SB FOOD ORDERING APP WITH MERN

NAAN MUDHALVAN
A Project Report
Submitted by

JAVAKAR N	210921205021
SELVAM S	210921205047
POOVARASAN M	210921205038
PURUSHOTHAMAN D	210921205039

In partial fulfillment for the award of the degree
Of
BACHELOR OF TECHNOLOGY
In
INFORMATION TECHNOLOGY

LOYOLA INSTITUTE OF TECHNOLOGY, CHENNAI
ANNA UNIVERSITY:CHENNAI 600 025

Project Overview:

Purpose of the App:

- Convenient Meal Ordering:

  Provide users with an intuitive interface for browsing menus, selecting dishes, and placing orders without hassle.
- Support Local Businesses:

  Empower local restaurants by offering them a digital platform to increase visibility and grow their customer base.
- Personalized Experience:

  Leverage AI and user preferences to deliver tailored recommendations, making meal selection effortless.
- Real-Time Tracking:

  Ensure transparency and reliability with features like live tracking of delivery and status updates.
- Cost Savings and Deals:

  Offer exclusive discounts, loyalty programs, and promotions to give users more value with every order.

Features:

- User-Friendly Interface:

  Intuitive design allows users to browse restaurants, view menus, and place orders effortlessly.
- Real-Time Order Tracking:

  Provides live updates on the status of orders, from preparation to delivery, ensuring transparency and reliability.
- Personalized Recommendations:

  AI-powered suggestions based on user preferences, past orders, and trending cuisines.
- Multiple Payment Options:

  Supports a variety of payment methods, including credit cards, digital wallets, and cash-on-delivery, ensuring flexibility for all users.
- Promotions and Rewards:

  Offers exclusive discounts, loyalty rewards, and special deals to enhance user engagement and satisfaction

System Architecture:

Frontend Architecture: React

The frontend of the SB Food Ordering App is designed using **React**, ensuring a dynamic and responsive user interface.

1. **Component-Based Structure:**
   - The app uses reusable React components for modularity and maintainability.
   - Key components include navigation bars, restaurant listings, menus, cart, and order tracking.
2. **State Management:**
   - **Redux/Context API** is used for efficient state management to handle user sessions, cart data, and order status.
3. **Routing:**
   - Implemented with **React Router** for seamless navigation between pages like Home, Menu, Cart, and Profile.
4. **API Integration:**
   - Fetches and updates data via RESTful APIs provided by the backend.
5. **Responsive Design:**
   - Ensures compatibility across devices using **CSS frameworks (e.g., Tailwind CSS, Material-UI)** or custom styling.

Backend Architecture: Node.js and Express.js

The backend is developed using **Node.js** for high performance and scalability, with **Express.js** as the framework.

1. **API Design:**
   - RESTful APIs are used to handle user authentication, order management, and real-time updates.
2. **Middleware:**
   - Middleware is implemented for authentication (e.g., JWT), request validation, and logging.
3. **Error Handling:**
   - Centralized error-handling mechanisms ensure smooth operations and better debugging.
4. **Scalability:**
   - Asynchronous, non-blocking architecture of Node.js allows for handling multiple requests concurrently.

Database: MongoDB

The database is designed to store and retrieve application data efficiently.

1. **Schema Design:**

Users Collection:
```
{ "userId": ObjectId, "name": String, "email": String, "password": String, "address": Array }
```

Restaurants Collection:
```
{ "restaurantId": ObjectId, "name": String, "menu": Array, "location": String }
```

Orders Collection:
```
{ "orderId": ObjectId, "userId": ObjectId, "restaurantId": ObjectId, "items": Array, "status": String }
```

2. **Database Interactions:**
   - **Mongoose ODM** is used for schema validation and database interactions.
   - Queries are optimized using indexing for faster performance.
3. **Relationships:**
   - The database uses references (e.g., userId in Orders) to link collections, maintaining a flexible schema.
4. **Data Security:**
   - User passwords are hashed (e.g., using **bcrypt**) and sensitive information is encrypted to protect against breaches.

Setup Instructions:

1. Prerequisites :

Before setting up the SB Food Ordering App, ensure you have the following software installed:

- **Node.js** (v.16)
- **npm** or **yarn** (Comes with Node.js)
- **MongoDB** (v4.4)
- **Git** (Latest version recommended)
- A code editor (e.g., **VS Code**)

2. Installation Steps:

1. Clone the Repository:

- Open a terminal or command prompt.
- Run the following command to clone the repository from GitHub to your local machine:

git clone https://github.com/username/sb-food-ordering-app.git

- This command creates a copy of the project on your local system.

Navigate to the project directory:

cd sb-food-ordering-app

This ensures you're inside the app's root directory, where both frontend and backend folders are located.

2. Install Dependencies:

To make sure all required packages are installed, you need to install dependencies for both the **backend** and **frontend**.

- **Backend Dependencies:**
  - Go to the **backend** directory:

    cd backend

Install all backend dependencies using npm (Node package manager):

```
npm install
```

This installs all required packages listed in the `package.json` file in the backend directory, such as **Express.js**, **Mongoose**, etc.

- **Frontend Dependencies:**

Navigate to the **frontend** directory:

```
cd ../frontend
```

Run the following command to install frontend dependencies:

```
npm install
```

This installs frontend packages like **React**, **React Router**, and **Axios**, which are necessary for building the user interface and interacting with the backend API.

3. Set Up Environment Variables:

- **Backend:**

In the backend directory, create a `.env` file to store sensitive information like database connection details and JWT secret. Example `.env` content:

```
PORT=5000

MONGO_URI=mongodb://localhost:27017/sbfoodapp

JWT_SECRET=your_secret_key
```

- `PORT=5000` configures the backend server to run on port 5000.
- `MONGO_URI` is the MongoDB connection string. If you're using a local MongoDB setup, this will point to your local database.
- `JWT_SECRET` is used for securing user sessions and authentication tokens.

- **Frontend:**

In the **frontend** directory, create a `.env` file to configure the API URL. Example `.env` content:

```
REACT_APP_API_URL=http://localhost:5000/api
```

- This sets the base URL for API requests from the frontend to the backend.

4. Start MongoDB:

- If you are using **local MongoDB**, ensure MongoDB is running:

Run the following command in your terminal to start the MongoDB server:

```
mongod
```

- This will start the MongoDB instance on your local machine, and the app will connect to it using the `MONGO_URI` set in the `.env` file.

- If using **MongoDB Atlas**, make sure the connection string in the `.env` file reflects the cluster URI from Atlas.

5. Run the Application:

- **Backend Server:**

Navigate to the backend folder:

```
cd backend
```

Start the backend server using:

```
npm start
```

- This will start the Express.js server and begin listening for incoming API requests on `http://localhost:5000`.

- **Frontend Server:**

Now, go to the frontend folder:

```
cd ../frontend
```

Start the frontend server with:

```
npm start
```

        - This will start the React development server, and the app will be accessible via `http://localhost:3000` in your browser.

6. Access the App:

- **Frontend (User Interface):**
  Open a browser and navigate to:
  `http://localhost:3000`
    - This is the interface where users can interact with the app, view restaurants, place orders, etc.
- **Backend API (Server):**
  The backend API, where all the business logic, database queries, and user management are handled, can be accessed at:
  `http://localhost:5000/api`
    - You can test the API endpoints using tools like **Postman** or through the frontend.

5. Folder Structure:

Client: React Frontend Folder Structure:

The React frontend follows a modular structure for scalability and maintainability. Here's the typical organization:

```
frontend/

├── public/

|   ├── index.html              # Main HTML file

|   ├── favicon.ico             # App favicon

|   └── assets/                 # Static assets (images, fonts, etc.)

├── src/

|   ├── components/             # Reusable UI components

|   |   ├── Navbar.jsx          # Navigation bar component

|   |   ├── Footer.jsx          # Footer component

|   |   └── ...                 # Other shared components

|   ├── pages/                  # Pages representing different routes

|   |   ├── Home.jsx            # Homepage layout

|   |   ├── Menu.jsx            # Menu or restaurant listing page

|   |   ├── Cart.jsx            # Shopping cart page
```

```
|   |   └── OrderHistory.jsx   # Order history and tracking page
|   ├── context/               # State management (e.g., Context API, Redux)
|   |   └── CartContext.js     # Context for managing cart state
|   ├── services/              # API calls and business logic
|   |   └── api.js             # Handles API requests to the backend
|   ├── styles/                # CSS or SCSS files for styling
|   |   └── App.css            # Global styles
|   ├── App.js                 # Main app component
|   ├── index.js               # Entry point of the app
|   └── .env                   # Environment variables (API endpoints, etc.)
├── package.json               # Project metadata and dependencies
└── README.md                  # Project documentation
```

Explanation:

- **`components/`:** Houses reusable components to keep the code modular.
- **`pages/`:** Represents application views tied to specific routes.
- **`context/`:** Used for state management across components.
- **`services/`:** Abstracts API calls, ensuring a clean separation of concerns.
- **`styles/`:** Centralized styling files for consistency.

Server: Node.js Backend Folder Structure:

The Node.js backend is organized for clear separation of concerns, using modular routes, controllers, and models.

```
backend/
├── config/
|   └── db.js                # MongoDB connection configuration
├── controllers/             # Handles business logic
|   ├── userController.js    # User-related logic
```

```
|   ├── orderController.js     # Order-related logic
|   └── restaurantController.js # Restaurant-related logic
├── middleware/                # Middleware for authentication, error handling
|   ├── authMiddleware.js      # JWT-based authentication
|   └── errorMiddleware.js     # Error handling middleware
├── models/                    # MongoDB schemas and models
|   ├── User.js                # User model
|   ├── Order.js               # Order model
|   └── Restaurant.js          # Restaurant model
├── routes/                    # API route definitions
|   ├── userRoutes.js          # Routes for user operations
|   ├── orderRoutes.js         # Routes for order operations
|   └── restaurantRoutes.js    # Routes for restaurant operations
├── utils/                     # Utility functions
|   └── generateToken.js       # Helper to generate JWT tokens
├── .env                       # Environment variables (e.g., DB URI, secret keys)
├── server.js                  # Main entry point for the backend server
├── package.json               # Project metadata and dependencies
└── README.md                  # Project documentation
```

Explanation:

- **`config/`:** Centralizes configuration, such as database connection setup.
- **`controllers/`:** Contains logic for handling requests and responses for various features (e.g., users, orders, restaurants).
- **`middleware/`:** Includes custom middleware for authentication and error handling.
- **`models/`:** Defines the structure of MongoDB documents using Mongoose schemas.
- **`routes/`:** Manages API endpoints, separating them by functionality.
- **`utils/`:** Utility functions for repetitive tasks (e.g., token generation).

6. Running the Application

Frontend: Start the React App:

- ★ Navigate to the **frontend** directory:
  `cd frontend`
- ★ Start the React development server:
  `npm start`
- ★ The React app will now start and should automatically open in your default browser at:
  plaintext
  Copy code
  `http://localhost:3000`

This is where users will interact with the app's interface.

Backend: Start the Node.js Server:

- ★ Navigate to the **backend** directory:
  `cd backend`
- ★ Start the backend server:
  `npm start`
- ★ The backend server will now run on port `5000` (or the port defined in your `.env` file) and can be accessed at:
  `http://localhost:5000/api`

This server handles API requests, database interactions, and other backend logic.

Verify Both Servers Are Running:

- Ensure both servers are running simultaneously in separate terminal windows.
- You can now use the app fully:
  - The **frontend** interacts with the user.
  - The **backend** handles requests, authentication, and database queries.

7.API Documentation:

1. User Authentication:

1.1. Register User:

- **Method:** `POST`
- **Endpoint:** `/api/users/register`
- **Description:** Registers a new user.

Request Body:

```
{
  "name": "John Doe",
  "email": "john@example.com",
  "password": "password123"
}
```

Response:

```
{
  "id": "641c3c",
  "name": "John Doe",
  "email": "john@example.com",
  "token": "JWT_TOKEN"
}
```

1.2. Login User

- **Method:** `POST`
- **Endpoint:** `/api/users/login`
- **Description:** Logs in an existing user and returns a token.

Request Body:

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

Response:

```
{
  "id": "641c3c",
  "name": "John Doe",
  "email": "john@example.com",
  "token": "JWT_TOKEN"
}
```

2. Restaurant Management:

2.1. Get All Restaurants:

- **Method:** `GET`
- **Endpoint:** `/api/restaurants`
- **Description:** Retrieves a list of all restaurants.

Response:

```
  {
    "id": "r1",
    "name": "Pizza Palace",
    "cuisine": "Italian",
    "rating": 4.5,
    "menu": [...]
  },
  {
    "id": "r2",
    "name": "Sushi World",
    "cuisine": "Japanese",
    "rating": 4.7,
    "menu": [...]
```

```
  }
```

2.2. Get Restaurant by ID:

- **Method:** `GET`
- **Endpoint:** `/api/restaurants/:id`
- **Description:** Retrieves details of a specific restaurant.

Response:

```
{
  "id": "r1",
  "name": "Pizza Palace",
  "cuisine": "Italian",
  "rating": 4.5,
  "menu": [
    { "id": "m1", "name": "Margherita Pizza", "price": 12.99 },
    { "id": "m2", "name": "Pepperoni Pizza", "price": 14.99 }
  ]
}
```

3. Order Management:

3.1. Place an Order:

- **Method:** `POST`
- **Endpoint:** `/api/orders`
- **Description:** Places a new order.

Request Body:

```
{
  "userId": "641c3c",
  "restaurantId": "r1",
  "items": [
    { "menuItemId": "m1", "quantity": 2 },
    { "menuItemId": "m2", "quantity": 1 }
```

```
  ],
  "totalPrice": 40.97
}
```

Response:

```
{
  "orderId": "o123",
  "status": "Order Placed",
  "estimatedDeliveryTime": "45 mins"
}
```

3.2. Get Order History

- **Method:** `GET`
- **Endpoint:** `/api/orders/history/:userId`
- **Description:** Retrieves the order history of a specific user.

Response:

```
[
  {
    "orderId": "o123",
    "restaurantName": "Pizza Palace",
    "items": [
      { "name": "Margherita Pizza", "quantity": 2 },
      { "name": "Pepperoni Pizza", "quantity": 1 }
    ],
    "totalPrice": 40.97,
    "status": "Delivered"
  },
  {
    "orderId": "o124",
    "restaurantName": "Sushi World",
```

```
  "items": [
    { "name": "California Roll", "quantity": 3 }
  ],
  "totalPrice": 25.50,
  "status": "In Progress"
}
```

4. Miscellaneous:

4.1. Search Restaurants by Cuisine or Name:

- **Method:** `GET`
- **Endpoint:** `/api/restaurants/search`
- **Description:** Searches restaurants by name or cuisine.
- **Query Parameters:**
  - `query` (required): Search term (e.g., "Pizza", "Japanese").

Response:

```
{
  "id": "r1",
  "name": "Pizza Palace",
  "cuisine": "Italian",
  "rating": 4.5
}
```

Authentication Note:

- Protected endpoints (e.g., placing orders, viewing order history) require a **JWT token** in the request headers:
  `Authorization: Bearer <your_jwt_token>`

8. Authentication:

1.Authentication:

Authentication ensures that users are who they claim to be. It involves logging in or registering and validating credentials.

Steps for Authentication:

1. **User Registration:**
   - A new user provides their name, email, and password via the `/api/users/register` endpoint.
   - The password is hashed using **bcrypt** before being stored in the database for security.
2. **User Login:**
   - The user provides their email and password via the `/api/users/login` endpoint.
   - The system checks the email in the database and verifies the provided password against the hashed password using bcrypt.
   - If the credentials are valid, a **JWT token** is generated and returned.
3. **JWT Token:**
   - The token is signed using a secret key (`JWT_SECRET`) defined in the `.env` file.

Example payload of the token:

```
{
  "id": "user_id",
  "email": "john@example.com",
  "iat": 1686582823,
  "exp": 1686586423
}
```

- The token contains user details and has an expiration time (e.g., 1 hour).

Token Storage:

- The token is sent to the client (frontend) and is typically stored in **localStorage** or **HTTP-only cookies** for subsequent requests.

2. Authorization

Authorization ensures that only authenticated users can access certain endpoints and features based on their roles or permissions.

Middleware for Authorization:

1. **Auth Middleware:**
   - The backend includes a custom middleware (`authMiddleware.js`) to validate JWT tokens for protected routes.
   - This middleware:
     - Extracts the token from the `Authorization` header (e.g., `Bearer <token>`).
     - Verifies the token using the secret key.
     - Decodes the payload and attaches the user details to the `req` object for use in the route handlers.

Example:

```
const jwt = require('jsonwebtoken');

const User = require('../models/User');


const protect = async (req, res, next) => {

    let token;

    if (req.headers.authorization &&
req.headers.authorization.startsWith('Bearer')) {

        try {

            token = req.headers.authorization.split(' ')[1];

            const decoded = jwt.verify(token,
process.env.JWT_SECRET);

            req.user = await
User.findById(decoded.id).select('-password');

            next();

        } catch (error) {
```

```
            res.status(401).json({ message: 'Not authorized, token failed' });
        }
    } else {
        res.status(401).json({ message: 'Not authorized, no token' });
    }
};
module.exports = { protect };
```

2. **Role-Based Authorization:**
   - For specific routes (e.g., admin-only actions), additional middleware checks the user's role in the decoded token or user record.

Protected Routes:

- Examples of protected endpoints:
  - Placing an order (`POST /api/orders`)
  - Viewing order history (`GET /api/orders/history/:userId`)

3. Password Security

- **Hashing:**
  - User passwords are hashed using **bcrypt** with a salt value to prevent storage of plaintext passwords.
- **Password Verification:**
  - At login, bcrypt compares the provided password with the hashed one in the database.

4. Benefits of JWT-Based Authentication

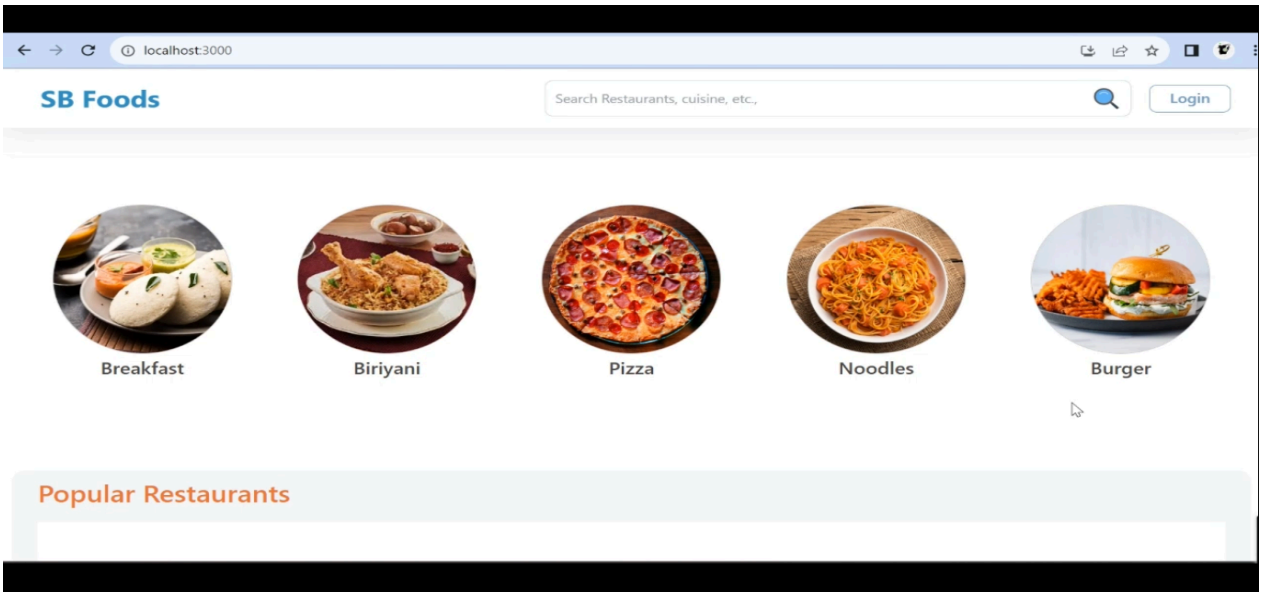
- **Stateless:**
  - Tokens are stateless and do not require session management on the server, making the system scalable.
- **Secure:**
  - Tokens are signed, making them tamper-proof. Sensitive information (like passwords) is never exposed.
- **Portable:**
  - Tokens can be used across domains, enabling flexibility for API integration.

5. Security Enhancements

1. **HTTPS:**
   - Use HTTPS to encrypt all communication between the client and server.
2. **HTTP-Only Cookies:**
   - Store tokens in HTTP-only cookies to reduce the risk of XSS attacks.
3. **Token Expiry:**
   - Ensure tokens have a short expiration time and provide a mechanism to refresh them.
4. **Environment Variables:**
   - Store sensitive keys (e.g., `JWT_SECRET`) in environment variables.

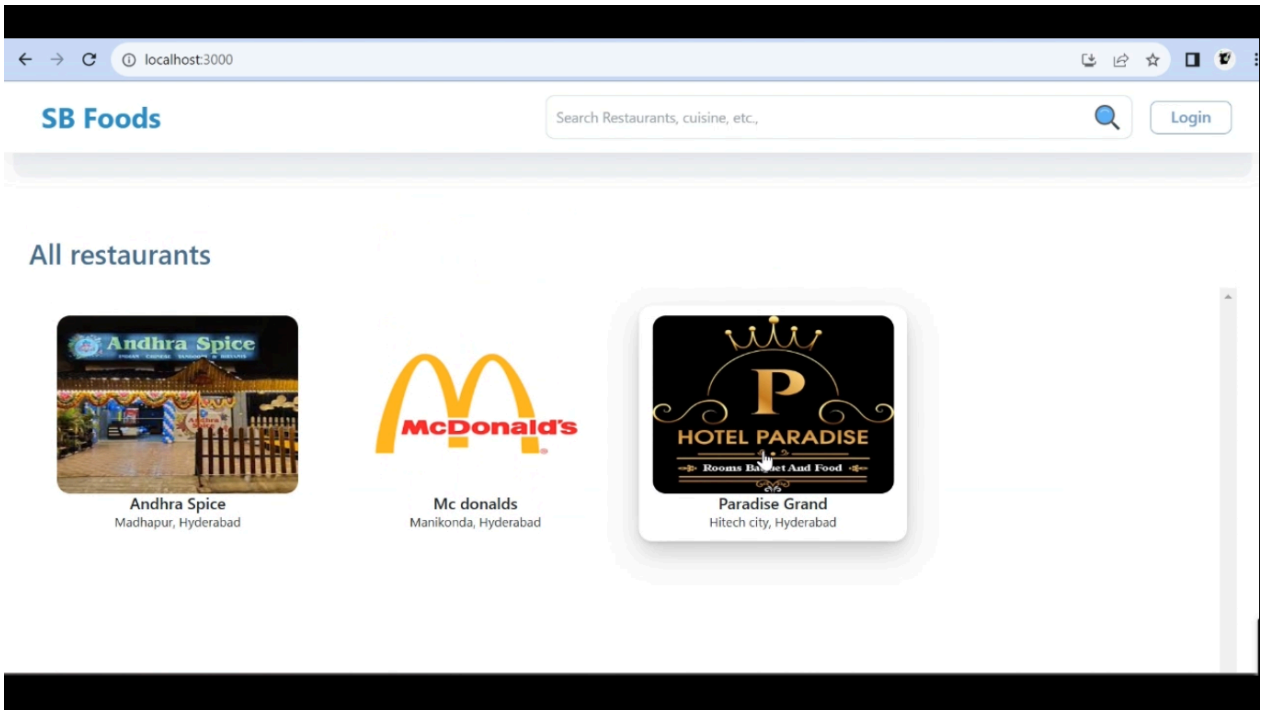
9. User Interface:

Homepage:



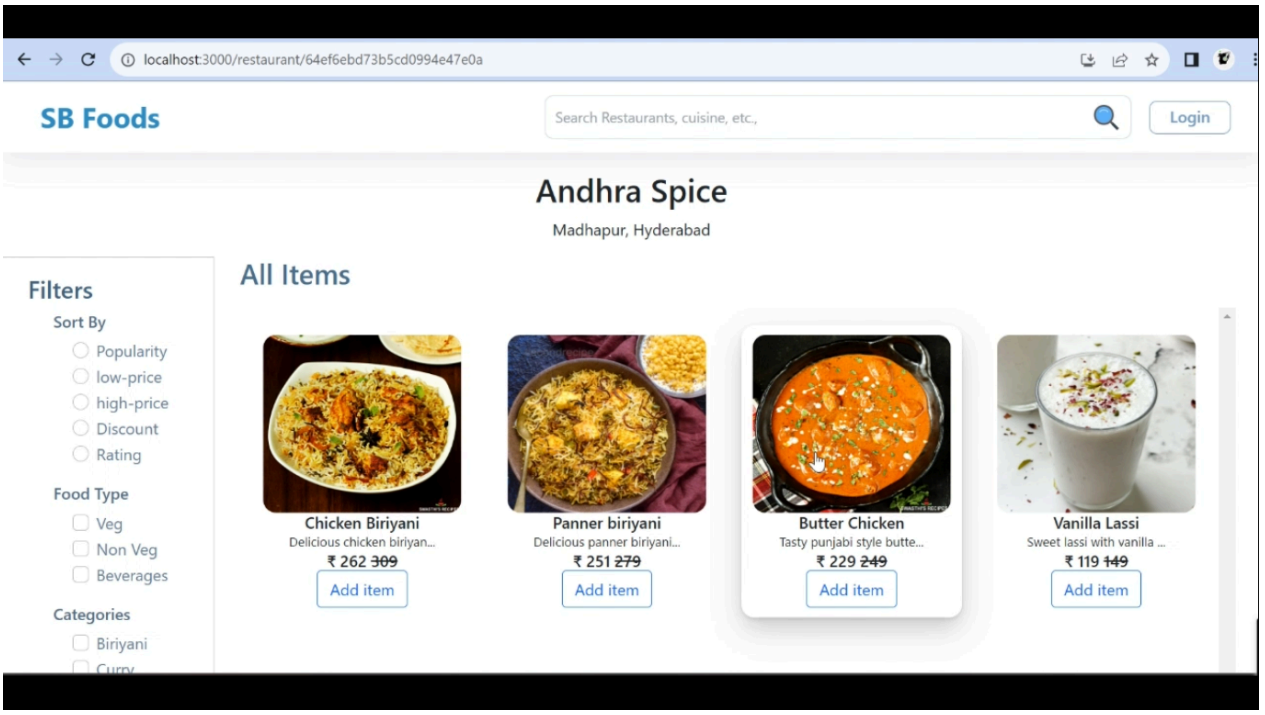
Description: A clean, user-friendly homepage with a search bar and featured items.

Restaurant List:



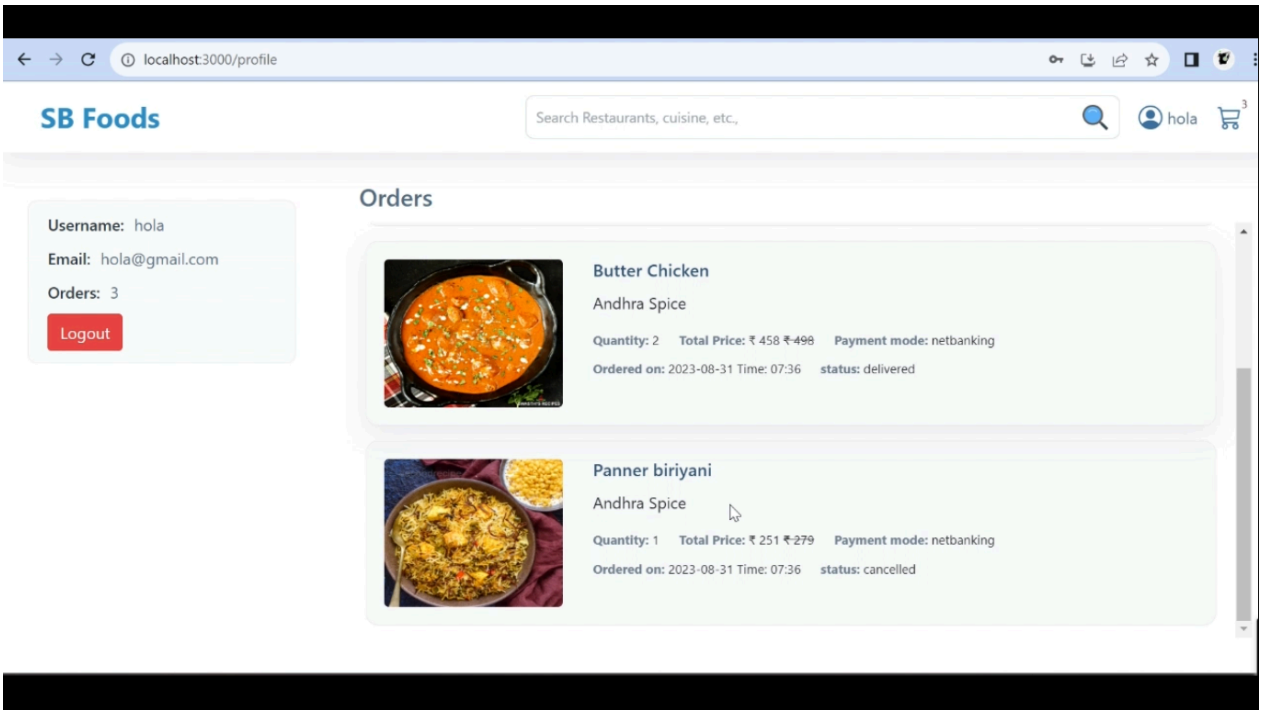
Description: Display of available restaurants with filters and ratings.

Cart:



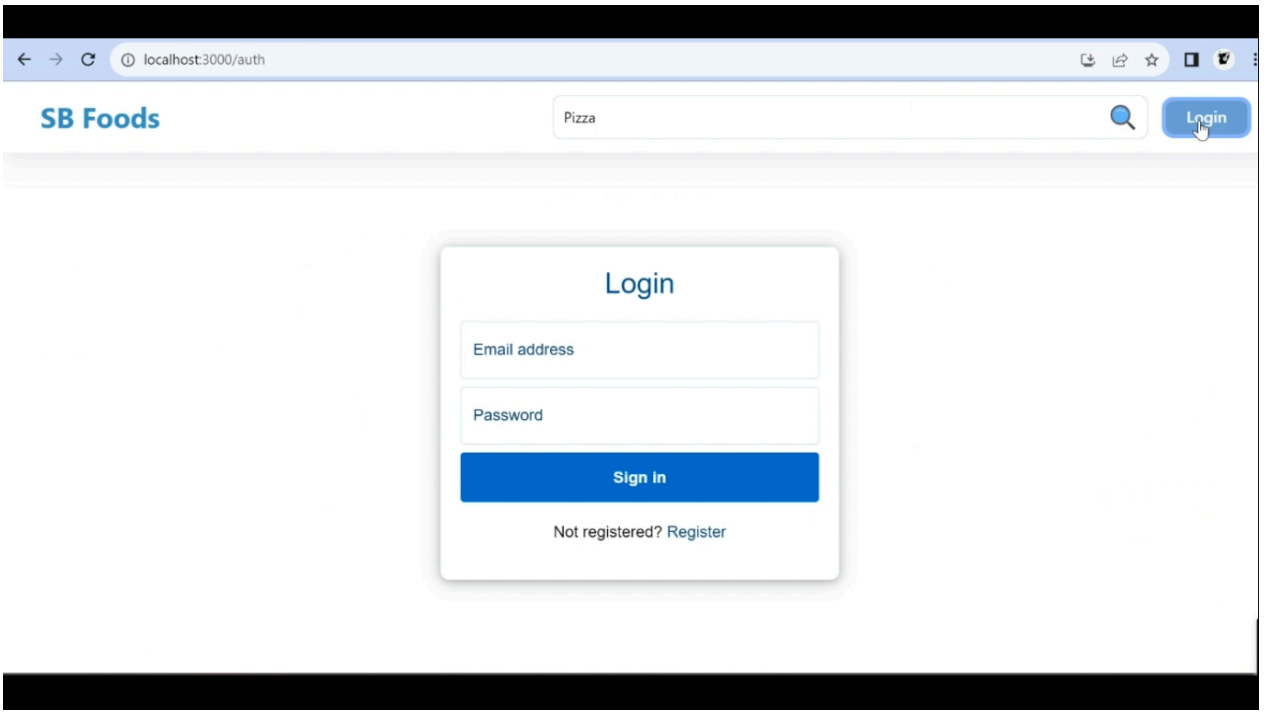
Description: A summary of selected items with options to modify or proceed to checkout.

Order Tracking:

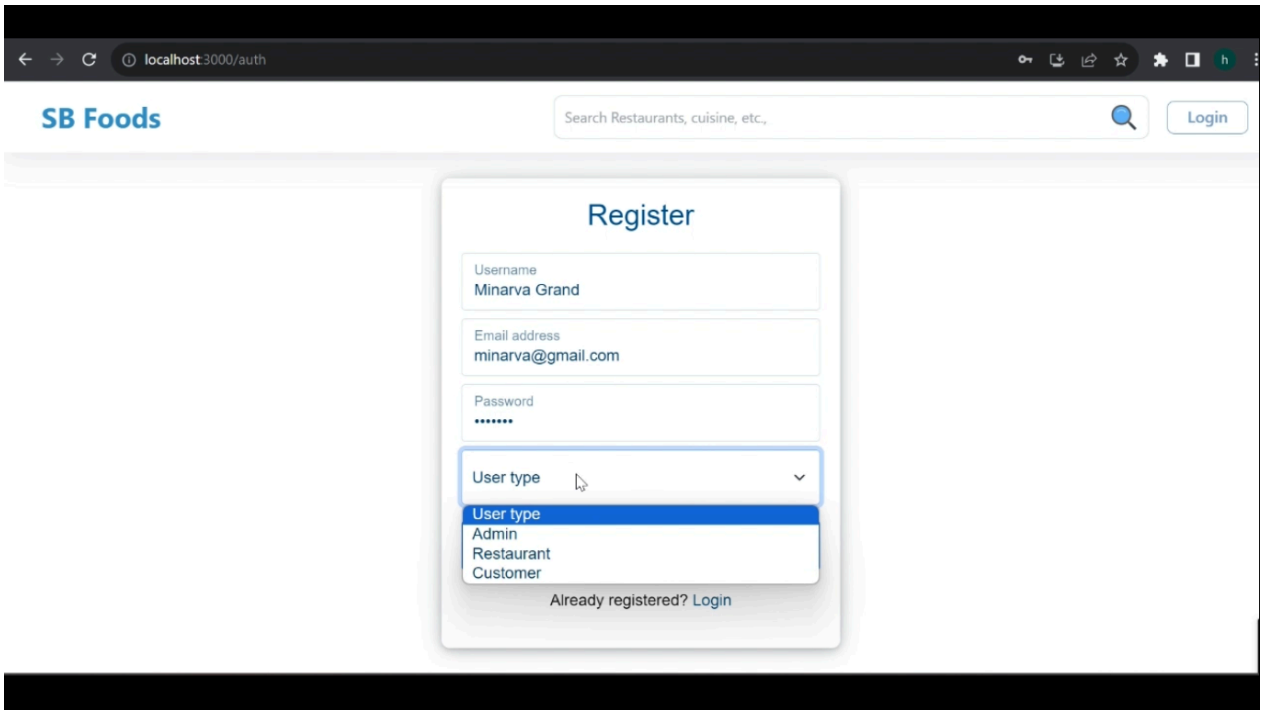


Description: A live update interface showing the status of an ongoing order.

Login page :



Register page:



10. Testing Strategy:

a. Unit Testing:

- **Purpose:** Ensure individual components (e.g., functions, APIs, or services) perform as expected in isolation.
- **Approach:** Test the smallest units of the application, like user login, payment gateway, or cart functionality, to verify that each part works independently.

b. Integration Testing:

- **Purpose:** Verify that different modules or services work together seamlessly.
- **Approach:** Test interactions between front-end components (e.g., UI) and back-end services (e.g., database, payment processing).

c. Functional Testing:

- **Purpose:** Validate the app's core functionalities from the user's perspective, ensuring the app performs tasks like browsing menus, adding items to the cart, and completing the checkout process.

d. UI/UX Testing:

- **Purpose:** Ensure the app's interface is user-friendly and consistent across different devices and screen sizes.
- **Approach:** Test for accessibility, responsiveness, and intuitive design, ensuring a smooth experience for users on both mobile and desktop versions.

e. Security Testing:

- **Purpose:** Ensure the app is secure, protecting user data and transactions.
- **Approach:** Test encryption, authentication, and authorization protocols, and prevent SQL injections, XSS, and other vulnerabilities.

f. Load and Performance Testing:

- **Purpose:** Ensure the app can handle a large number of simultaneous users and requests without crashing or slowing down.
- **Approach:** Test the system under high loads, e.g., during a large order rush, to ensure scalability and performance.

g. Regression Testing:

- **Purpose:** Verify that new updates or features don't break existing functionalities.
- **Approach:** Re-run previously passed test cases to confirm that the system's core functions continue to work after new code is integrated.

h. Acceptance Testing:

- **Purpose:** Confirm that the app meets the business requirements and user expectations.
- **Approach:** Involves real-world scenarios, checking if the app delivers on the promises made in the requirements documentation.

2. Tools Used for Testing:

a. Unit Testing Tools

- **JUnit** (for Java): Used to test Java components.
- **Mockito**: A mock object framework used in Java to simulate interactions with external services or components.
- **Jest**: A JavaScript testing framework used for testing React components, APIs, and other JavaScript functions.

b. Integration Testing Tools

- **Postman**: Helps to test APIs and simulate how back-end services interact with the front-end.
- **Swagger**: A tool for documenting and testing APIs to ensure they are functioning properly.

c. Functional Testing Tools

- **Selenium**: Automates browsers for testing the front-end functionality, including actions like clicks, typing, and navigation.
- **Appium**: Used for automating mobile applications, both Android and iOS, for functional testing.
- **Cypress**: A testing framework focused on front-end integration testing with easy setup and fast execution.

d. UI/UX Testing Tools

- **BrowserStack**: Allows testing across different browsers and devices for cross-platform compatibility.
- **Responsinator**: Checks how the app looks across different screen sizes to ensure responsiveness.
- **UserTesting**: Provides feedback from real users for UI/UX testing and improvement.

e. Security Testing Tools

- **OWASP ZAP (Zed Attack Proxy)**: A penetration testing tool to find security vulnerabilities.
- **Burp Suite**: Another tool for vulnerability scanning, particularly for web applications.

f. Load and Performance Testing Tools:

- **JMeter**: A popular tool for load testing and measuring performance under heavy usage.
- **LoadRunner**: Provides performance testing for large-scale applications.
- **New Relic**: Used for monitoring the application's performance in real-time.

g. Continuous Integration/Continuous Deployment (CI/CD):

- **Jenkins**: Automates the testing and deployment process, integrating with various testing tools for continuous testing.
- **CircleCI**: Another CI tool used to automatically run test cases and deploy the app.

11. Known Issues:

1. Functional Issues:

- **Duplicate Orders Issue**:
  On rare occasions, refreshing the order confirmation page results in duplicate orders being placed.
  Status: Under investigation.
  Workaround: Avoid refreshing the page after placing an order.
- **Cart Synchronization Delay**:
  Changes made to the cart (e.g., adding or removing items) take several seconds to reflect

on the checkout page.
Status: Planned fix in the next release.
Workaround: Wait a few seconds before navigating to checkout.

- **Promo Code Errors**:
  Some valid promo codes are not recognized by the system, especially during peak hours.
  Status: Under review.
  Workaround: Contact support to manually apply the discount.

2. UI/UX Issues

- **Button Overlap on Smaller Screens**:
  On devices with smaller screens, the “Place Order” button partially overlaps the “Modify Order” option.
  Status: Fix scheduled for next sprint.
  Workaround: Use landscape mode or larger screens.
- **Slow Loading Menu Images**:
  High-resolution food item images take time to load on slower networks.
  Status: Optimization is in progress.
  Workaround: Use Wi-Fi or a high-speed mobile connection.

3. Performance Issues

- **High Load Latency**:
  During peak hours, the app experiences slow response times when loading the menu or checkout.
  Status: Performance tuning underway.
  Workaround: Place orders during off-peak hours if possible.
- **Push Notifications Delay**:
  Notifications (e.g., order updates) are sometimes delayed by up to 10 minutes.
  Status: Under investigation.

4. Security Issues

- **Payment Gateway Timeout**:
  Payment processing fails intermittently when using certain third-party wallets.
  Status: Ongoing discussions with the payment gateway provider.
  Workaround: Retry the payment or use alternative payment methods.
- **Session Expiry Bug**:
  User sessions are not consistently timing out after inactivity, posing potential security risks.
  Status: Fix under testing.

5. Miscellaneous Issues

- **Order History Missing Entries**:
  Some past orders are not appearing in the order history for certain users.
  Status: Data migration issue identified and being resolved.
- **Location Services Accuracy**:
  The app occasionally selects incorrect delivery addresses when auto-detecting locations.
  Status: Enhancements planned.
  Workaround: Manually enter the delivery address.

Guidance for Developers

- **Log Reports**: Enable detailed logging in the staging environment to capture real-time error data.
- **Testing Focus**: Prioritize testing on payment gateway integration and cart functionality in upcoming cycles.
- **User Feedback Monitoring**: Actively review user complaints to detect additional edge cases.

12. Screenshots or Demo:

Demo video link:https://drive.google.com/file/d/1RJzLnoh63AlDz6dUwKgoZcZq9fA9gZwX/view?usp=drivesdk

13. Future Enhancements:

1. Enhanced User Experience:

- Personalized Recommendations:
  Use AI/ML to recommend food items based on user preferences, order history, and current trends.
  Benefit: Increases user engagement and order value.
- Voice-Assisted Ordering:
  Integrate voice assistants like Siri, Alexa, or Google Assistant to allow hands-free order placement.
  Benefit: Makes the app more accessible for differently-abled users.
- Dark Mode:
  Add a dark mode feature for users who prefer it, especially for nighttime use.
  Benefit: Improves usability and reduces eye strain.

2. Improved Functionality:

- Multi-Cart Feature:
  Allow users to place multiple orders from different restaurants simultaneously.
  Benefit: Adds convenience for users ordering for groups or events.
- Scheduled Ordering:
  Enable users to place orders in advance for a specific time and date.
  Benefit: Increases flexibility for busy users or event planning.
- In-App Wallet:
  Implement an SB wallet to store funds and provide cashback incentives.
  Benefit: Improves transaction speed and enhances user loyalty.
- Table Reservation Feature:
  Add an option for users to reserve tables at restaurants directly through the app.
  Benefit: Expands app utility beyond delivery and takeout.

3. Performance and Scalability

- Progressive Web App (PWA) Development:
  Develop a PWA version of the app to improve accessibility for users on low-end devices or unstable networks.
  Benefit: Expands user base without requiring app downloads.
- Faster Loading Times:
  Optimize backend APIs and implement CDN solutions for image and data caching.
  Benefit: Reduces app latency, especially in high-traffic scenarios.
- Offline Mode:
  Allow users to browse the menu and pre-fill orders even when offline. Orders can sync once the user reconnects to the internet.
  Benefit: Increases usability in low-connectivity areas.

4. Gamification and Rewards

- Loyalty Program:
  Introduce a tiered rewards system where users earn points for every order, redeemable for discounts or free items.
  Benefit: Boosts customer retention.
- Challenges and Badges:
  Add fun challenges like “Order from 5 different cuisines this month” and award digital badges.
  Benefit: Encourages more frequent usage.

5. Advanced Features

- Real-Time Order Tracking:
  Enhance live tracking with a more detailed map interface and estimated delivery time updates.
  Benefit: Improves transparency and user satisfaction.
- Augmented Reality (AR) Menu:
  Allow users to view AR previews of menu items to better understand portion sizes and presentation.
  Benefit: Reduces order confusion and increases satisfaction.
- Subscription Plans:
  Offer subscription options for regular users with perks like free delivery, exclusive discounts, or premium customer support.
  Benefit: Adds a recurring revenue stream.

6. Security Enhancements

- Biometric Authentication:
  Enable fingerprint or facial recognition login for improved account security.
  Benefit: Adds convenience and security for users.
- End-to-End Encryption:
  Upgrade data transmission protocols to ensure maximum security for sensitive data, including payment information.
  Benefit: Builds trust among users.

7. Partner and Vendor Features

- Vendor Dashboard:
  Develop a comprehensive dashboard for restaurant partners to manage menus, orders, and analytics.
  Benefit: Streamlines operations for vendors and improves accuracy in menu updates.
- Driver App Enhancements:
  Add features like optimized delivery routes and earnings analytics for delivery personnel.
  Benefit: Improves delivery efficiency and partner satisfaction.

8. Community and Feedback Integration

- Customer Reviews and Ratings:
  Allow users to rate individual menu items and provide feedback directly to restaurants.
  Benefit: Improves food quality and helps other customers make informed decisions.
- Social Media Sharing:
  Enable users to share their orders or favorite meals on social platforms.
  Benefit: Enhances brand visibility and engagement.