**FULL STACK DEVELOPMENT WITH MERN**

****

**SB Foods - Food Ordering App**

**(Revolutionizing Your Online Food Ordering Experience)**

**TEAM MEMBERS:**

* **NANDHINI R 311421104052**
* **BHARATHI S 311421104012**
* **GAYATHRI N 311421104020**
* **INDHU R V 311421104030**
* **JANANI A 311421104032**

**PROJECT OVERVIEW**

**Purpose**

The **SB Foods Management System** is a comprehensive software solution designed to streamline food business operations. Its primary aim is to automate processes like order management, inventory control, customer interactions, and data analysis to enhance the efficiency and profitability of food establishments, from small cafés to large restaurants.

The key goals of the project are:

* **Automate Order Processing:** Speed up order handling while minimizing human errors.
* **Improve Inventory Control:** Provide real-time updates and alerts to prevent stock issues.
* **Enhance Customer Experience:** Offer a user-friendly interface for customers and managers.
* **Provide Business Insights:** Analyze data to support informed decision-making.

**Features**

The system includes a range of features tailored to meet the operational needs of a food business:

1. **Order Management:**
   * Customers can place orders online with real-time updates on order status.
   * Supports order customization and tracking.
2. **Inventory Management:**
   * Real-time tracking of stock levels, with low-stock alerts for timely restocking.
   * An inventory dashboard displaying stock usage trends for better management.
3. **Customer Feedback:**
   * Collects and stores customer feedback, enabling analysis for service improvements.
   * Automated notifications for positive and negative feedback handling.
4. **Menu Management:**
   * Easy updates to the menu via an admin panel, including price changes and new items.
5. **Secure Payments:**
   * Integrates with multiple payment gateways like PayPal, Stripe, and Razorpay.
   * Offers secure and flexible payment options for customers.
6. **CRM and Personalization:**
   * Maintains customer data and preferences to offer personalized recommendations.
   * Automated notifications for order updates and promotions.
7. **Analytics and Reporting:**
   * Provides a dashboard with sales metrics, top-selling items, and customer insights.
   * Generates detailed reports for business analysis.
8. **User Role Management:**
   * Supports Admin, Manager, and Customer roles with varying access permissions.
   * Role-specific dashboards for streamlined management.
9. **Security Features:**
   * Implements strong authentication using Firebase and JSON Web Tokens (JWT).
   * Data encryption for secure handling of sensitive information.
10. **API Integrations:**
    * Integrates with Google Maps for delivery tracking and email services for notifications.
    * Social media integration for enhanced customer engagement.

**User Scenarios**

1. **Customer Experience:**
   * A customer logs in, places an order, chooses a payment method, and receives a confirmation. They can track their order status until delivery.
2. **Manager’s Workflow:**
   * A manager checks stock levels, responds to low-stock alerts, and places restocking orders. The inventory dashboard updates automatically.
3. **Admin Data Review:**
   * The admin accesses the analytics dashboard to review sales performance and adjusts promotional offers based on data insights.

**ARCHITECTURE**

The **SB Foods Management System** is designed with a modular architecture, consisting of three main components: Frontend, Backend, and Database. This structure ensures scalability, flexibility, and ease of maintenance.

**Frontend Architecture**

The **frontend** is developed using **React**, a powerful JavaScript library known for building dynamic and interactive user interfaces. The design focuses on providing a seamless experience for users across different devices, including desktops, tablets, and mobile phones.

**Key Components:**

1. **User Interface (UI):**
   * Responsive design using **CSS modules** and **React Bootstrap** for consistent styling.
   * Utilizes **React Router** for navigation, enabling a single-page application (SPA) experience.
   * Built-in components for features like product listings, order forms, feedback submissions, and user profiles.
2. **State Management:**
   * Uses **React Context API** for managing global state, including user authentication, cart items, and order details.
   * Implements **Redux** for handling complex state updates and asynchronous actions, such as API calls for fetching product data.
3. **User Experience Enhancements:**
   * **Form validation** using React hooks for improved input handling.
   * Loading spinners and toast notifications provide feedback to the user during API requests and form submissions.

**Backend Architecture**

The **backend** is built using **Node.js** with **Express.js**, a popular framework for building robust server-side applications. The backend serves as the core of the system, handling API requests, business logic, and third-party integrations.

**Key Components:**

1. **API Design:**
   * Follows **RESTful API** principles for clear and consistent endpoints.
   * API routes include /products, /orders, /users, /feedback, and /inventory.
2. **Authentication & Authorization:**
   * Uses **JWT (JSON Web Tokens)** for secure user authentication.
   * Implements role-based access control (RBAC) to manage permissions for Admin, Manager, and Customer roles.
3. **Middleware:**
   * Custom middleware for **input validation** using the **Joi** library.
   * **Error handling** middleware captures and logs errors, providing user-friendly error messages.
   * **Cors middleware** enables cross-origin requests from the React frontend.
4. **Payment Integration:**
   * Supports multiple payment gateways like **PayPal**, **Stripe**, and **Razorpay** for secure transactions.
   * Uses **webhooks** to handle payment confirmation and update order status in real time.

**Database Architecture**

The project uses **MongoDB**, a NoSQL database known for its flexibility and scalability, as the primary data storage solution. MongoDB’s document-based structure allows for efficient storage of complex data like product information, customer orders, and inventory details.

**Key Components:**

1. **Database Design:**
   * Uses a **flexible schema** to handle varied data structures for different products and customer orders.
   * Collections include:
     + **Users:** Stores user information, hashed passwords, and roles (Admin, Manager, Customer).
     + **Products:** Contains product details like name, category, price, and stock quantity.
     + **Orders:** Tracks order information, including user ID, product IDs, total amount, and order status.
     + **Feedback:** Stores customer feedback and ratings for analysis.
     + **Inventory:** Monitors stock levels and updates based on order placements.
2. **Data Interactions:**
   * Utilizes **Mongoose**, an Object Data Modeling (ODM) library for MongoDB, to simplify data operations.
   * Implements **indexes** for faster search queries on frequently accessed fields, like product names and order status.
3. **Data Security:**
   * Sensitive data, such as passwords and payment details, are encrypted using **bcrypt** for secure storage.
   * Implements **role-based access** control at the database level to prevent unauthorized access to sensitive information.
4. **Backup and Recovery:**
   * Automated **database backup** schedule using MongoDB’s built-in tools to ensure data is not lost in case of server failure.
   * Regular data exports allow for easy restoration and disaster recovery.

**System Flow**

1. **User Request:**
   * A customer accesses the React frontend and places an order, which triggers a POST request to the backend.
2. **Backend Processing:**
   * The backend API validates the input data, checks inventory availability, and processes the order.
   * If the payment is successful, the order status is updated, and a confirmation email is sent to the customer.
3. **Database Interaction:**
   * The MongoDB database updates the product stock, creates a new order document, and stores the transaction details.
4. **Response:**
   * The backend sends a response back to the frontend, where the order confirmation is displayed to the customer.

**SETUP INSTRUCTIONS**

The **SB Foods Management System** requires certain software dependencies and environment configurations for smooth setup and deployment. Follow the instructions below to install and set up the project on your local machine.

**Prerequisites**

Ensure the following software and tools are installed on your system:

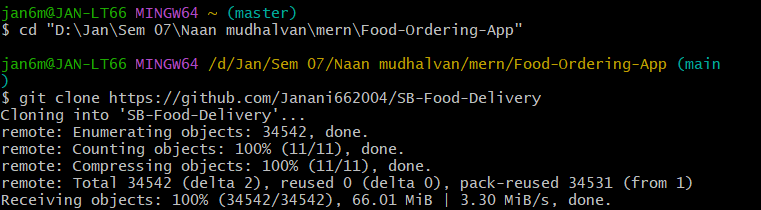
* **Node.js**: Version 18.0 or higher. This is required to run the backend server and manage project dependencies.
  + Download Node.js
* **MongoDB**: MongoDB Atlas (cloud-based) or a local MongoDB setup is needed for data storage.
  + [MongoDB Atlas](https://www.mongodb.com/atlas/database)
* **Git**: Version control system to clone the project repository.
  + [Download Git](https://git-scm.com/downloads)

**Installation**

Follow these steps to set up the project locally:

1. **Clone the Repository:**

First, open a terminal window and clone the project repository from GitHub.



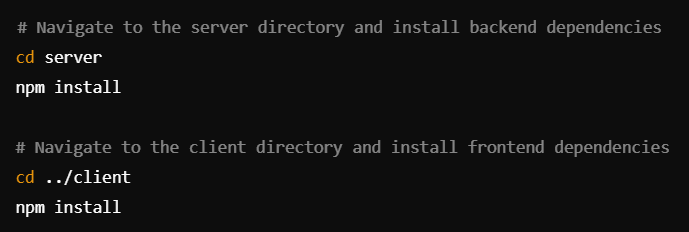
1. **Navigate to the Project Directory:**

Change your working directory to the project folder.



1. **Install Dependencies:**

The project contains two main directories: client (for the React frontend) and server (for the Node.js backend). You need to install dependencies for both.



1. **Environment Setup:**

Create a .env file in the root directory of both server and client. This file will store sensitive information like API keys, database URIs, and secret tokens.

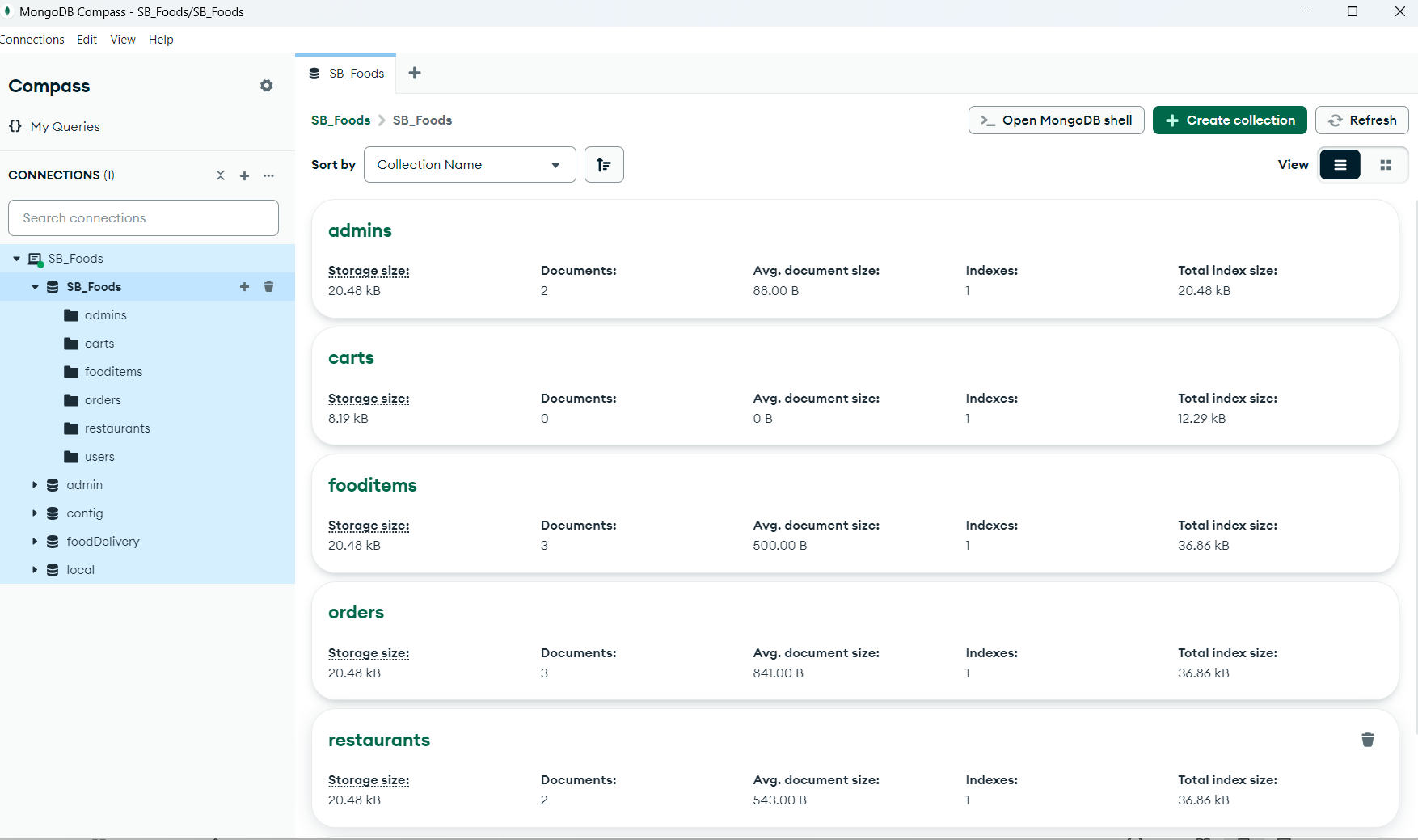
1. **Database Setup:**

If you are using MongoDB Atlas:

* + Create a cluster in MongoDB Atlas and get the **connection URI**.
  + Replace the MONGODB\_URI in the server .env file with your connection string.

If you prefer a local MongoDB setup:

* + Start MongoDB locally and use the URI mongodb://localhost:27017/sbfoods in the .env file.



1. **Configure API Keys:**
   * Obtain API keys for any third-party services used, such as **Stripe** for payments or **Razorpay**.
   * Add these API keys to the server .env file:

**FOLDER STRUCTURE**

The project is divided into two main parts: the **Client** (React frontend) and the **Server** (Node.js backend). Below is the breakdown of the folder structure:

**Client (React Frontend)**

The React frontend handles the user interface and interaction with the backend API. The main structure is as follows:

* **public/**  
  Contains static assets like the index.html template, favicon, and manifest files.
* **src/**  
  This is the core of the React application, containing all source code.
  + **src/components/**  
    Houses reusable UI components that are used across the application.
    - **ProductCard.js**: Component to display product information.
    - **Navbar.js**: Navigation bar component for page navigation.
    - **Sidebar.js**: Sidebar for quick access to different sections.
  + **src/pages/**  
    Contains the main pages of the application, organized for better routing and code separation.
    - **Dashboard.js**: Main dashboard displaying key metrics and charts.
    - **Orders.js**: Page for managing orders and viewing order history.
    - **Products.js**: Page for managing product listings.
  + **src/services/**  
    Includes API service files for making HTTP requests using Axios or Fetch API.
    - **apiService.js**: Centralized service for API calls (e.g., fetching products, updating orders).
  + **src/context/**  
    Manages global state using React Context API for user session and authentication.
    - **AuthContext.js**: Provides user authentication state and methods.
  + **src/utils/**  
    Helper functions and utilities used across the application.
    - **dateFormatter.js**: Utility for formatting dates.
    - **constants.js**: Stores common constants and configuration values.
  + **src/App.js**  
    The main application file that defines routes and integrates all components.
  + **src/index.js**  
    Entry point of the React application, rendering the main App component.

**Server (Node.js Backend)**

The Node.js backend handles API requests, business logic, and database interactions. The structure is organized for clarity and modularity:

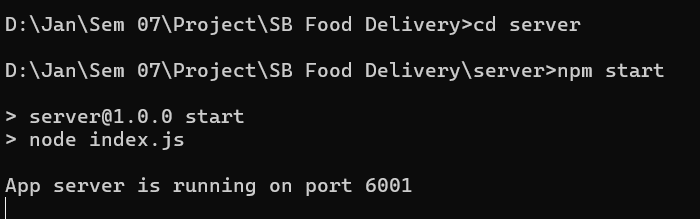
* **server.js**  
  The main entry point for the backend application. Configures Express, connects to the MongoDB database, and starts the server.
* **config/**  
  Contains configuration files for database and server settings.
  + **dbConfig.js**: Manages MongoDB connection using Mongoose.
  + **env.js**: Environment variables configuration for secure credentials.
* **models/**  
  Mongoose schemas defining the structure of database collections.
  + **Product.js**: Schema for products, including fields for name, price, stock, and category.
  + **Order.js**: Schema for orders, including fields for order items, total price, and customer details.
  + **User.js**: Schema for user accounts, handling roles (e.g., admin, user).
* **routes/**  
  Defines all API endpoints for the application, organized by feature.
  + **productRoutes.js**: Routes for product-related operations (e.g., create, update, delete products).
  + **orderRoutes.js**: Routes for handling order operations (e.g., place an order, view orders).
  + **userRoutes.js**: Routes for user authentication and profile management.
* **controllers/**  
  Contains the business logic and functionality for handling requests and responses.
  + **productController.js**: Handles product creation, retrieval, updates, and deletion.
  + **orderController.js**: Manages order placement, updates, and viewing order history.
  + **userController.js**: Manages user registration, login, and profile updates.
* **middlewares/**  
  Middleware functions for request handling and validation.
  + **authMiddleware.js**: Ensures routes are accessed by authenticated users only.
  + **errorHandler.js**: Centralized error handling middleware for consistent error responses.
* **helpers/**  
  Utility functions for server-side operations.
  + **tokenHelper.js**: Handles JWT token creation and verification for authentication.
* **views/**  
  Contains static files or templates, if needed for server-side rendering or email templates.
* **tests/**  
  Holds unit and integration tests for backend components.
  + **product.test.js**: Tests product-related functionalities.
  + **order.test.js**: Tests order processing logic.

**RUNNING THE APPLICATION**

1. **Start the Backend Server:**

In the server directory, start the Node.js server:

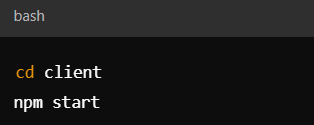


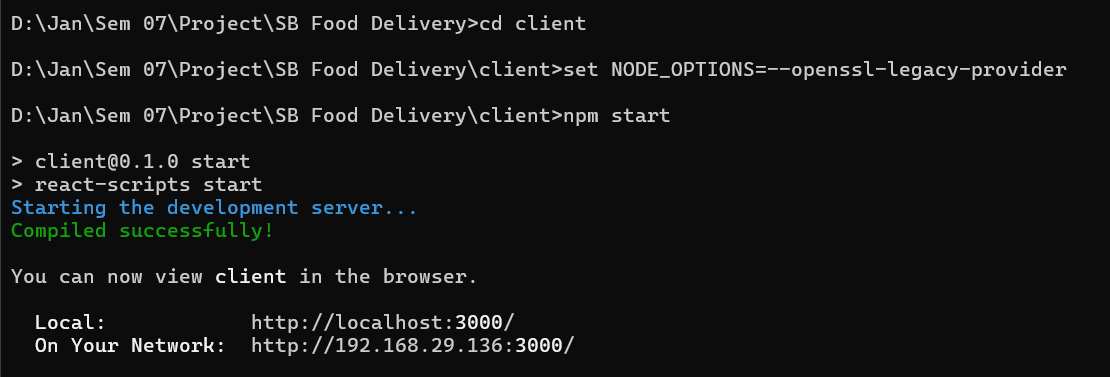


The backend server will run on <http://localhost:6001>.

1. **Start the Frontend Server:**

In the client directory, start the React app:





The frontend server will run on <http://localhost:3000>.

**API DOCUMENTATION**

This project provides a RESTful API for managing products, orders, and user authentication. The API follows a consistent structure with standard HTTP methods and status codes.

 **Product API:**

* GET /api/products: Fetch all products.
* POST /api/products: Add a new product (Admin only).

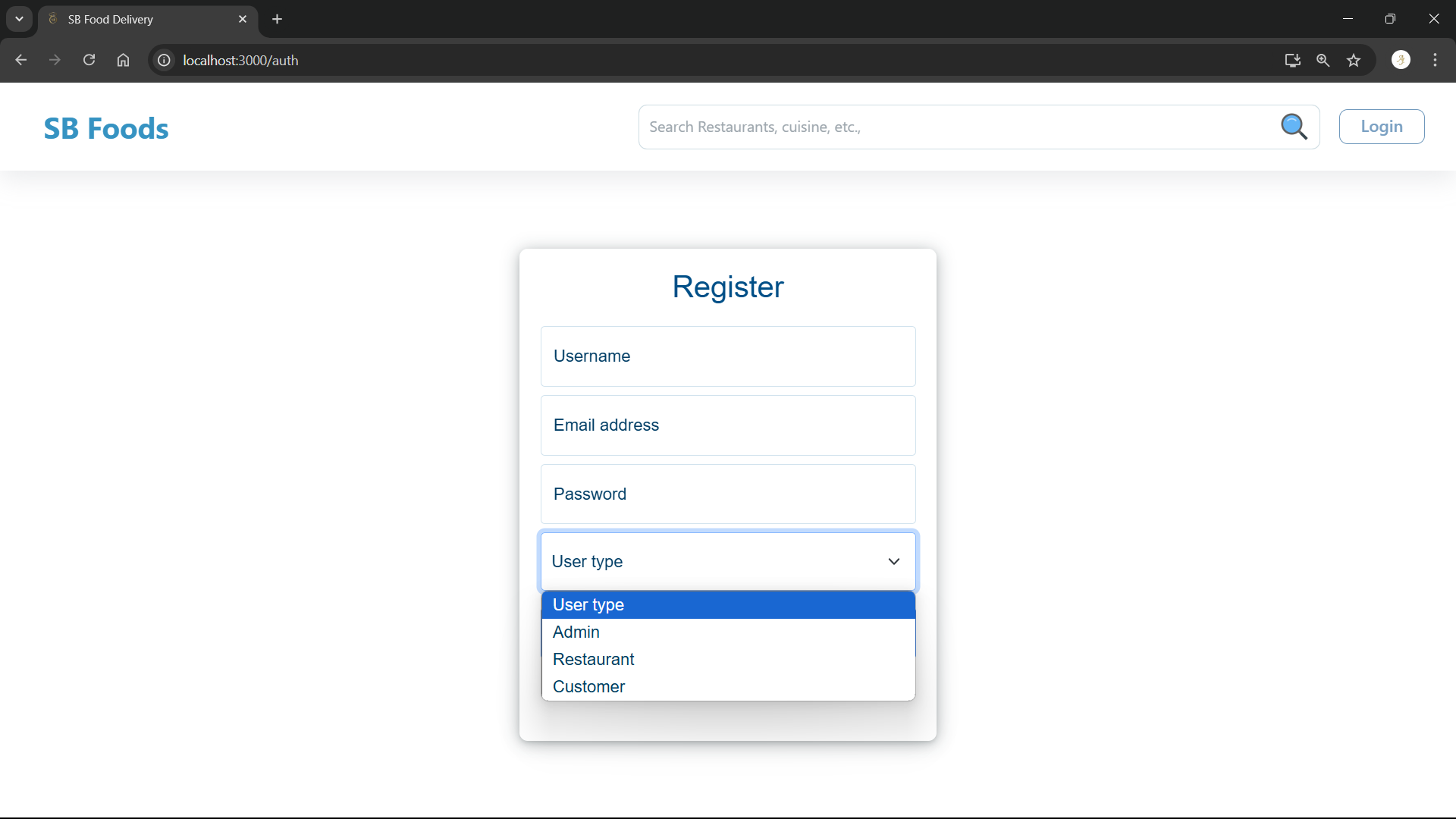
 **Order API:**

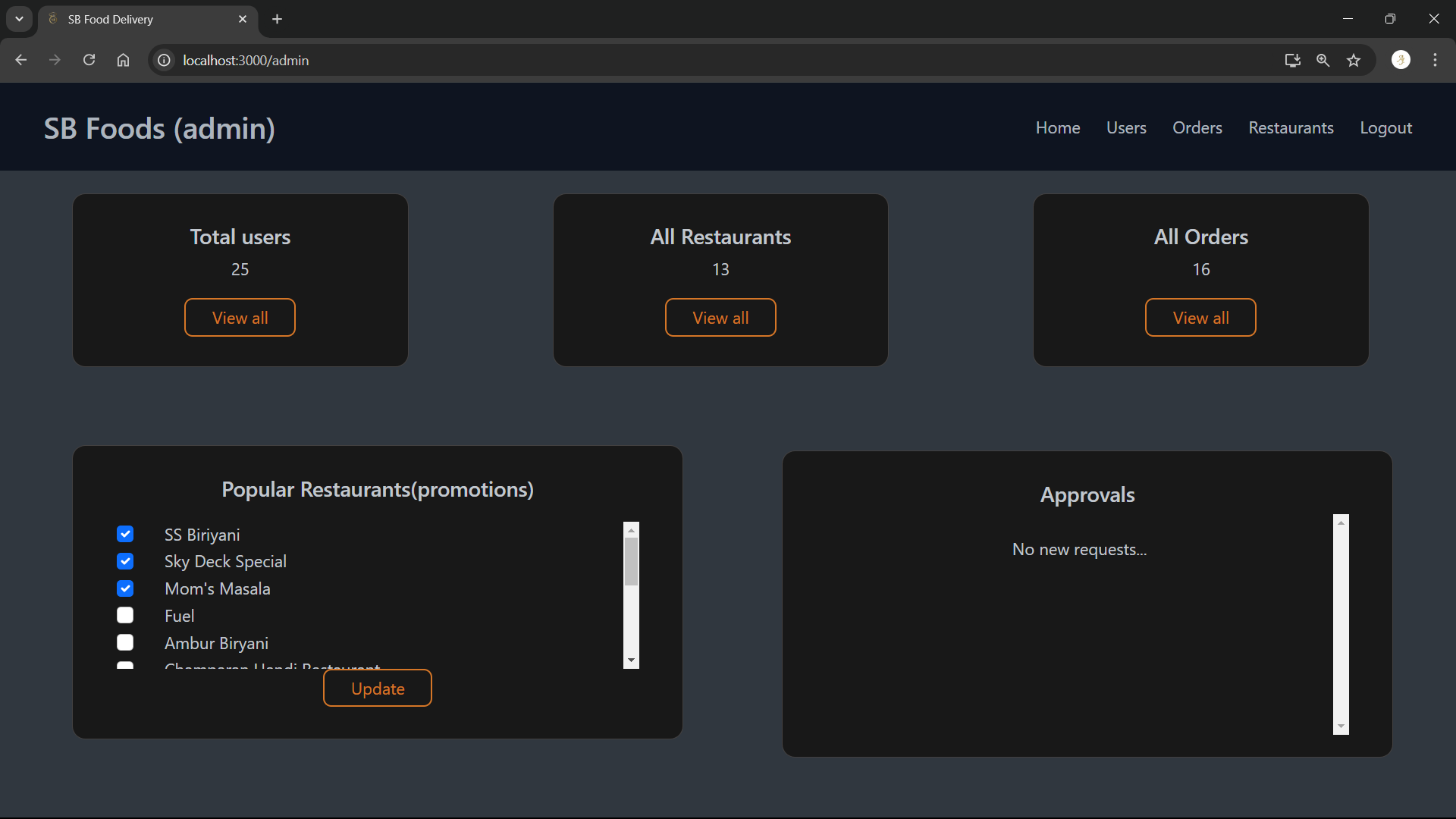
* GET /api/orders: Fetch all orders.
* POST /api/orders: Create a new order.

 **User API:**

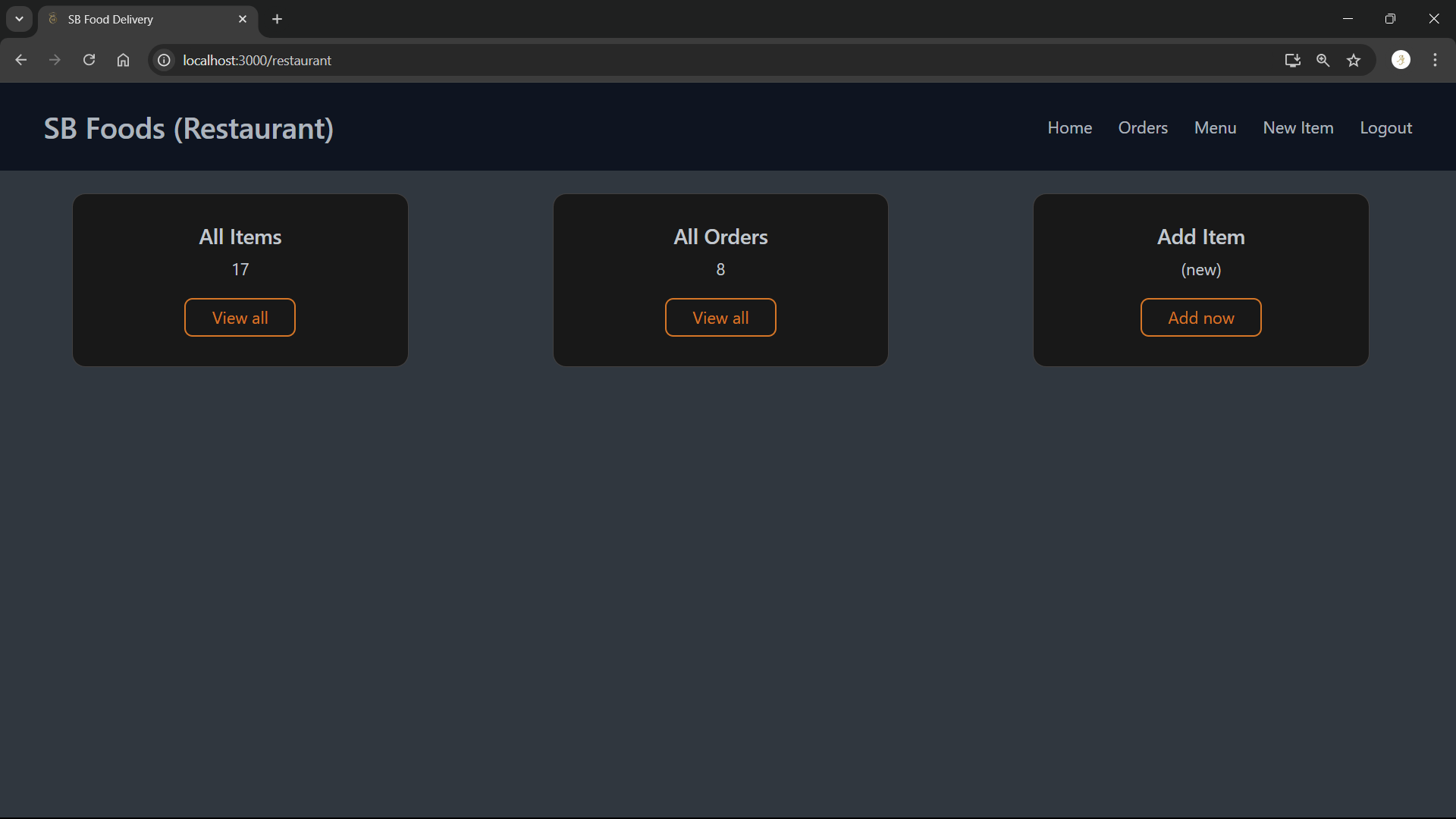
* POST /api/auth/login: User login with JWT token.

**USER INTERFACE**

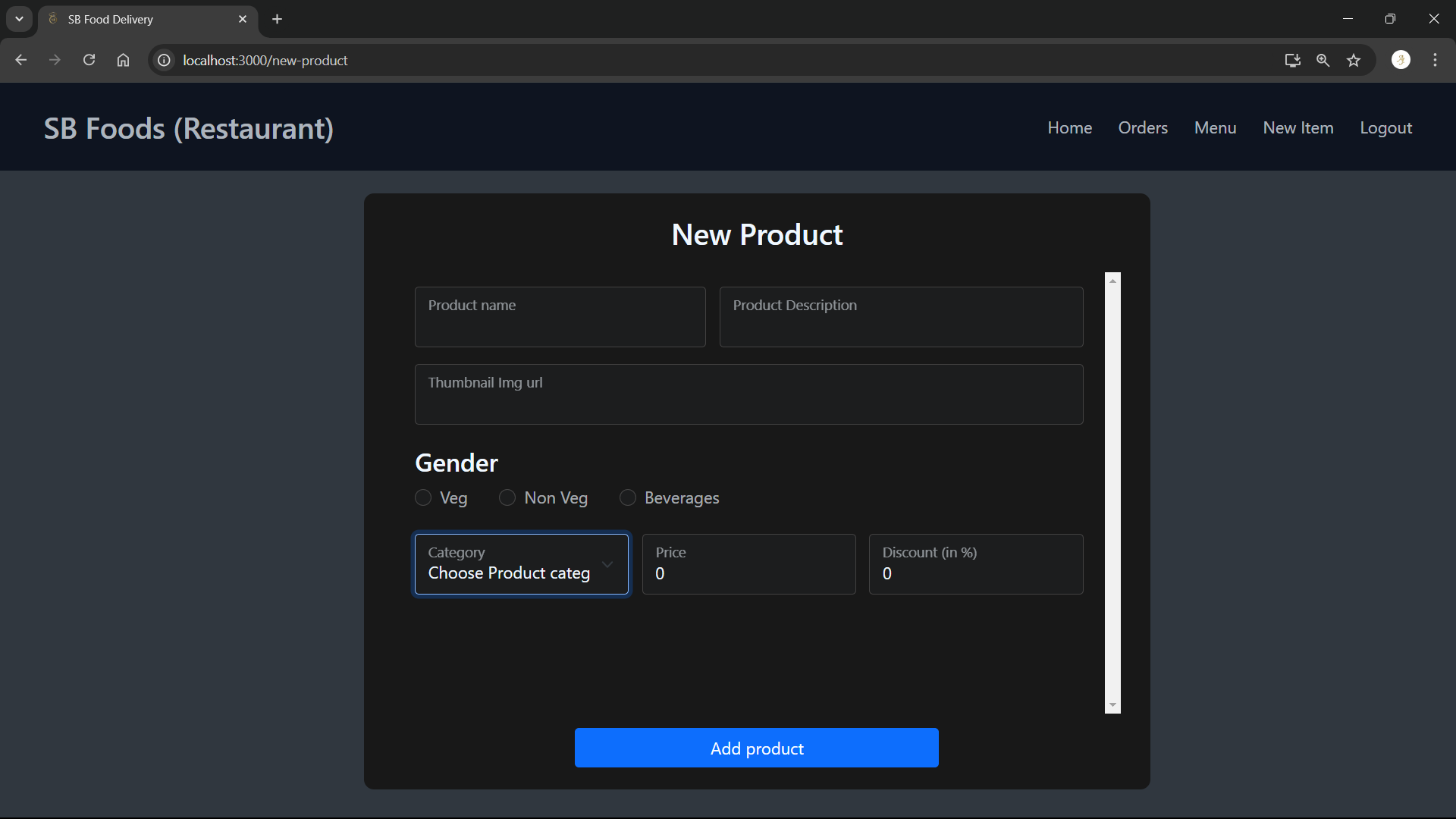
* **Registration page**
* **Admin dashboard**



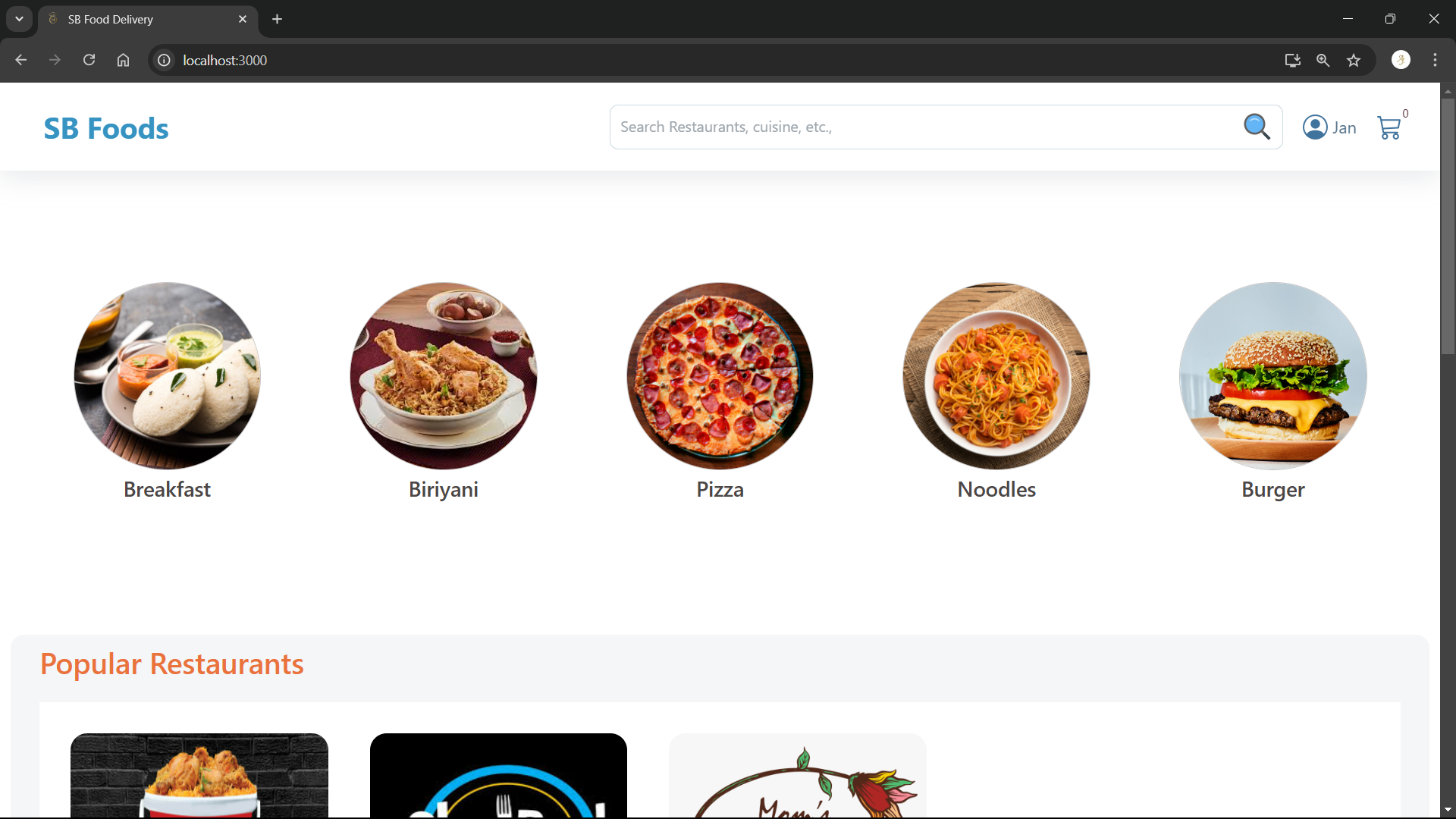
* **Restaurant dashboard**

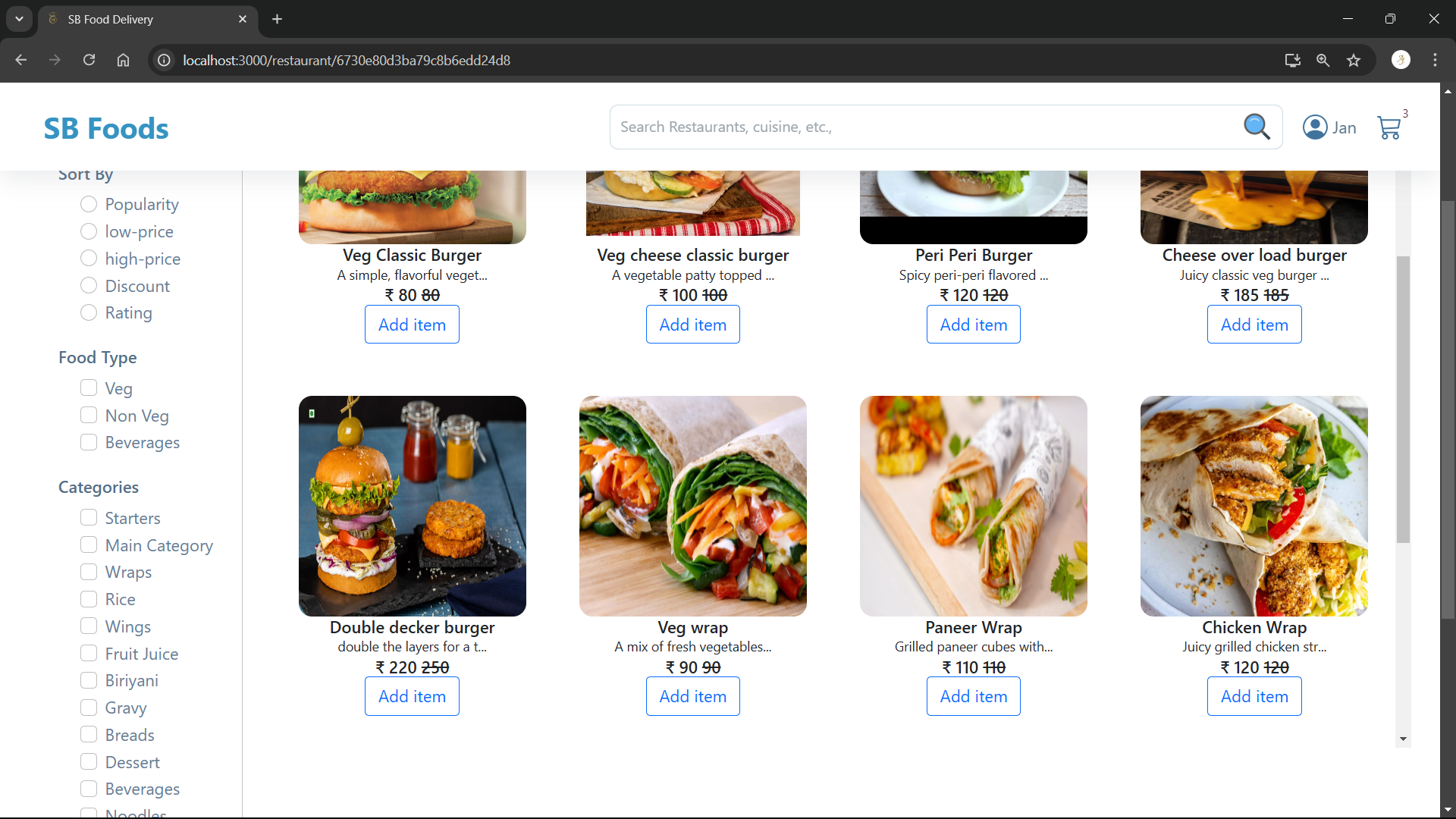
****

* **Restaurant : Add item**

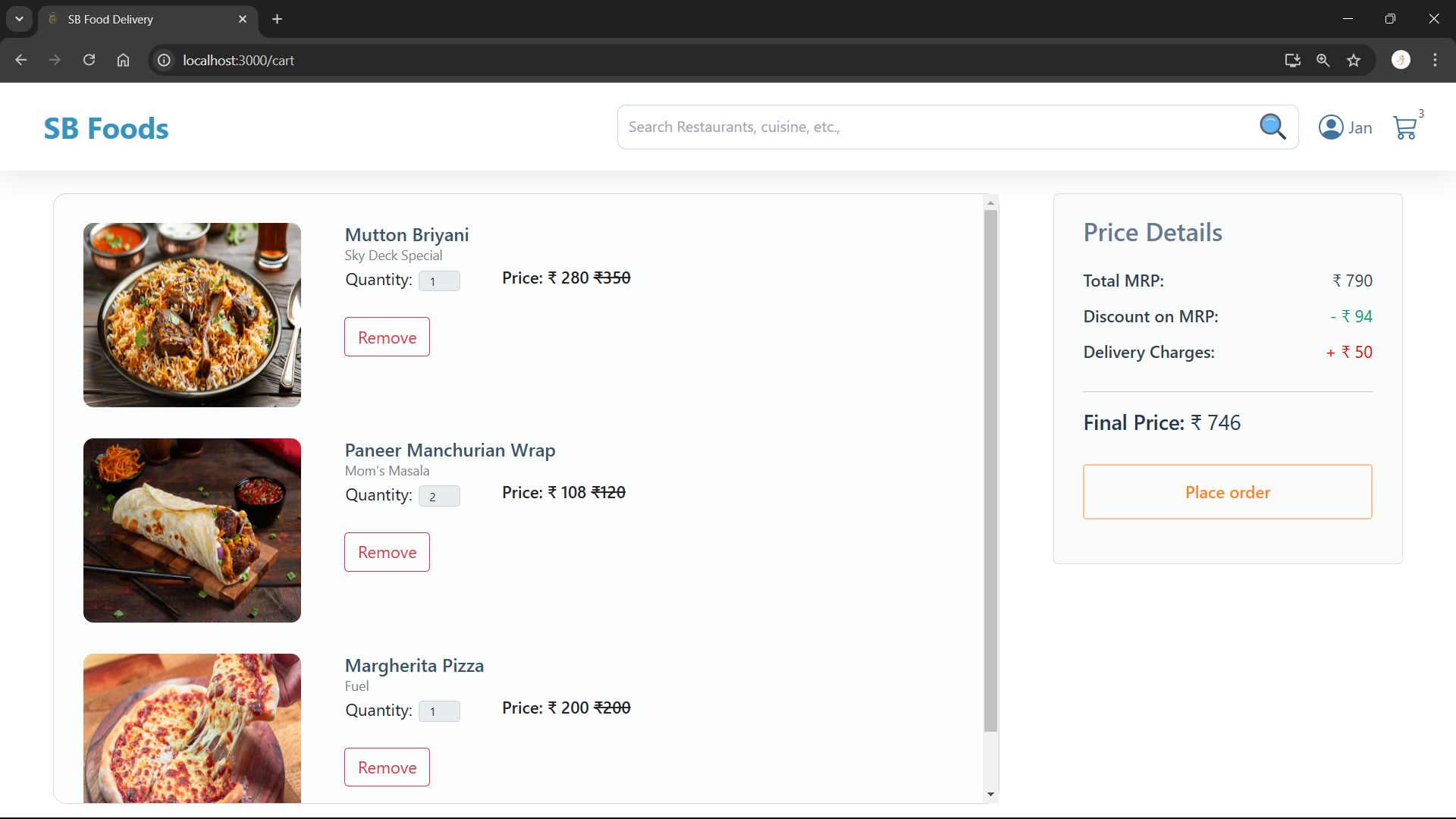
****

* **Customer home page**

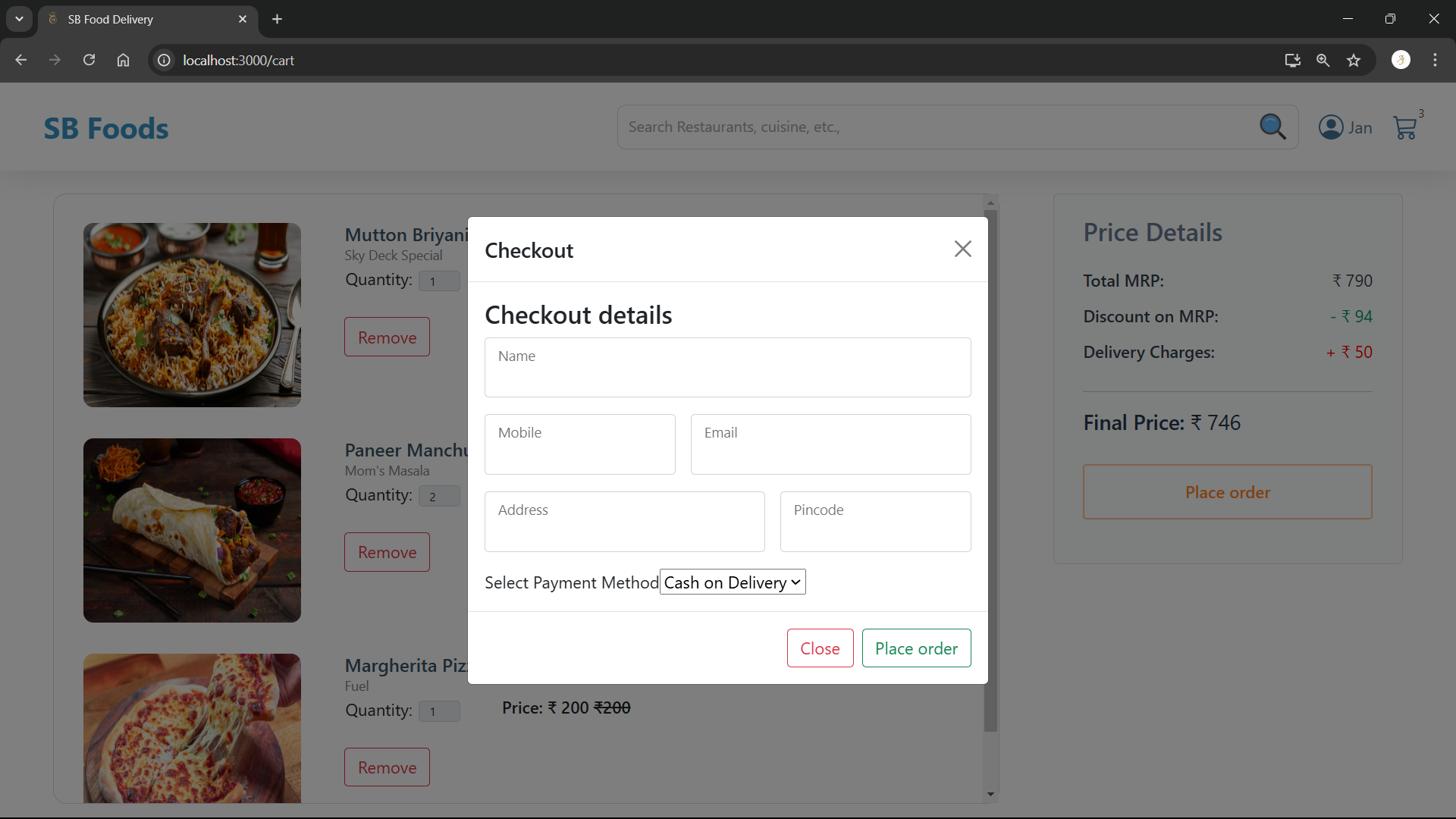


****

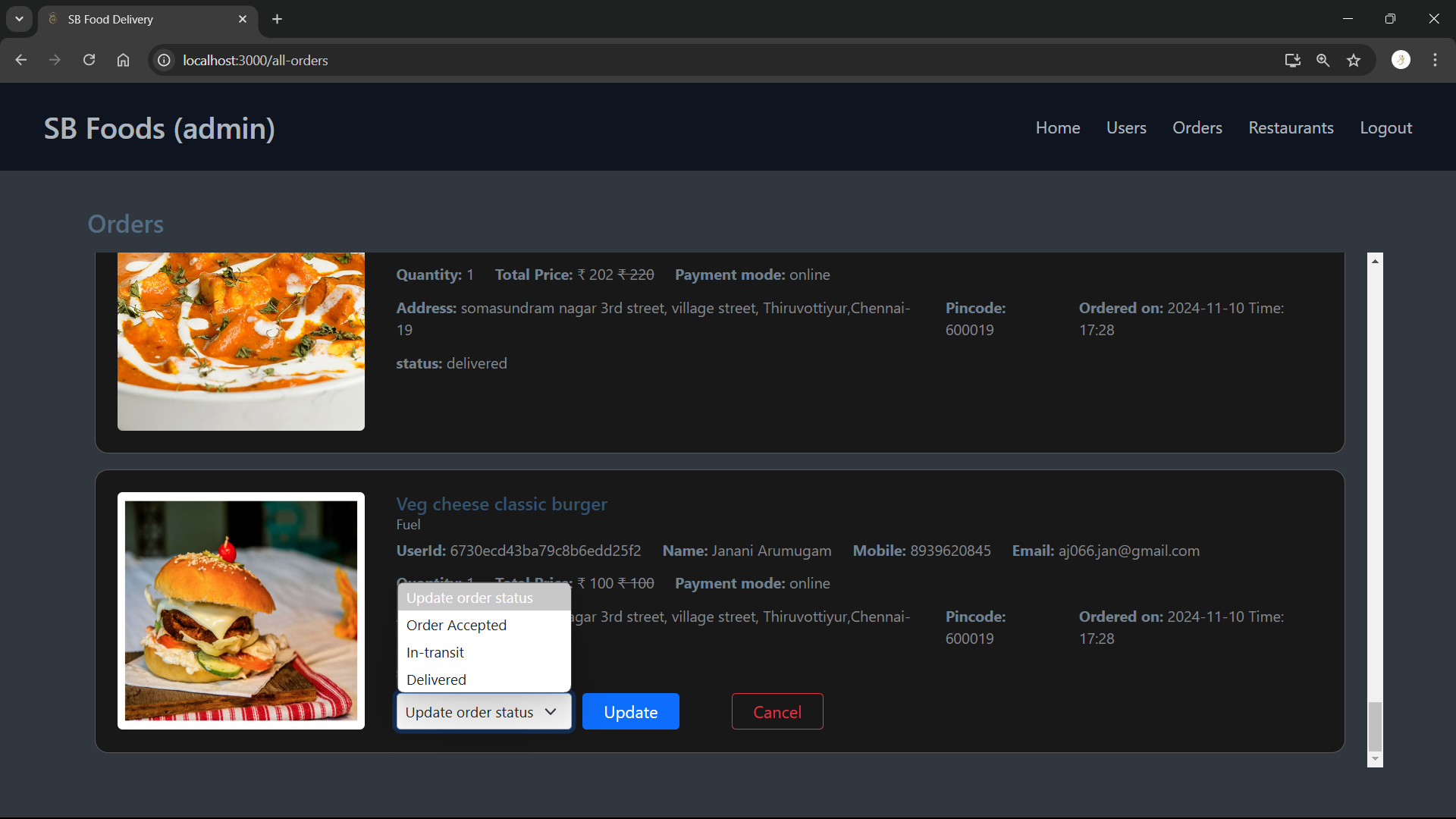
* **Customer cart list**

****

* **Customer checkout details**

****

* **Admin order tracker**

****

**TESTING**

1. **Unit Testing**:
   * **Backend**: Test API with Jest.
   * **Frontend**: Test components with React Testing Library.
2. **Integration Testing**:
   * Test frontend and backend together using Super test.
3. **End-to-End Testing**:
   * Simulate user actions with Cypress.
4. **Validation**:
   * **Backend**: Validate data with Mongoose.
   * **Frontend**: Validate forms using Formik and Yup.
5. **Performance**:
   * Test app performance with JMeter.
6. **Cross-Browser Testing**:
   * Ensure app works on all browsers using Browser Stack.

**FUTURE ENHANCEMENTS**

* Implement automated email notifications for order updates.
* Add real-time chat support for customer service.
* Enhance the search functionality with advanced filters.