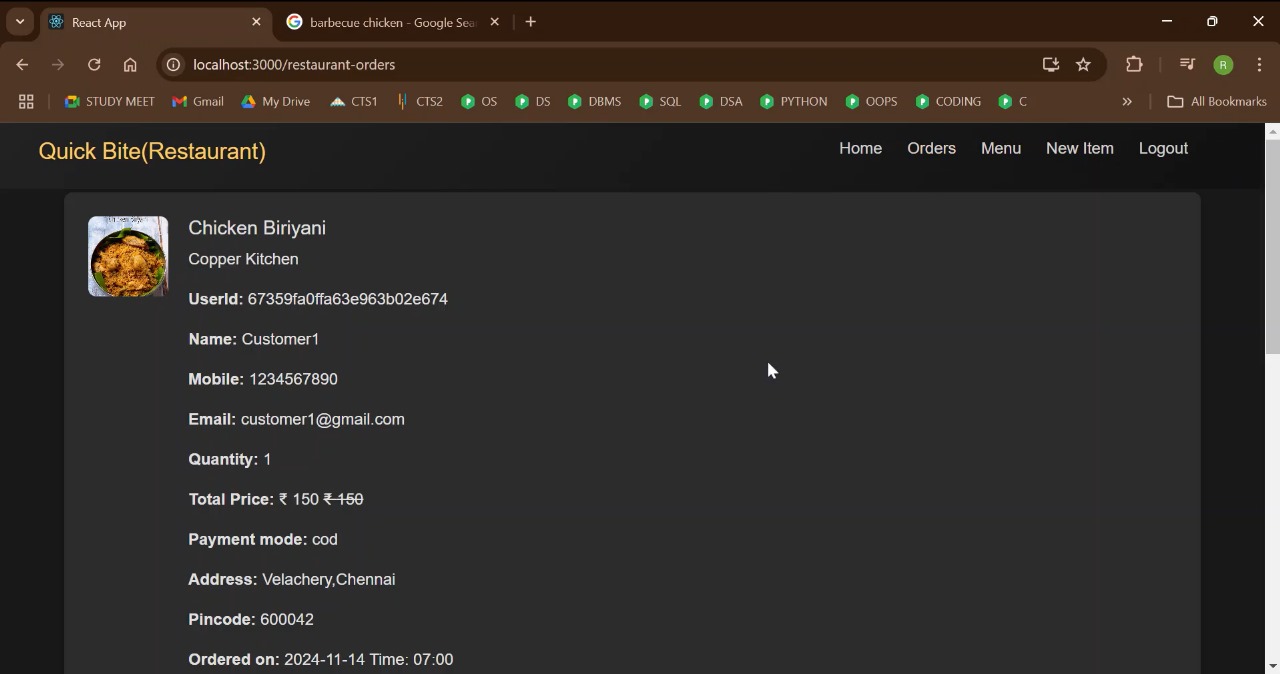


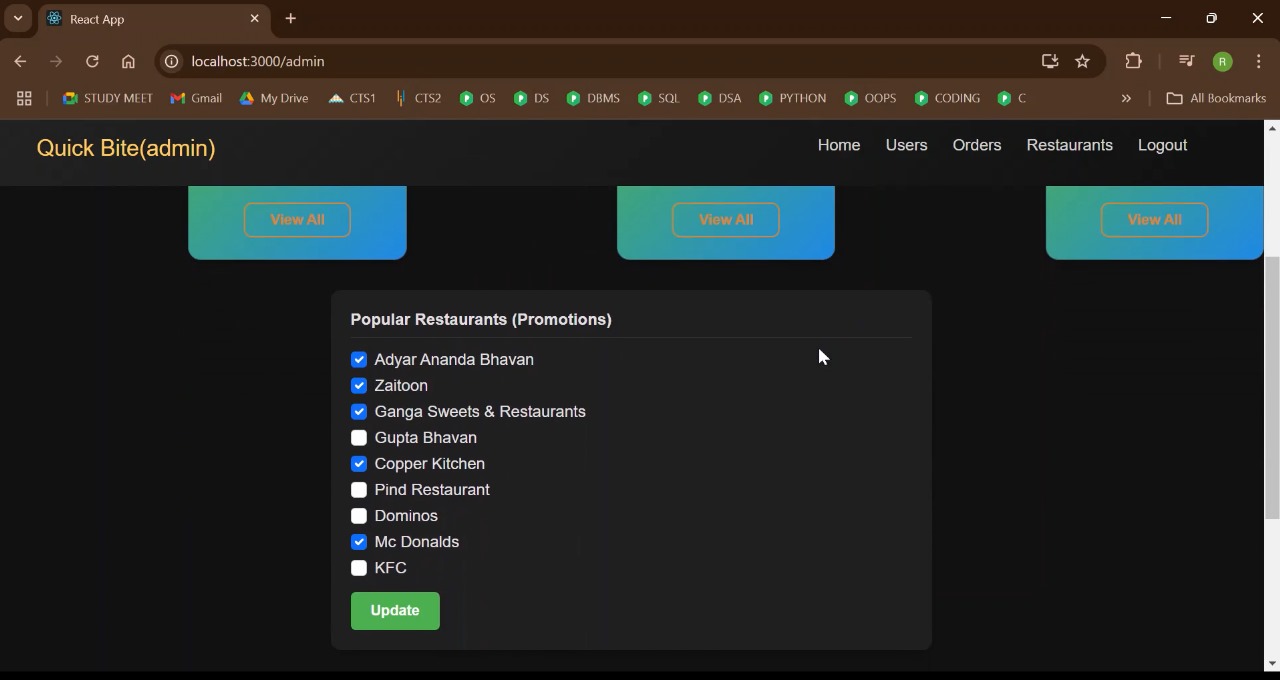


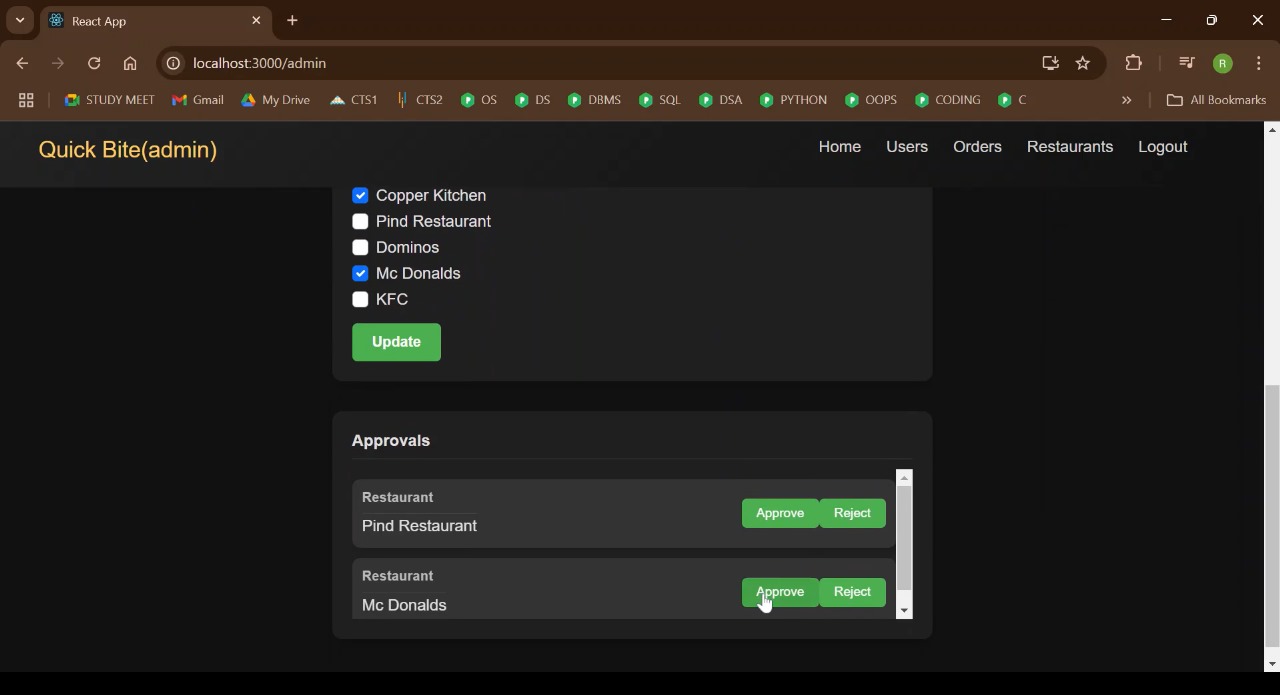


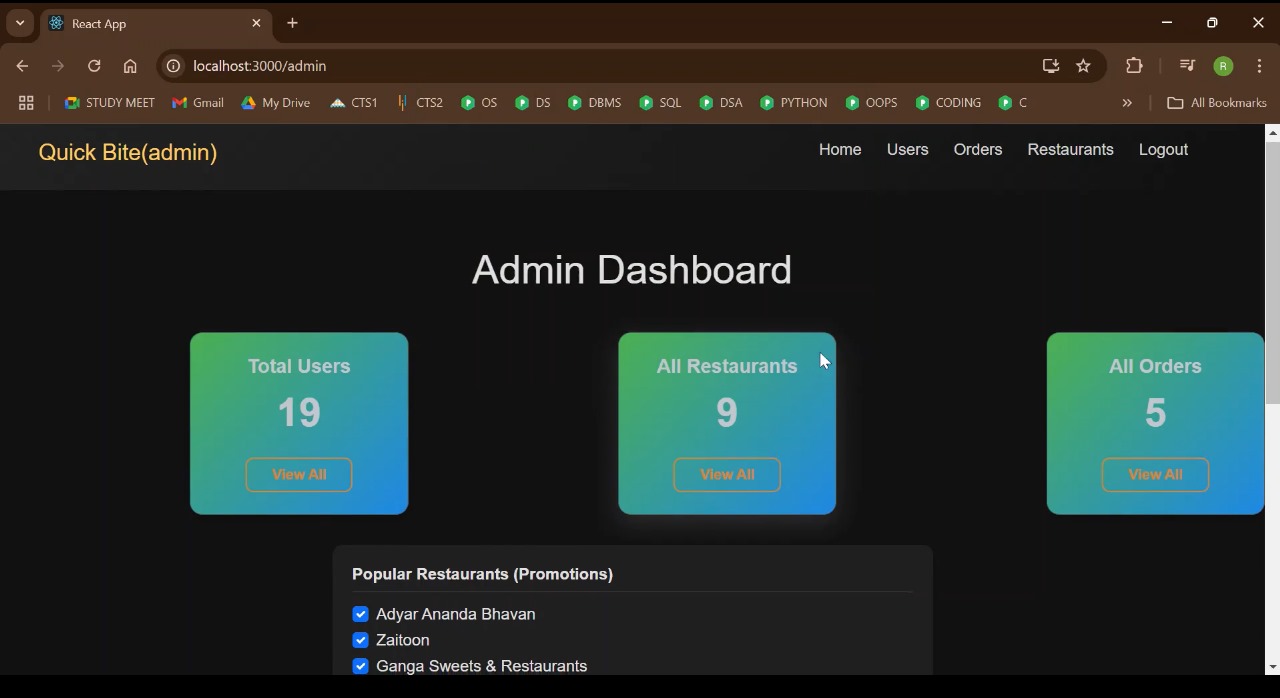


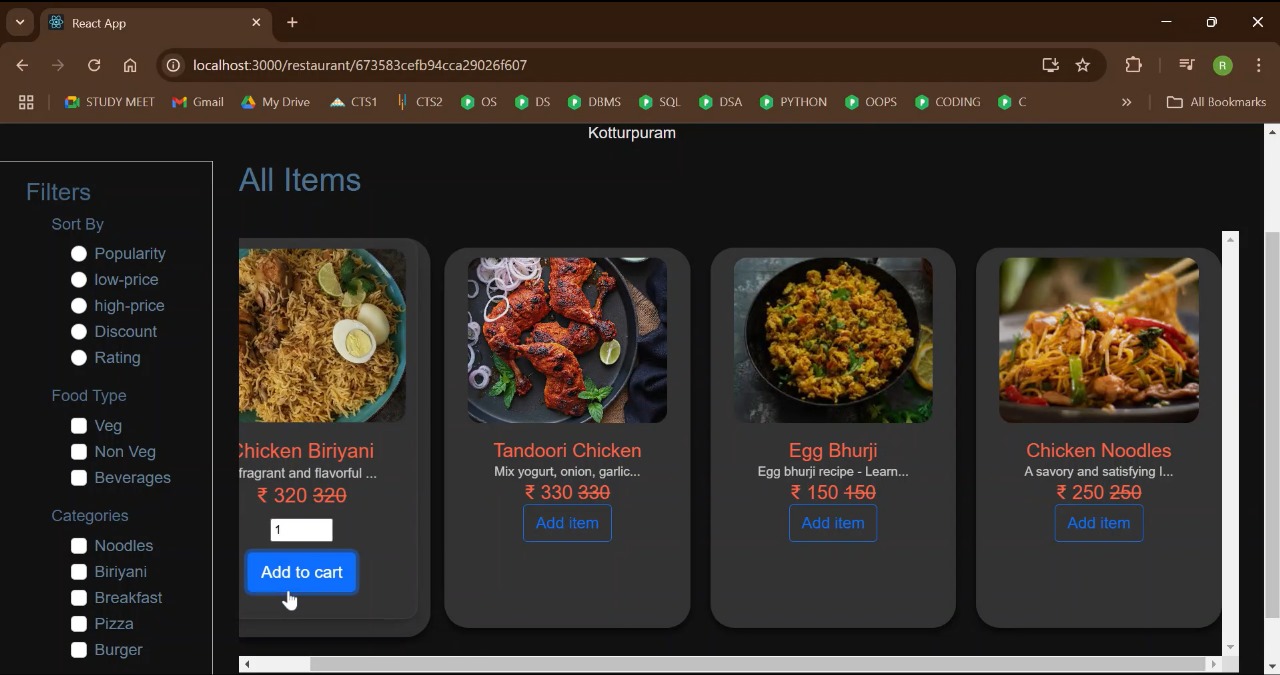


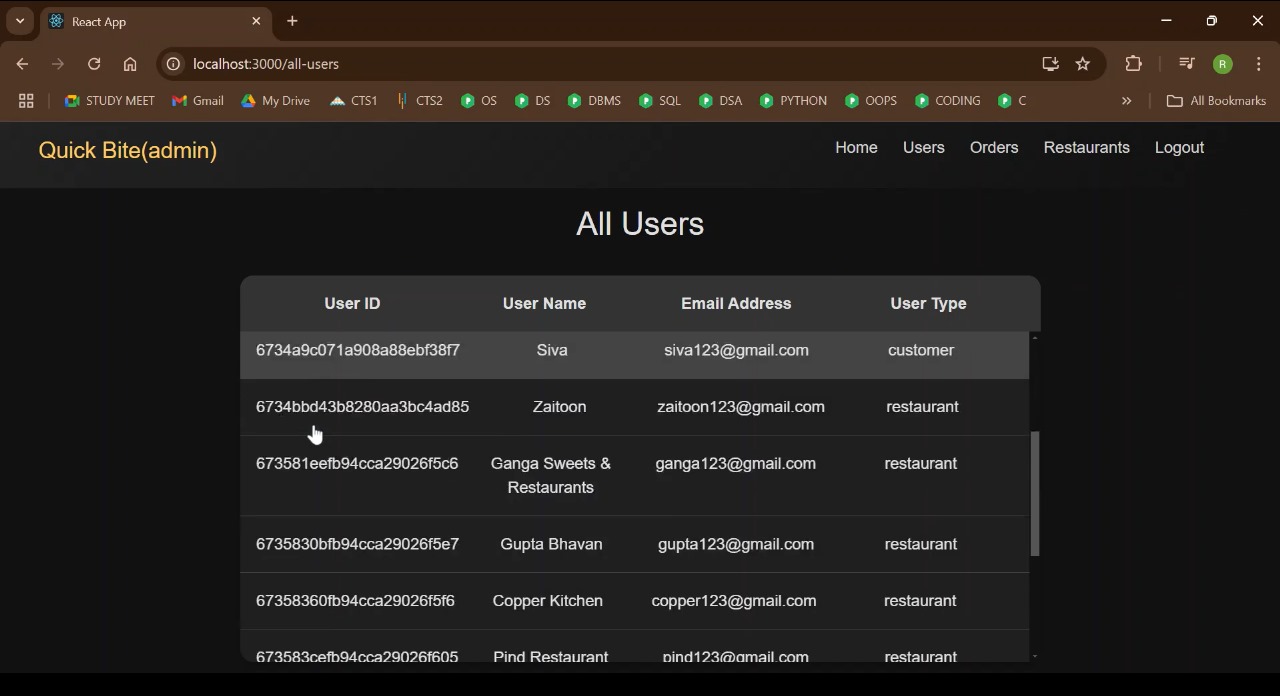












**12. KNOWN ISSUES**

1. **Forgot Password Functionality**:

* **Description**: The forgot password functionality does not currently include a verification step to confirm the validity of the email address before proceeding with the password reset process.
* **Impact**: If a user mistypes their email address, they will not receive any feedback, potentially leading to confusion as the reset link may not reach them.

1. **Cross-browser Compatibility**:

* **Description**: Certain features, like the image carousel and zoom functionality, may not display or function properly in all browsers, particularly older versions of Internet Explorer or Safari.
* **Impact**: Users on specific browsers may encounter layout or functionality issues, resulting in a suboptimal experience.

1. **Mobile Responsiveness**:

* **Description**: Some pages, including the Admin Dashboard and All foods view, may not be fully responsive on smaller devices or screens.
* **Impact**: Mobile users may experience issues with page alignment, layout, or broken UI components when accessing the app on smaller screens.

1. **Authentication Token Expiry Handling**:

* **Description**: The system does not consistently handle token expiry, which may lead to users staying logged in after their session expires or requiring a manual refresh to reauthenticate.
* **Impact**: Users may experience inconsistencies in session management, such as being logged out unexpectedly or having difficulty accessing features after token expiry.

These issues are being actively worked on and will be addressed in future updates to improve the overall user experience and functionality.

**13. FUTURE ENHANCEMENTS**

### 1. AI-Powered Personalization

* **Smart Recommendations**: Use machine learning to suggest menu items based on past orders, browsing behavior, and user preferences.
* **Dynamic Promotions**: Offer personalized discounts and promotions based on user behavior, location, and time of day.
* **Order Prediction**: Use predictive analytics to prompt users with a “one-click reorder” option based on their typical ordering patterns.

### 2. Real-Time Order Tracking with Live Updates

* **Map-Based Tracking**: Integrate live GPS tracking so users can see the real-time location of their delivery.
* **ETA Updates**: Provide accurate and dynamic ETA updates by tracking traffic patterns and preparation times.
* **In-App Notifications**: Notify users about each stage of the order (e.g., order confirmed, preparing, out for delivery) through push notifications or SMS.

### 3. Enhanced Customer Support

* **In-App Chatbot**: Offer a chatbot that can help with order queries, cancellations, and common FAQs. Integrate it with live customer support for complex issues.
* **Order Modification**: Allow users to modify orders within a short timeframe after placement, such as adding items or changing delivery details.
* **Feedback and Issue Reporting**: Allow users to rate items individually, provide feedback on orders, and report issues directly in the app.

### 4. Advanced Search and Filtering Options

* **Voice Search**: Enable voice search to make it easier for users to find items or restaurants.
* **Dietary and Allergy Filters**: Allow users to filter by dietary needs (e.g., vegan, gluten-free) and allergens.
* **Ingredient-Based Search**: Let users search for dishes containing specific ingredients.

### 5. Group Ordering and Splitting Bills

* **Group Orders**: Enable users to create group orders where multiple people can add items to a shared cart.
* **Bill Splitting**: Integrate a feature to split bills easily among multiple people with various payment methods.

### 6. Sustainable and Eco-Friendly Options

* **Eco-Friendly Packaging Option**: Let users choose eco-friendly packaging and track the environmental impact of their order choices.
* **Food Waste Management**: Offer discounts on soon-to-expire food items or create “chef’s special” options to reduce waste.
* **Carbon Footprint Tracking**: Show the carbon footprint of each order and suggest low-impact meals or delivery options.

### 7. Health and Nutrition Tracking

* **Caloric and Nutritional Information**: Provide caloric breakdowns and nutritional facts for each item.
* **Health Goals Integration**: Let users set dietary goals (e.g., low-calorie, high-protein), and provide recommendations.
* **Integration with Fitness Apps**: Sync with health apps to help users track calories and nutritional intake directly from their orders.