

# GOJAN SCHOOL OF BUSINESS AND TECHNOLOGY

ANNA UNIVERSITY CHENNAI-600052



## TO DEVELOP A PROJECT ON

**FOOD ORDERING APP USING (MERN stack by MongoDB)**

A PROJECT REPORT

*Submitted by*

**SURESH KUMAR D - 110521104049**

**STELLA L - 110521104047**

**SOUNTHARIYAN S - 110521104046**

**SOUMYA SHIBU - 110521104045**

*In partial fulfilment for the award of the degree*

*Of*

**BACHELOR OF ENGINEERING**

**IN**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

In association with **Smart Internz** and **Naan Mudhalvan**



## **TABLE OF CONTENT**

### **INTRODUCTION**

### **PROJECT OVERVIEW**

### **EXECUTIVE SUMMARY**

### **LITERATURE REVIEW**

1. Existing Solutions and Market Analysis
2. Technologies in Food Ordering Applications
3. Advantages of Using the MERN Stack for the Food Ordering Application
4. Security and Data Privacy
5. Scalability and Future-Proofing

### **SCENARIO-BASED CASE STUDY**

### **SYSTEM REQUIREMENTS**

1. Functional Requirements
  - 1.1 Menu Management and Orders
  - 1.2 Search and Filter Options
  - 1.3 Dashboard for Restaurant Owners and Admins
  - 1.4 Order Details and User Interaction
2. Non-Functional Requirements
  - 2.1 Performance
  - 2.2 Scalability
  - 2.3 Security
  - 2.4 Usability and Accessibility

2.5 Reliability and Availability

2.6 Maintainability

3. Technology Requirements

3.1 Frontend

3.2 Backend

3.3 Database

3.4 Deployment and Hosting

3.5 Testing and Quality Assurance

## **SYSTEM ARCHITECTURE**

1. MERN Stack Architecture

2. COMPONENT DIAGRAM

## **DATABASE DESIGN**

1. E.R DIAGRAM

2. APPLICATION OVERVIEW

## **BACKEND DEVELOPMENT**

1. Setting Up the Project

2. Folder Structure

3. Database Setup (MongoDB)

4. Controllers (Business Logic)

5. Routes (API Endpoints)

6. Authentication (Middleware)

7. Starting the Server

## 8. PROJECT STRUCTURE

### **FRONTEND DEVELOPMENT**

1. Setting Up the React Project
2. Core Components
3. User Pages
4. State Management and Context
5. API Integration and Services
6. Protected Routes and Authorization
7. UI/UX Design
8. Testing and Debugging

### **USER INTERFACE DESIGN**

1. Landing Page / Home Page
2. Search Results Page
3. Restaurant Details Page
4. User Registration and Login Pages
5. User Dashboard
6. Add/Edit Menu Page (for Restaurant Owners)

### **KEY FEATURES & FUNCTIONALITY**

1. For Customers
2. For Restaurant Owners:
3. For Admin:
4. Search and Filtering Capabilities

## 5. Menu Listings

### **DASHBOARD FOR RESTAURANT OWNER, CUSTOMER AND ADMIN**

1. OWNER
2. ADMIN
3. CUSTOMER

### **DEPLOYMENT FOR BACKEND & FRONTEND**

1. GIT EXPLANATION
  - Tracking and Versioning Code
  - Distributed System
  - Branches and Merging
  - Commit History and Blame Tracking
  - Undoing Changes
  - Staging Area
  - Why Git is Important
2. BACKEND DEPLOYMENT
3. FRONTEND DEPLOYMENT

### **USER AUTHENTICATION AND AUTHORIZATION**

### **TESTING AND QUALITY ASSURANCE**

1. Requirements Analysis and Test Planning
2. Functional Testing
3. User Interface (UI) and Usability Testing

4. Performance Testing
5. Security Testing
6. Database Testing
7. Regression Testing
8. Non-Functional Testing
9. Continuous Integration (CI) and Continuous Deployment (CD)

## **CHALLENGES AND SOLUTIONS**

1. Scalability
2. Data Consistency
3. User Authentication and Security
4. Search and Filtering Performance
5. Cross-Platform Compatibility
6. User Experience (UX)
7. Payment Processing
8. Maintaining Code Quality
9. Deployment and DevOps

## **KNOWN ISSUES**

## **FUTURE ENHANCEMENT**

## **CONCLUSION**

## **REFERENCE**

# INTRODUCTION

## Team Members (Roles and Responsibilities):

### 1. SURESH KUMAR D – Team Leader & Full Stack Developer

- Oversaw the development process and ensured seamless integration between the frontend, backend, and database.
- Contributed to both frontend and backend development, debugging, and deployment.

### 2. SOUNTHARIYAN S – Frontend Developer

- Designed and implemented the React-based user interface.
- Focused on creating reusable components, integrating APIs, and ensuring responsiveness.

### 3. STELLA L – Backend Developer

- Developed the Node.js and Express.js server, including API endpoints for user and admin functionalities.
- Managed business logic for order processing and authentication workflows.

### 4. SOUMYA SHIBU – Frontend Developer

- Assisted in creating UI components and enhancing user experience through optimized styling and interactions.
- Contributed to client-side logic for handling cart management and API integration.

## PROJECT OVERVIEW

### Purpose:

The Food Ordering Application is designed to bridge the gap between restaurant owners and customers by simplifying the food ordering process and optimizing order management. It provides a robust platform where customers can browse menus, place orders, and track their delivery status, while restaurant admins can manage menus, process orders, and analyze performance. This project ensures a seamless and efficient user experience for both ends of the food ordering system.

## **Features:**

### **User Features:**

#### **1. Register and Log In:**

- Users can create accounts, securely log in, and manage their profiles.
- The system uses JWT-based authentication to ensure security.

#### **2. Browse Menu:**

- Customers can view a categorized menu with food items, complete with descriptions, images, and prices.
- Categories (e.g., Appetizers, Mains, Desserts) allow users to navigate easily.

#### **3. Cart Management:**

- Add or remove items from the cart.
- Adjust item quantities with real-time price updates.

#### **4. Place Orders:**

- Users can proceed to checkout and confirm their orders.
- Payment integration is planned for future updates.

#### **5. Order Tracking:**

- Customers can view their past orders and check the status of current ones.

### **Admin Features:**

#### **1. Menu Management:**

- Admins can add, update, or remove items from the menu.
- They can upload images and set availability statuses for each item.

#### **2. Order Management:**

- Admins have access to a real-time dashboard displaying customer orders.
- They can update order statuses (e.g., Pending, In Progress, Delivered).



### 3. Analytics (Future Enhancement):

- Planned features include sales insights and user behaviour analysis.
- This overview highlights the application's dual focus on improving the customer experience while empowering restaurant admins with efficient tools for management.

## EXECUTIVE SUMMARY :

The Food Ordering Application is a modern web-based platform developed using the MERN stack (MongoDB, Express.js, React.js, Node.js) to streamline the food ordering process. This application aims to bridge the gap between restaurant owners and customers by providing an intuitive, efficient, and secure platform for menu listing, searching, and order management.

## LITERATURE REVIEW:

The Food Ordering Application using the MERN stack builds upon the success and limitations of existing food ordering platforms and leverages modern technologies to meet the evolving needs of both restaurant owners and customers. This review examines existing solutions, discusses the role of technology in food ordering, and highlights the benefits of using the MERN stack (MongoDB, Express.js, React.js, and Node.js) for this application.

**1. Existing Solutions and Market Analysis:** Several well-known platforms like Uber Eats, DoorDash, Grubhub, and Zomato dominate the food ordering space, each offering unique features to cater to their user base. While these platforms have introduced significant innovation, they often come with limitations and challenges:

- **Uber Eats and DoorDash** provide robust delivery options but may have high service fees and limited control for restaurant owners over the customer experience.
- **Grubhub** excels in a broad range of restaurant choices but can face challenges with delivery times and service quality.
- **Zomato** offers comprehensive restaurant listings and reviews but may lack advanced features for order management and customer engagement.

These platforms address specific segments of the food ordering market but often overlook the need for flexible order management and personalized customer interactions. Additionally, most of these platforms rely on monolithic or hybrid architectures, which can be difficult to scale and adapt to new user demands. The Food Ordering Application aims to fill this gap by combining robust order management

features with a clean, user-friendly design tailored to both restaurant owners and customers.

## **2. Technologies in Food Ordering Applications:**

The role of technology in food ordering has grown substantially, with a shift from traditional phone-based orders to highly interactive, data-driven web applications. According to research in FoodTech, digital solutions enhance customer engagement, streamline transactions, and improve data accessibility for all stakeholders. Key technological advancements have made it possible for applications to support real-time order updates, complex search functionality, and enhanced data security.

Technologies like cloud computing, NoSQL databases, and JavaScript frameworks allow applications to handle large-scale data operations efficiently, making them ideal for food ordering solutions. Additionally, web technologies like REST APIs and microservices provide flexible data flow between the front end and back end, enabling faster user interactions and seamless application scaling. The adoption of JavaScript frameworks, especially those that support Single Page Applications (SPAs), has greatly improved the user experience by reducing loading times and providing more dynamic, interactive interfaces.

However, a common challenge in food ordering applications is the need for real-time data synchronization to reflect order status instantly, ensuring that customers see only accurate, up-to-date information. Furthermore, security and user privacy have become essential, particularly with respect to handling personal data for orders, payments, and account management.

## **3. Advantages of Using the MERN Stack for the Food Ordering Application**

The MERN stack (MongoDB, Express.js, React.js, and Node.js) is well-suited for building a dynamic, data-intensive, and scalable application like a Food Ordering platform.

Below are key advantages of each component and how they contribute to this application's unique needs:

- **MongoDB:** As a NoSQL database, MongoDB stores data in a JSON-like format, making it ideal for handling unstructured and varied data typical in menu listings (e.g., images, descriptions, prices, and user information). MongoDB's schema-less nature allows easy adaptation and scalability, which is useful as new menu items or categories are added. Its ability to handle large data volumes ensures the platform can scale with more users and orders over time.
- **Express.js:** This lightweight web application framework provides a robust routing system, which is essential for handling various endpoints such as menu

listings, user profiles, and order functionalities. Express.js facilitates seamless communication between the front end and MongoDB, efficiently managing data flows while maintaining a secure, API-driven architecture.

- **React.js:** React is a powerful JavaScript library for building user interfaces, particularly Single Page Applications (SPAs). In this Food Ordering Application, React allows the creation of a responsive, interactive UI that enhances the user experience by providing fast, dynamic interactions with minimal loading times. React's component-based architecture makes it easy to develop and manage reusable UI elements, such as menu cards, search bars, and user dashboards. Furthermore, React's support for state management and integration with libraries like Redux makes it ideal for managing complex data flows and user interactions across multiple components.
- **Node.js:** As a runtime environment that executes JavaScript on the server side, Node.js enables the Food Ordering Application to be entirely JavaScript-based, which improves development consistency and performance. Node.js is also asynchronous and event-driven, allowing it to handle multiple simultaneous requests efficiently, which is critical for a real-time application like this one. Node's non-blocking architecture is particularly beneficial for a platform that requires high performance and scalability, as it can handle requests from a large number of users concurrently.

#### 4. Security and Data Privacy

Given the nature of food ordering applications, security and data privacy are essential. The MERN stack provides several ways to enforce security, from using JWT (JSON Web Tokens) for secure user authentication to employing encrypt for password hashing. MongoDB Atlas, the managed database service for MongoDB, also offers end-to-end encryption and advanced security protocols, ensuring user data protection.

With sensitive data like payment information, contact details, and personal information often being exchanged, secure authentication and authorization methods are vital. JWT tokens in the Food Ordering Application ensure that only authenticated users can access sensitive routes, while MongoDB's access control and encryption features protect data at rest and in transit. This is crucial in establishing user trust and meeting legal requirements for data protection.

#### 4. Scalability and Future-Proofing

One of the significant advantages of the MERN stack is its ability to support rapid scaling and future-proofing. As the user base and data grow, MongoDB's distributed nature can scale horizontally across multiple servers, ensuring consistent performance. Node.js, with its asynchronous event handling, allows the application to manage more requests without requiring additional computational resources. React's modular, component-based structure also allows for easy updates and new feature integration without overhauling the existing codebase.

The MERN stack thus enables continuous integration and deployment, allowing developers to quickly add features or respond to user feedback, making it adaptable for future demands. This adaptability is critical for a platform like a Food Ordering Application, which may require rapid updates in response to market trends, new functionalities, or expanded geographies.

## SCENARIO-BASED CASE STUDY :

1. **User Registration:** Alice, who is looking to order food online, downloads your food ordering app and registers as a Customer. She provides her email and creates a password.
2. **Browsing Menus:** Upon logging in, Alice is greeted with a dashboard showcasing available restaurant menus. She can see listings with detailed descriptions, photos, and prices. She applies filters to narrow down her search, specifying her desired cuisine, price range, and dietary preferences.
3. **Order Inquiry:** Alice finds a dish she likes and clicks on it to get more information. She sees the meal details and restaurant's contact information. Interested in ordering, Alice fills out a small form with her details and places the order.
4. **Order Confirmation:** The restaurant receives Alice's inquiry and reviews her details. Satisfied, the restaurant confirms Alice's order. Alice receives a notification that her order is confirmed, and the status in her dashboard changes to "pending restaurant confirmation."
5. **Admin Approval (Background Process):** In the background, the admin reviews new restaurant registrations and approves legitimate users who want to add menus to the app.
6. **Owner Management:** Bob, a restaurant owner, signs up for an Owner account on the app and submits a request for approval. The admin verifies Bob's credentials and approves his Owner account.

7. **Menu Management:** With his Owner account approved, Bob can now add, edit, or delete menu items in his account. He updates the status and availability of his dishes based on inventory.
8. **Platform Governance:** Meanwhile, the admin ensures that all users adhere to the platform's policies, terms of service, and privacy regulations. The admin monitors activities to maintain a safe and trustworthy environment for all users.
9. **Transaction and Payment:** Once Alice's order is confirmed, she and the restaurant finalize the payment details within the app, ensuring transparency and security.
10. **Delivery Process:** Alice successfully receives her ordered food, marking the completion of the order process facilitated by the food ordering app. This scenario highlights the main functionalities of your MERN-based food ordering app, including user registration, menu browsing, order inquiry and confirmation process, admin approval, owner management, platform governance, and the overall order transaction.

## SYSTEM REQUIREMENTS :

The **Food Ordering Application** requires a robust, scalable, and secure setup to handle diverse user roles and a dynamic data environment. Below are the detailed system requirements, broken down into functional and non-functional requirements.

### 1. Functional Requirements

**1. Functional Requirements** The functional requirements outline essential features and operations of the application, focusing on what the system should perform to meet the needs of customers, restaurant owners, and administrators.

#### 1.1 User Management and Authentication

- **Registration and Login:** Users (customers, restaurant owners, and admins) should be able to register and log in securely.
- **Role-Based Access Control (RBAC):** The application should distinguish between three types of users—customer, restaurant owner, and admin—with specific permissions for each role:
  - **Customer:** Can browse and view menu listings, place orders, and save favorite dishes.
  - **Restaurant Owner:** Can add, update, and delete menu listings, view analytics, and respond to customer inquiries.

- **Admin:** Has full control over user management, menu listings, and the ability to monitor platform usage.

## 1.2 Menu Listings and Management

- **Menu Addition and Editing:** Restaurant owners should be able to add new menu items with details like name, description, price, ingredients, images, and category (appetizer, main course, dessert, etc.).
- **Listing Updates:** Owners should be able to edit and update menu items as needed.
- **Menu Deletion:** Owners can remove a menu item if it is no longer available.

## 1.3 Search and Filter Options

- **Search Functionality:** Customers should be able to search for menu items using keywords.
- **Filtering Options:** Allow filtering by price range, cuisine type, dietary preferences, and availability.
- **Sorting Options:** Enable sorting by price, newest listings, and popularity.

## 1.4 Dashboard for Restaurant Owners and Admins

- **Owner Dashboard:** Restaurant owners should have access to a dashboard displaying menu listings, analytics (such as views and orders), and contact information for interested customers.
- **Admin Dashboard:** Admins should be able to view platform statistics, manage users and listings, monitor content for quality, and handle reports or complaints.

## 1.5 Menu Details and User Interaction

- **Menu Detail Page:** Display detailed menu information, images, and contact details for the restaurant.
- **Favorites and Bookmarking:** Customers should be able to save menu items for future reference.
- **Contact and Inquiry System:** Customers should be able to contact restaurant owners directly via an inquiry form.

## 2. Non-Functional Requirements

The non-functional requirements detail the system's quality attributes, including performance, scalability, security, and usability.

### 2.1 Performance

- **Response Time:** The application should have a response time of less than 2 seconds for user actions like login, search, and filtering.
- **Data Loading Speed:** Food and restaurant listings and images should load quickly and efficiently, especially on the homepage and search results.
- **Concurrent Users:** The system should support at least 1,000 concurrent users, with the ability to scale up as needed.

### 2.2 Scalability

- **Horizontal Scalability:** The application should be able to scale horizontally to handle an increasing number of users and menu listings.
- **Data Scalability:** MongoDB's sharding and replication features should be used to scale the database as the number of menu items and users grows.
- **Microservices Support:** The application should be designed to support a microservices architecture in the future, allowing features to be independently developed and scaled.

### 2.3 Security

- **User Authentication:** Implement secure user authentication using JWT (JSON Web Tokens) and bcrypt for password hashing.
- **Data Encryption:** Sensitive data, such as passwords and user contact details, should be encrypted in transit and at rest.
- **Role-Based Authorization:** Ensure that only users with the correct permissions can access specific functionalities (e.g., only admins can delete user accounts).
- **Data Validation and Sanitization:** Prevent SQL injection, XSS attacks, and other security vulnerabilities by validating and sanitizing all user inputs.

### 2.4 Usability and Accessibility

- **User-Friendly Interface:** Provide a simple, clean, and intuitive interface that makes it easy for customers and restaurant owners to navigate and perform actions.

- **Responsive Design:** Ensure the application is fully responsive and accessible on desktops, tablets, and mobile devices.
- **Accessibility Standards:** Follow WCAG guidelines to make the application accessible to users with disabilities, including support for screen readers and keyboard navigation.

## 2.5 Reliability and Availability

- **Data Backups:** Implement regular data backups for MongoDB to ensure data recovery in case of failures.
- **Server Uptime:** The application should have at least 99.9% uptime, with provisions for handling downtime and failures.
- **Fault Tolerance:** Ensure redundancy in critical components, especially for the database, to minimize downtime and prevent data loss.

## 2.6 Maintainability

- **Code Modularity:** Follow modular coding practices with separate components for frontend and backend, making it easier to update or replace features.
- **Documentation:** Maintain comprehensive documentation for code, APIs, and deployment processes, allowing future developers to understand and extend the application efficiently.
- **Automated Testing:** Implement automated testing for key features to ensure functionality during updates or changes to the codebase.

## 3. Technology Requirements

### Frontend

- Framework: React.js
- UI Libraries: Material-UI or Bootstrap for reusable components and styling
- State Management: Redux or Context API for managing global state
- API Requests: Axios or Fetch for handling HTTP requests to the backend

### Backend

- Runtime: Node.js for the server environment
- Framework: Express.js for handling routing and middleware



- Authentication: JSON Web Tokens (JWT) for user authentication
- Data Validation: Joi or validator.js for server-side data validation

## Database

- Database: MongoDB for flexible, schema-less data storage
- Deployment: MongoDB Atlas for a managed, cloud-based solution
- ORM/ODM: Mongoose for modelling and managing MongoDB data in Node.js

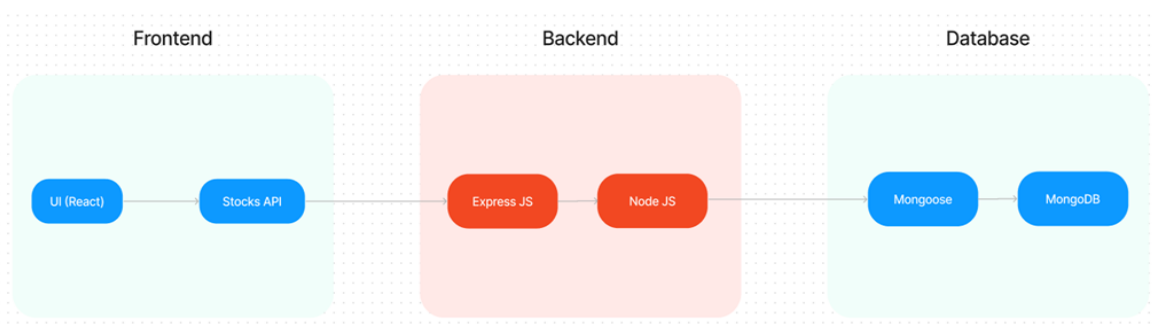
## Deployment and Hosting

- Cloud Provider: MongoDB for hosting the application
- Frontend Hosting: Vercel or Netlify for hosting the frontend React application
- Continuous Integration/Continuous Deployment (CI/CD): GitHub Actions or Jenkins for automating build and deployment processes

## Testing and Quality Assurance

- Frontend Testing: Jest and React Testing Library for unit and integration tests
- **Backend Testing:** Mocha and Chai for API testing
- **End-to-End Testing:** Cypress or Selenium to test full workflows

## SYSTEM ARCHITECTURE :



- The technical architecture of our Food Ordering app follows a client-server model, where the frontend serves as the client and the backend acts as the server. The frontend encompasses not only the user interface and presentation but also incorporates the axios library to connect with the backend easily by using RESTful APIs.
- The frontend utilizes the Bootstrap and Material UI library to establish a real-time and better UI experience for any user, whether it is an admin, restaurant owner, or ordinary user.

- On the backend side, we employ the Express.js framework to handle the server-side logic and communication.
- For data storage and retrieval, our backend relies on MongoDB. MongoDB allows for efficient and scalable storage of user data, including user profiles, menu items, and orders. It ensures reliable and quick access to the necessary information.
- Together, the frontend and backend components, along with Axios, Express.js, and MongoDB, form a comprehensive technical architecture for our Food Ordering app. This architecture enables real-time communication, efficient data exchange, and seamless integration, ensuring a smooth and immersive food ordering experience for all users.

## MERN Stack Architecture

The **MERN Stack** is a popular JavaScript stack used for building dynamic web applications. It comprises **MongoDB**, **Express.js**, **React.js**, and **Node.js**. These four technologies work together to allow developers to build complete, end-to-end applications entirely in JavaScript, making development more efficient and cohesive. Below is a detailed breakdown of each component and how they interact within the architecture of a MERN application.

The application is built using the following layered architecture:

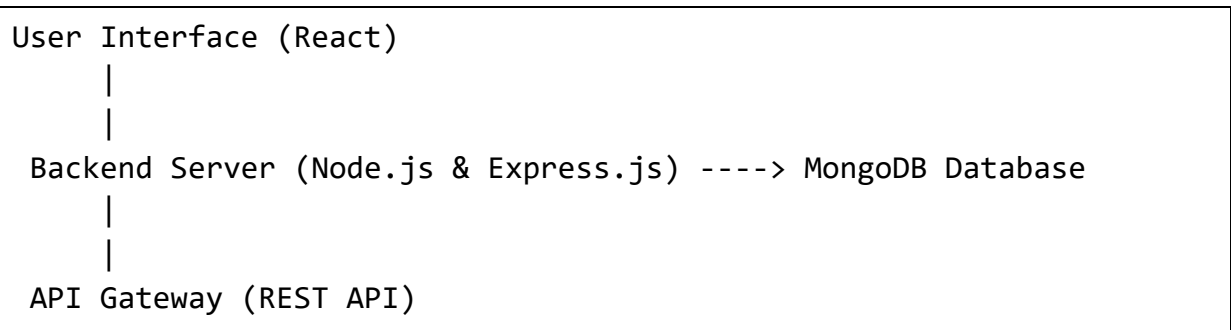
1. **Frontend** (React): Handles the client-side logic, rendering UI components, and managing the application state.
2. **Backend** (Node & Express): Handles REST API requests, authentication, and business logic.
3. **Database** (MongoDB): Stores user data, property listings, and favorite properties.

### ❖ Key Benefits of MERN Architecture

- **End-to-End JavaScript:** MERN allows for consistent use of JavaScript across the entire stack, simplifying development and making it easier to share code between frontend and backend.
- **Scalability and Performance:** MongoDB and Node.js are built for scalable, high-performance applications. MongoDB's distributed database model and Node's non-blocking I/O operations enable efficient handling of high traffic and large data loads.

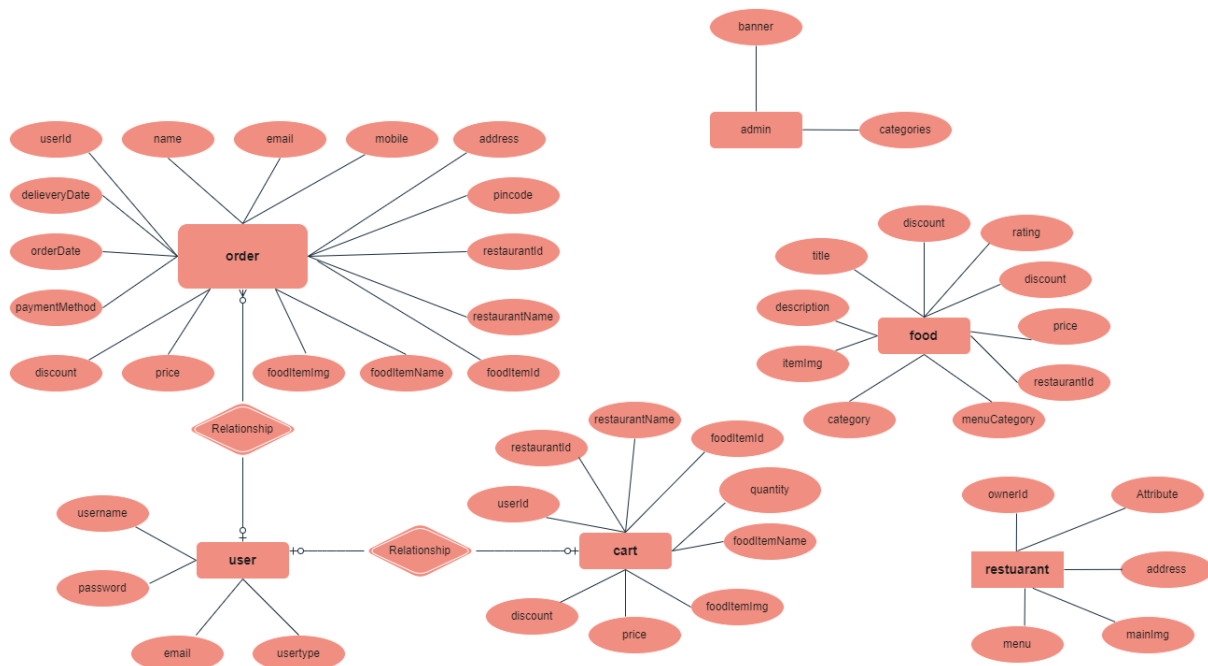
Below is a sample report structure for the **Food Ordering Application** using the MERN stack, incorporating images and the technical structure we discussed.

- COMPONENT DIAGRAM :



## DATABASE DESIGN :

### E.R DIAGRAM :



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

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

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

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

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

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

## APPLICATION OVERVIEW :

- **Database:**  
MongoDB for storing data.
- **Authentication:**  
JWT for secure authentication
- **Deployment:**  
GitHub

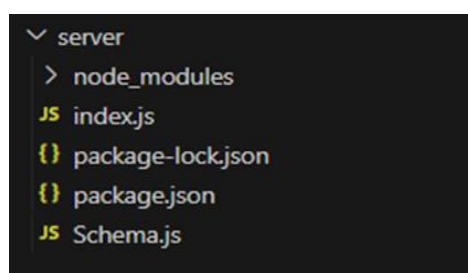
Here are the tools required for developing a full-stack application using Node.js, Express.js, MongoDB, React.js:

- ❖ **Node.js and npm:** Node.js is a powerful JavaScript runtime environment that allows you to run JavaScript code on the server-side. It provides a scalable and efficient platform for building network applications.
- ❖ **Express.js:** Express.js is a fast and minimalist web application framework for Node.js. It simplifies the process of creating robust APIs and web applications, offering features like routing, middleware support, and modular architecture. Install Express.js, a web application framework for Node.js, which handles server-side routing, middleware, and API development.
- ❖ **MongoDB:** MongoDB is a flexible and scalable NoSQL database that stores data in a JSON-like format. It provides high performance, horizontal scalability, and seamless integration with Node.js, making it ideal for handling large amounts of structured and unstructured data.
- ❖ **React.js:** React.js is a popular JavaScript library for building user interfaces. It enables developers to create interactive and reusable UI components, making it easier to build dynamic and responsive web applications. Install React.js, a JavaScript library for building user interfaces.
- ❖ **HTML, CSS, and JavaScript:** Basic knowledge of HTML for creating the structure of your app, CSS for styling, and JavaScript for client-side interactivity is essential.

- ❖ **Database Connectivity:** Use a MongoDB driver or an Object-Document Mapping (ODM) library like Mongoose to connect your Node.js server with the MongoDB database and perform CRUD (Create, Read, Update, Delete) operations.
- ❖ **Front-end Framework:** Utilize React-Js to build the user-facing part of the application, including entering booking room, status of the booking, and user interfaces for the admin dashboard. For making better UI we have also used some libraries like material UI and Boot Strap.
- ❖ **Version Control:** Use Git for version control, enabling collaboration and tracking changes throughout the development process. Platforms like GitHub or Bitbucket can host your repository.
- ❖ **Development Environment:** Choose a code editor or Integrated Development Environment (IDE) that suits your preferences, such as Visual Studio Code, Sublime Text, or WebStorm.

## BACKEND DEVELOPMENT :

The backend for this project using the MERN stack is responsible for handling data storage, business logic, authentication, and API endpoints that serve data to the frontend. This section outlines how to set up and implement the backend using **Node.js** with **Express.js** as the framework, **MongoDB** as the database, and **Mongoose** for data modeling.



- This is the structure of backend that consists of JAVASCRIPT and JSON packages. These structure mainly based on configuration , controllers , middleware , node\_modules that consists of modules to design and make the web page more effective , routes, schemas of javascript, uploads of version controls like “GIT”.
- The structure for this backend services and packages is given below the diagrammatic representation .

Here's a simplified breakdown:

## 1. Setting Up the Project

- **Initialize the Project:** Start by creating a folder for the project, then initialize it as a Node.js project. Install required libraries like Express (for server), Mongoose (to work with MongoDB), JWT (for user authentication), and Bcrypt (to encrypt passwords).
- **Environment Configuration:** Use a .env file to securely store sensitive data, like your MongoDB connection string and JWT secret key.

## 2. Folder Structure

### 1. node\_modules/

- This directory contains all the Node.js modules and dependencies installed via npm (Node Package Manager). When you run npm install, the necessary packages are downloaded and placed in this folder.

### 2. index.js

- This is the main entry point of the application. It usually contains the code to initialize the Express server, connect to the MongoDB database, set up middleware, and define routes. Essentially, it acts as the central hub that brings all components together to run the application.

### 3. package-lock.json

- This file is automatically generated by npm and contains the exact versions of all installed packages and their dependencies. It ensures that the same versions are used when the project is installed on different environments, helping maintain consistency and preventing version conflicts.

### 4. package.json

- This file is crucial for any Node.js project. It contains metadata about the project, including the project name, version, description, main file, scripts, and a list of dependencies required by the project. It is used by npm to manage the project, handle dependencies, and run scripts.

## 3. Database Setup (MongoDB)

- **Connection:** Use Mongoose to connect to MongoDB, which will store user details, menu items, and more.
- **Data Models:** Define schemas for data storage. For example, a User model has fields like name, email, password, and role (customer or owner). A Menu model stores details about food items like name, description, price, and restaurant.

#### 4. Controllers (Business Logic)

- **User Controller:** Handles user actions like registration (stores encrypted password) and login (validates user and returns a JWT token).
- **Menu Controller:** Handles creating, reading, and updating menu listings. Only authenticated users can create listings.

#### 5. Routes (API Endpoints)

- **User Routes:** Routes handle requests for user actions, like /register and /login, which go to userController functions.
- **Menu Routes:** Routes like /menus enable users to view, create, or manage menu listings.

#### 6. Authentication (Middleware)

- **JWT Middleware:** JWT (JSON Web Token) is used for secure authentication. Middleware checks if a user has a valid token before accessing protected routes

#### 7. Starting the Server

- **Express Server:** In the main file, index.js, initialize Express, connect to MongoDB, set up middleware, and define routes. The server listens on a specified port, ready to handle API requests.

## 8.1 PROJECT STRUCTURE :

Organize the backend project into a structured format for scalability and maintainability:

food-ordering-server/

```
├── node_modules/    # Node.js modules
├── .env             # Environment variables
├── index.js         # Entry point of the application
├── package-lock.json # Package lock file
└── package.json     # Package configuration file
```

This folder structure outlines the organization of a Node.js project, including the dependencies (node\_modules), configuration (environment variables in .env), the main entry point for the application (index.js), and the files that manage project dependencies and their versions (package.json and package-lock.json). This setup ensures a well-organized, maintainable, and scalable development environment.



## FRONTEND DEVELOPMENT :



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

Here's a brief explanation of the main directories and files:

- src/components: Contains components related to the application such as, register, login, home, etc.,
- src/pages has the files for all the pages in the application.

The frontend of a **Food Ordering Application** using the **MERN stack** is built with **React.js** to create a dynamic, user-friendly interface. React handles the **view layer** of the application, providing responsive pages and components that display property listings, handle user inputs, and interact with backend APIs. Here's a breakdown of the main components involved in frontend development for this type of application.

## 1. Setting Up the React Project

- **Initialize the Project:** Create a new React project using Create React App or another preferred setup method.
- **Install Dependencies:** Add necessary libraries such as Axios for API requests, React Router for routing, Redux for state management (optional), and Material-UI or Bootstrap for styling.
- **Environment Configuration:** Use a .env file to store environment variables like API endpoints and keys securely.

## 2. Core Components

- **Header:** Contains navigation links and user authentication options (login, signup).
- **Footer:** Provides links to contact information, privacy policies, and social media.
- **MenuItem:** Displays individual food items with details like name, description, price, and image.
- **MenuList:** Renders a list of MenuItem components.
- **OrderSummary:** Shows the details of the user's order before checkout.
- **Cart:** Manages and displays items added to the user's cart.

## 3. User Pages

- **Home Page:** Showcases featured restaurants, popular dishes, and offers.
- **Menu Page:** Displays the full menu of a selected restaurant with filtering and sorting options.
- **Restaurant Page:** Provides detailed information about the restaurant, including its menu, reviews, and location.
- **Cart Page:** Shows the items in the user's cart and provides an option to proceed to checkout.
- **Checkout Page:** Allows the user to enter payment information and confirm their order.
- **User Profile:** Displays user information, order history, and options to update details or logout.

## 4. State Management and Context

- **Global State:** Use Context API or Redux to manage global state, including user authentication status, cart items, and order details.
- **Local State:** Use React's useState and useReducer hooks for local state management within components.
- **Authentication Context:** Manages the user's authentication state and provides methods for login, logout, and signup.

## 5. API Integration and Services

- **API Configuration:** Set up Axios or Fetch to handle API requests to the backend.
- **Service Layer:** Create a service layer to manage API calls for user authentication, menu retrieval, order processing, and payment handling.
- **Error Handling:** Implement global error handling to catch and display errors from API calls gracefully.

## 6. Protected Routes and Authorization

- **Protected Routes:** Use React Router to create protected routes that only authenticated users can access, such as the profile or checkout pages.
- **Authorization:** Check user roles (e.g., admin, customer) and grant access to specific functionalities based on their role.

## 7. UI/UX Design

- **Responsive Design:** Ensure all pages are responsive and work well on various screen sizes using CSS frameworks like Bootstrap or Material-UI.
- **User-Friendly Interface:** Focus on creating a clean, intuitive interface that makes it easy for users to navigate, search for food, and place orders.
- **Accessibility:** Follow best practices for accessibility, including ARIA roles, keyboard navigation, and screen reader compatibility.

## 8. Testing and Debugging

- **Unit Testing:** Write unit tests for individual components using testing libraries like Jest and React Testing Library.
- **Integration Testing:** Test interactions between multiple components to ensure they work together as expected.
- **End-to-End Testing:** Use tools like Cypress to simulate real user interactions and test the entire application flow from start to finish.

- **Debugging:** Utilize browser developer tools, console logging, and React DevTools to debug and resolve issues during development.

## USER INTERFACE DESIGN:

Designing the User Interface (UI) for a food ordering app involves creating a clean, intuitive, and visually appealing experience that allows users to seamlessly browse, search, and manage food orders. In this app, React.js is used as the front-end framework to build the dynamic UI. Here's an outline of the key pages and components in the UI design, along with their functionalities and layout.

### 1. Landing Page / Home Page

- **Hero Section:** Features a large banner image related to food, with a prominent search bar to look for restaurants or dishes based on location, cuisine, and price.
- **Food Categories:** Quick links to different types of cuisines (e.g., Italian, Chinese, Indian) or meal types (e.g., breakfast, lunch, dinner).
- **Featured Restaurants:** A section showcasing popular or recently listed restaurants with thumbnails, brief descriptions, and ratings.
- **How It Works Section:** Short explanation of how the app works, such as browsing menus, placing orders, and tracking delivery.
- **Footer:** Contains links to About, Contact, and Support pages, as well as social media icons and copyright information.

### 2. Search Results Page

- **Search Filters:** Provides options to filter by cuisine, price range, ratings, and other preferences. Filters are usually displayed in a sidebar or above the results.
- **Restaurant Listings:** Displays a list or grid of restaurants based on the search criteria, each with a small image, name, location, cuisine type, and a "View Menu" button.
- **Map View Option:** Option to switch to a map view where users can see restaurant locations on a map. Each restaurant can be represented by a pin on the map for easy navigation.

### 3. Restaurant Details Page

- **Image Gallery:** A carousel of restaurant images to give users a detailed look at the establishment.

- **Restaurant Information:** Displays the restaurant's name, description, menu items, ratings, and hours of operation.
- **Menu Items:** List of available dishes with images, descriptions, and prices.
- **Order Section:** Includes a button to add items to the cart and proceed to checkout.
- **Contact Information:** Allows the user to contact the restaurant for questions or additional details.
- **Map Section:** Displays a small map with the restaurant location for reference.

#### 4. User Registration and Login Pages

- **Login Form:** Form for users to log in using their email and password. Optionally, social login (Google, Facebook) can be added.
- **Registration Form:** Form to create a new account with fields for name, email, password, and role (customer or restaurant owner).
- **Password Reset Option:** Link to reset the password, which sends a reset link to the user's email.

#### 5. User Dashboard

- **For Customers:**
  - **Saved Restaurants:** Displays restaurants that the user has bookmarked for later viewing.
  - **Order History:** Shows the list of previous and current orders, including order dates, restaurant details, and total cost.
- **For Restaurant Owners:**
  - **Manage Menu:** Shows a list of dishes the owner has listed. Owners can add, edit, or delete menu items.
  - **Order Requests:** Displays any orders placed by customers, including the option to accept or decline orders.
- **User Profile Settings:** Allows users to update their personal information, contact details, and password.

#### 6. Add/Edit Menu Page (for Restaurant Owners)

- **Menu Form:** Form to input all necessary information about the dish, such as name, description, price, and category.

- **Image Upload:** Section to upload multiple images of the dish, which will be displayed in the image gallery.
- **Categories Selection:** Options for selecting the dish category, such as appetizer, main course, dessert, etc.
- **Save Changes Button:** Allows the owner to save the information and add it to their menu, or update an existing item.

The UI design for the food ordering app in the MERN stack combines essential features for easy menu browsing, efficient ordering, and user account management. React enables a responsive and modular component-based UI, making the interface scalable and interactive.

## KEY FEATURES & FUNCTIONALITY:

### For Customers:

- **View Listings:** Browse all available restaurant menus.
- **Filter Options:** Narrow down listings by cuisine, price, and rating.
- **Dish Details:** Access in-depth information about dishes, including images, description, and price.
- **Favorite Listings:** Save favorite dishes or restaurants for later reference.

### For Restaurant Owners:

- **Menu Management:** List menu items with detailed descriptions and images.
- **Edit & Delete Listings:** Update or remove menu items as needed.

### For Admin:

- **User Management:** Manage all users on the platform.
- **Menu Management:** Access and manage all menu listings.

### Other Capabilities:

#### 1. Search and Filtering Capabilities

- **Search Bar:** Allows users to search for dishes or restaurants by keywords.
- **Filter Options:** Users can filter dishes or restaurants by various criteria, such as:
  - Price range
  - Cuisine type (e.g., Italian, Chinese)
  - Ratings

- Availability
- Dietary preferences (e.g., vegetarian, vegan)
- **Sort Functionality:** Allows sorting dishes or restaurants by price, rating, or newest listings for easy browsing.

## 2. Menu Listings

- **Add/Edit Menu:** Restaurant owners can add new menu items with details like name, description, price, category, and availability.
- **Image Upload:** Owners can upload images of their dishes, which are stored and displayed in an image gallery to give customers a visual preview.
- **Categories Selection:** Owners can specify categories (e.g., appetizers, main course, desserts), making it easier for customers to search for dishes with specific features.

## DEPLOYMENT FOR BACKEND & FRONTEND :

### GIT EXPLANATION :

Git is a **version control system (VCS)** used to track changes in files and manage source code history. It's widely used in software development to help teams and individuals collaborate on projects, track code changes, and manage multiple versions of codebases.

Here are the core concepts and functionalities of Git:

### 1. Tracking and Versioning Code

- Git monitors changes in files over time. This enables developers to keep a history of every change, making it easy to view, compare, or revert to previous versions.
- Every time a developer saves a change (known as a "commit"), Git records it with a unique identifier. This allows tracking the development process step-by-step.

### 2. Distributed System

- Unlike traditional VCS that rely on a central server, Git is **distributed**. Every developer has a full copy of the project's history on their local machine, allowing them to work offline and independently.

- This design also enhances collaboration and backup, as every user's local repository is a complete version of the project.

### 3. Branches and Merging

- Git allows developers to create **branches**, which are separate lines of development. For instance, one branch could be for a new feature, another for fixing bugs, and another for experimenting.
- Branching enables parallel work without interfering with the main codebase. Once changes are complete, branches can be **merged** back into the main branch, integrating all updates.

### 4. Collaboration and Pull Requests

- Git is ideal for collaborative development. Developers can contribute code, suggest changes, or review each other's work through pull requests (on platforms like GitHub or GitLab).
- This makes it easy to organize contributions, discuss potential improvements, and ensure code quality before merging.

### 5. Commit History and Blame Tracking

- Git records the details of each commit, including who made the change, when, and what was changed. This **commit history** allows developers to track the progress of a project, identify the source of bugs, or understand the evolution of the codebase.
- With **blame tracking**, it's easy to trace specific lines of code to the person who last modified them, aiding in debugging and accountability.

### 6. Undoing Changes

- Git provides options to undo changes, whether it's discarding recent modifications, restoring files to a previous state, or rolling back to an older commit. This flexibility is crucial for risk-free experimentation and development.

### 7. Staging Area

- Git has a **staging area** where changes are listed before committing them. This area acts as a buffer, allowing developers to select specific changes to commit, group related updates, and ensure only intended changes are included.



## Why Git is Important

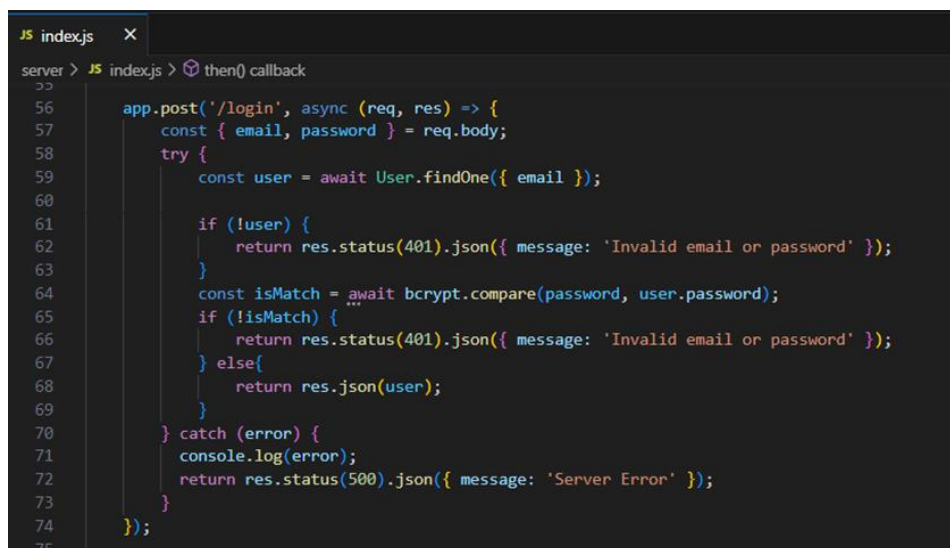
- **Efficiency:** Git handles changes quickly and effectively, even in large codebases.
- **Flexibility:** With branching and merging, Git enables flexible workflows that can adapt to various project needs.
- **Collaboration:** Git is built for teamwork, allowing multiple contributors to work simultaneously without overwriting each other's code.

Git has become an essential tool in modern development due to its robust capabilities, flexibility, and efficiency in managing complex projects with multiple contributors.

## User Authentication:

- **Backend**

Now, here we define the functions to handle http requests from the client for authentication.



```
JS index.js X
server > JS index.js > then() callback
56 app.post('/login', async (req, res) => {
57   const { email, password } = req.body;
58   try {
59     const user = await User.findOne({ email });
60
61     if (!user) {
62       return res.status(401).json({ message: 'Invalid email or password' });
63     }
64     const isMatch = await bcrypt.compare(password, user.password);
65     if (!isMatch) {
66       return res.status(401).json({ message: 'Invalid email or password' });
67     } else {
68       return res.json(user);
69     }
70   } catch (error) {
71     console.log(error);
72     return res.status(500).json({ message: 'Server Error' });
73   }
74 });
75
```

```

JS index.js X
server > JS index.js > then() callback > app.post('/login') callback
23 app.post('/register', async (req, res) => {
24   const { username, email, usertype, password, restaurantAddress, restaurantImage } = req.body;
25   try {
26     const existingUser = await User.findOne({ email });
27     if (existingUser) {
28       return res.status(400).json({ message: 'User already exists' });
29     }
30     const hashedPassword = await bcrypt.hash(password, 10);
31     if(usertype === 'restaurant'){
32       const newUser = new User({
33         username, email, usertype, password: hashedPassword, approval: 'pending'
34       });
35       const user = await newUser.save();
36       console.log(user._id);
37       const restaurant = new Restaurant({ownerId: user._id, title: username,
38         address: restaurantAddress, mainImg: restaurantImage, menu: []});
39       await restaurant.save();
40       return res.status(201).json(user);
41     } else{
42       const newUser = new User({
43         username, email, usertype, password: hashedPassword, approval: 'approved'
44       });
45       const userCreated = await newUser.save();
46       return res.status(201).json(userCreated);
47     }
48   } catch (error) {
49     console.log(error);
50     return res.status(500).json({ message: 'Server Error' });
51   }
52 });
53

```

## Frontend

Login:

```

JS GeneralContext.js U X
client > src > context > JS GeneralContext.js > GeneralContextProvider > register > then() callback
46 const login = async () =>{
47   try{
48     const loginInputs = {email, password}
49     await axios.post('http://localhost:6001/login', loginInputs)
50     .then( async (res)=>{
51
52       localStorage.setItem('userId', res.data._id);
53       localStorage.setItem('userType', res.data.usertype);
54       localStorage.setItem('username', res.data.username);
55       localStorage.setItem('email', res.data.email);
56       if(res.data.usertype === 'customer'){
57         navigate('/');
58       } else if(res.data.usertype === 'admin'){
59         navigate('/admin');
60       }
61     }).catch((err) =>{
62       alert("login failed!!");
63       console.log(err);
64     });
65   }catch(err){
66     console.log(err);
67   }
68 }
69

```

Logout:

```
GeneralContext.jsx U X
client > src > context > GeneralContext.jsx > [0] GeneralContextProvider > [0] login >
72
73   const logout = async () =>{
74
75     localStorage.clear();
76     for (let key in localStorage) {
77       if (localStorage.hasOwnProperty(key)) {
78         localStorage.removeItem(key);
79       }
80     }
81
82     navigate('/');
83   }
84
85
```

Register:

```
JS GeneralContext.js U X
client > src > context > JS GeneralContext.js > [0] GeneralContextProvider > [0] logout
75
76   const inputs = {username, email, usertype, password, restaurantAddress, restaurantImage};
77
78   const register = async () =>{
79     try{
80       await axios.post('http://localhost:6001/register', inputs)
81       .then( async (res)=>{
82         localStorage.setItem('userId', res.data._id);
83         localStorage.setItem('userType', res.data.usertype);
84         localStorage.setItem('username', res.data.username);
85         localStorage.setItem('email', res.data.email);
86
87         if(res.data.usertype === 'customer'){
88           navigate('/');
89         } else if(res.data.usertype === 'admin'){
90           navigate('/admin');
91         } else if(res.data.usertype === 'restaurant'){
92           navigate('/restaurant');
93         }
94       }).catch((err) =>{
95         alert("registration failed!!");
96         console.log(err);
97       });
98     }catch(err){
99       console.log(err);
100     }
101   }
102
```

## All Products (User):

- Frontend :

In the home page, we'll fetch all the products available in the platform along with the filters.

Fetching food items:

```
IndividualRestaurant.jsx 4, U X
client > src > pages > customer > IndividualRestaurant.jsx > IndividualRestaurant > handleCategoryCheckBox
33 const fetchRestaurants = async () =>{
34   await axios.get(`http://localhost:6001/fetch-restaurant/${id}`).then(
35     (response)=>{
36       setRestaurant(response.data);
37       console.log(response.data)
38     }
39   ).catch((err)=>{
40     console.log(err);
41   })
42 }
43
44 const fetchCategories = async () =>{
45   await axios.get('http://localhost:6001/fetch-categories').then(
46     (response)=>{
47       setAvailableCategories(response.data);
48     }
49   )
50 }
51
52 const fetchItems = async () =>{
53   await axios.get('http://localhost:6001/fetch-items').then(
54     (response)=>{
55       setItems(response.data);
56       setVisibleItems(response.data);
57     }
58   )
59 }
60
```

Filtering products:

```
Products.jsx 2, U X
client > src > components > Products.jsx > Products > useEffect() callback
38 const [sortFilter, setSortFilter] = useState('popularity');
39 const [categoryFilter, setCategoryFilter] = useState({});
40 const [genderFilter, setGenderFilter] = useState({});
41
42 const handleCategoryCheckBox = (e) =>{
43   const value = e.target.value;
44   if(e.target.checked){
45     setCategoryFilter({...categoryFilter, value});
46   }else{
47     setCategoryFilter(categoryFilter.filter(size=> size !== value));
48   }
49 }
50
51 const handleGenderCheckBox = (e) =>{
52   const value = e.target.value;
53   if(e.target.checked){
54     setGenderFilter({...genderFilter, value});
55   }else{
56     setGenderFilter(genderFilter.filter(size=> size !== value));
57   }
58 }
59
60 const handleSortFilterChange = (e) =>{
61   const value = e.target.value;
62   setSortFilter(value);
63   if(value === 'low-price'){
64     setVisibleProducts(visibleProducts.sort((a,b)=> a.price - b.price))
65   } else if (value === 'high-price'){
66     setVisibleProducts(visibleProducts.sort((a,b)=> b.price - a.price))
67   }else if (value === 'discount'){
68     setVisibleProducts(visibleProducts.sort((a,b)=> b.discount - a.discount))
69   }
70 }
71
72 useEffect(()=>{
73   if (categoryFilter.length > 0 && genderFilter.length > 0){
74     setVisibleProducts(products.filter(product=> categoryFilter.includes(product.category) && genderFilter.includes(product.gender) ));
75   }else if (categoryFilter.length === 0 && genderFilter.length > 0){
76     setVisibleProducts(products.filter(product=> genderFilter.includes(product.gender) ));
77   } else if (categoryFilter.length > 0 && genderFilter.length === 0){
78     setVisibleProducts(products.filter(product=> categoryFilter.includes(product.category)));
79   }else{
80     setVisibleProducts(products);
81   }
82 }, [categoryFilter, genderFilter])
83
84
85
86
```

- **Backend**

In the backend, we fetch all the products and then filter them on the client side.

```
JS index.js X
server > JS index.js > then() callback > app.get('/fetch-banner') callback
100
101 // fetch products
102
103 app.get('/fetch-products', async(req, res)=>{
104   try{
105     const products = await Product.find();
106     res.json(products);
107   }catch(err){
108     res.status(500).json({ message: 'Error occurred' });
109   }
110 }
111 })
```

**Add product to cart:**

- **Frontend**

Here, we can add the product to the cart and later can buy them.

```
IndividualRestaurant.jsx U X
client > src > pages > customer > IndividualRestaurant.jsx > IndividualRestaurant
114 const handleAddToCart = async(foodItemId, foodItemName, restaurantId,
115                                foodItemImg, price, discount) =>{
116   await axios.post('http://localhost:6001/add-to-cart', {userId, foodItemId,
117                                                         foodItemName, restaurantId, foodItemImg,
118                                                         price, discount, quantity}).then(
119     (response)=>{
120       alert("product added to cart!!");
121       setCartItem('');
122       setQuantity(0);
123       fetchCartCount();
124     }
125   ).catch((err)=>{
126     alert("Operation failed!!");
127   })
128 }
129
```

- **Backend**

Add product to cart:



```
JS index.js X
server > JS index.js > then() callback > app.put('/remove-item') callback

402 // add cart item
403
404 app.post('/add-to-cart', async(req, res)=>{
405     const {userId, foodItemId, foodItemName, restaurantId,
406         foodItemImg, price, discount, quantity} = req.body
407     try{
408         const restaurant = await Restaurant.findById(restaurantId);
409         const item = new Cart({userId, foodItemId, foodItemName,
410             restaurantId, restaurantName: restaurant.title,
411             foodItemImg, price, discount, quantity});
412         await item.save();
413         res.json({message: 'Added to cart'});
414     }catch(err){
415         res.status(500).json({message: "Error occurred"});
416     }
417 })
418
```

## Order products:

Now, from the cart, let's place the order

- Frontend

```
Cart.jsx 2, U X
client > src > pages > customer > Cart.jsx > [0] Cart

72 const placeOrder = async() =>{
73     if(cart.length > 0){
74         await axios.post('http://localhost:6001/place-cart-order', {userId, name,
75             mobile, email, address, pincode, paymentMethod,
76             orderDate: new Date()}).then(
77             (response)=>{
78                 alert('Order placed!!');
79                 setName('');
80                 setMobile('');
81                 setEmail('');
82                 setAddress('');
83                 setPincode('');
84                 setPaymentMethod('');
85                 navigate('/profile');
86             })
87     }
88 }
89
```

- Backend

In the backend, on receiving the request from the client, we then place the order for the products in the cart with the specific user Id.

```

JS index.js X
server > JS index.js > then() callback > app.listen() callback

435 // Order from cart
436
437 app.post('/place-cart-order', async(req, res)=>{
438     const {userId, name, mobile, email, address, pincode,
439           paymentMethod, orderDate} = req.body;
440     try{
441         const cartItems = await Cart.find({userId});
442         cartItems.map(async (item)=>{
443             const newOrder = new Orders({userId, name, email,
444                   mobile, address, pincode, paymentMethod,
445                   orderDate, restaurantId: item.restaurantId,
446                   restaurantName: item.restaurantName,
447                   foodItemId: item.foodItemId, foodItemName: item.foodItemName,
448                   foodItemImg: item.foodItemImg, quantity: item.quantity,
449                   price: item.price, discount: item.discount});
450             await newOrder.save();
451             await Cart.deleteOne({_id: item._id})
452         })
453         res.json({message: 'Order placed'});
454     }catch(err){
455         res.status(500).json({message: "Error occurred"});
456     }
457 })
458

```

## Add new product:

Here, in the admin dashboard, we will add a new product.

- Frontend:

```

NewProduct.jsx 1, U X
client > src > pages > restaurant > NewProduct.jsx > NewProduct

46 const handleNewProduct = async() =>{
47     await axios.post('http://localhost:6001/add-new-product', {restaurantId: restaurant._id,
48           productName, productDescription, productMainImg, productCategory, productMenuCategory,
49           productNewCategory, productPrice, productDiscount}).then(
50         (response)=>{
51             alert("product added");
52             setProductName('');
53             setProductDescription('');
54             setProductMainImg('');
55             setProductCategory('');
56             setProductMenuCategory('');
57             setProductNewCategory('');
58             setProductPrice(0);
59             setProductDiscount(0);
60             navigate('/restaurant-menu');
61         })
62     }
63 }
64

```

- Backend:

```
JS index.js X
server > JS index.js > then() callback
285 // Add new product
286 app.post('/add-new-product', async(req, res)=>{
287   const {restaurantId, productName, productDescription,
288         productMainImg, productCategory, productMenuCategory,
289         productNewCategory, productPrice, productDiscount} = req.body;
290   try{
291     if(productMenuCategory === 'new category'){
292       const admin = await Admin.findOne();
293       admin.categories.push(productNewCategory);
294       await admin.save();
295       const newProduct = new FoodItem({restaurantId, title: productName,
296                                       description: productDescription, itemImg: productMainImg,
297                                       category: productCategory, menuCategory: productNewCategory,
298                                       price: productPrice, discount: productDiscount, rating: 0});
299       await newProduct.save();
300       const restaurant = await Restaurant.findById(restaurantId);
301       restaurant.menu.push(productNewCategory);
302       await restaurant.save();
303     } else{
304       const newProduct = new FoodItem({restaurantId, title: productName,
305                                       description: productDescription, itemImg: productMainImg,
306                                       category: productCategory, menuCategory: productMenuCategory,
307                                       price: productPrice, discount: productDiscount, rating: 0});
308       await newProduct.save();
309     }
310     res.json({message: "product added!!"});
311   }catch(err){
312     res.status(500).json({message: "Error occured"});
313   }
314 })
315
```

Along with this, implement additional features to view all orders, products, etc., in the admin dashboard.

## TESTING AND QUALITY ASSURANCE :

- Testing and Quality Assurance (QA) are essential to ensure that a MERN-based house rental app runs smoothly, is secure, and offers an optimal user experience. This process covers both functional and non-functional aspects of the application, such as usability, security, and performance.

### 1. Requirements Analysis and Test Planning

- **Define Requirements:** Identify functional requirements (like property search, user authentication, and payment processing) and non-functional requirements (like response time and security).
- **Test Plan Creation:** Develop a comprehensive test plan that outlines test types, objectives, environments, tools, and timelines.



## 2. Functional Testing

- **Unit Testing:** Test individual components of the app in isolation to ensure they perform correctly. For example, testing the search bar, property listing display, and filters separately.
- **Integration Testing:** Ensure that the components work together as expected. For example, testing that the search function interacts correctly with the database to retrieve filtered listings.
- **API Testing:** Validate that the backend API endpoints (like login, property listings, and user interactions) respond correctly, handle errors, and provide accurate data to the frontend.
- **User Acceptance Testing (UAT):** Verify with end users (both renters and property owners) to confirm that the app meets their needs and expectations.

## 3. User Interface (UI) and Usability Testing

- **UI Consistency:** Ensure design elements like fonts, colors, and layouts are consistent across different pages and devices.
- **Navigation and Flow Testing:** Check the ease of navigating the app, whether users can easily locate features like searching properties, applying for listings, or messaging owners.
- **Responsive Design Testing:** Test the app on various devices (e.g., mobile, tablet, and desktop) to confirm that it adapts to different screen sizes and resolutions.

## 4. Performance Testing

- **Load Testing:** Simulate high user traffic to evaluate how the app handles multiple users searching, messaging, and making payments simultaneously.
- **Stress Testing:** Push the app beyond its typical operational capacity to observe behavior under extreme conditions and identify breakpoints.
- **Response Time Monitoring:** Ensure the app loads listings quickly and responds to user actions (like applying filters) in an acceptable timeframe.

## 5. Security Testing

- **Authentication and Authorization:** Test user authentication processes to verify that only authorized users can access specific features
- **Data Protection:** Ensure sensitive data, like personal details and payment information, is secure, using encryption and safe handling of user data.

- **Vulnerability Testing:** Identify and mitigate potential security risks, such as SQL injections, cross-site scripting (XSS), and cross-site request forgery (CSRF).

## 6. Database Testing

- **Data Integrity:** Ensure that data entered by users, like property listings and user profiles, is correctly saved and retrieved from MongoDB.
- **Data Consistency:** Verify that the data remains consistent across different modules and that updates (like price changes or new listings) appear instantly.

## 7. Regression Testing

- **Re-test Existing Features:** Run tests on previously tested functionality whenever new features are added or existing features are updated to ensure no new bugs were introduced.
- **Automated Regression Testing:** Set up automated test suites to frequently verify core functionalities, reducing manual effort and catching issues early.

## 8. Non-Functional Testing

- **Performance Metrics:** Measure key metrics like load times, response times, and memory usage.
- **Usability Testing:** Conduct usability testing with a sample of end-users, gathering feedback on the app's layout, design, and ease of use.
- **Accessibility Testing:** Verify that the app is accessible to users with disabilities, ensuring compatibility with screen readers and keyboard navigation.

## 9. Continuous Integration (CI) and Continuous Deployment (CD)

- **Automated Testing in CI/CD Pipelines:** Integrate automated tests within the CI/CD pipeline so that code is tested each time there is a new commit. This ensures rapid detection of issues.
- **Continuous Monitoring:** Set up monitoring tools to track app health, error rates, and user behavior after deployment.

### 1. Functional Test Cases

Test Case ID	Description	Expected Outcome	Status
TC_FT_01	Verify login with valid credentials	User is redirected to the dashboard	Pass
TC_FT_02	Verify search functionality	Products matching the search term are displayed	Pass
TC_FT_03	Add property to existing test	Property added detailed show up	Pass
TC_FT_04	Apply valid number of price	Price is applied, and final price is adjusted	Pass
TC_FT_05	Verify admin can add new property for renter	New property appears on the site	Pass

### 2. Usability Test Cases

Test Case ID	Description	Expected Outcome	Status
TC_UT_01	Verify intuitive navigation for first-time users	Users easily locate major sections (like newly updated locations and price)	Pass
TC_UT_02	Test product filtering by category and price	Property filter correctly, and interface is clear	Pass

### 3. Performance Test Cases

Test Case ID	Description	Expected Outcome	Status
TC_PT_01	Simulate 100 concurrent users on the platform	Platform performs within acceptable speed limits	Pass

TC_PT_02	Test load time under peak traffic	Page load times are below 3 seconds	Pass
----------	-----------------------------------	-------------------------------------	------

#### 4. Security Test Cases

Test Case ID	Description	Expected Outcome	Status
TC_ST_01	Test for SQL injection vulnerabilities	System prevents SQL injection attempts	Pass
TC_ST_02	Verify password encryption	Passwords are securely hashed and stored	Pass
TC_ST_03	Check for data transmission over HTTPS	All data is transmitted securely	Pass

#### 5. Compatibility Test Cases

Test Case ID	Description	Expected Outcome	Status
TC_CT_01	Verify site compatibility with Chrome	Site functions smoothly without UI issues	Pass
TC_CT_02	Verify site compatibility with mobile Safari	Site is responsive and fully functional	Pass
TC_CT_03	Test screen responsiveness across devices	Layout adapts correctly to desktop, tablet, mobile	Pass

- **Overall Outcome:** Testing results were positive, confirming that the platform is robust, user-friendly, secure, and compatible across multiple devices and browsers.

## CHALLENGES AND SOLUTIONS :

- Developing a house rental app using the MERN stack comes with its own set of challenges. Below are some common challenges faced during development, along with potential solutions to address them:

### 1. Scalability

- **Challenge:** As the number of users and properties increases, the application must efficiently handle larger amounts of data and user requests without performance degradation.
- **Solution:**
  - Implement **load balancing** to distribute incoming traffic across multiple servers.
  - Utilize **MongoDB's sharding** capabilities to split data across different databases, improving read and write performance.
  - Optimize the application code and database queries to reduce response times.

### 2. Data Consistency

- **Challenge:** Ensuring data integrity and consistency when multiple users are making changes (e.g., updating property details or applying for rentals).
- **Solution:**
  - Use **transactions** in MongoDB for operations that require multiple document updates to ensure all changes either succeed or fail together.
  - Implement server-side validation checks to prevent data corruption and ensure all incoming data is properly formatted.

### 3. User Authentication and Security

- **Challenge:** Protecting user data and preventing unauthorized access while ensuring a smooth login experience.
- **Solution:**
  - Use **JWT (JSON Web Tokens)** for secure, stateless authentication. This allows for easy validation and user session management.

- Implement robust **password hashing** using libraries like bcrypt to securely store user passwords.
- Regularly conduct **security audits** and vulnerability assessments to identify and mitigate risks.

#### 4. Search and Filtering Performance

- **Challenge:** Efficiently searching and filtering through potentially large datasets of properties can lead to slow response times.
- **Solution:**
  - Implement **indexing** in MongoDB on fields commonly queried (e.g., location, price, and property type) to speed up searches.
  - Utilize **aggregation pipelines** for complex queries that involve filtering and sorting.
  - Cache frequently accessed data using tools like Redis to reduce load on the database.

#### 5. Cross-Platform Compatibility

- **Challenge:** Ensuring the application provides a consistent experience across various devices and screen sizes.
- **Solution:**
  - Use **responsive design principles** in the frontend (e.g., with CSS frameworks like Bootstrap or Material-UI) to ensure the app adapts to different screen sizes.
  - Conduct thorough testing on multiple devices and browsers to identify and resolve compatibility issues.

#### 6. User Experience (UX)

- **Challenge:** Creating an intuitive and engaging user experience for both renters and property owners.
- **Solution:**
  - Conduct **user research** to gather insights on user needs and pain points, then iterate on design based on feedback.
  - Implement features like **real-time chat** or notifications to enhance communication between renters and property owners.
  - Utilize UI/UX design tools (e.g., Figma or Adobe XD) to prototype and test user flows before full implementation.

## 7. Payment Processing

- **Challenge:** Implementing a secure and reliable payment system for processing rent payments and handling transactions.
- **Solution:**
  - Use established payment gateways like **Stripe or PayPal** that provide secure APIs for handling transactions and managing payment data.
  - Ensure compliance with regulations (like PCI-DSS) when handling financial transactions and user payment information.

## 8. Maintaining Code Quality

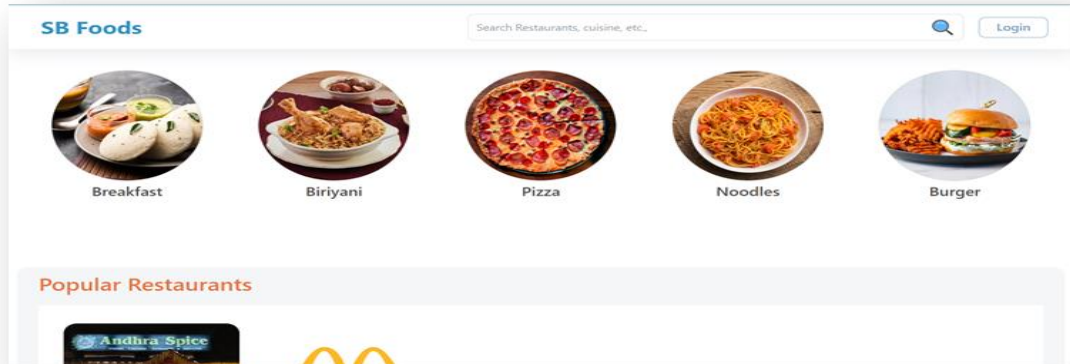
- **Challenge:** As the application grows, maintaining clean, modular, and well-documented code becomes challenging.
- **Solution:**
  - Adopt **coding standards and best practices** across the development team, using tools like ESLint for JavaScript code quality.
  - Implement a **code review process** to ensure code changes are thoroughly vetted before being merged into the main branch.
  - Use **automated testing** frameworks (like Jest or Mocha) to regularly test code changes and maintain functionality.

## 9. Deployment and DevOps

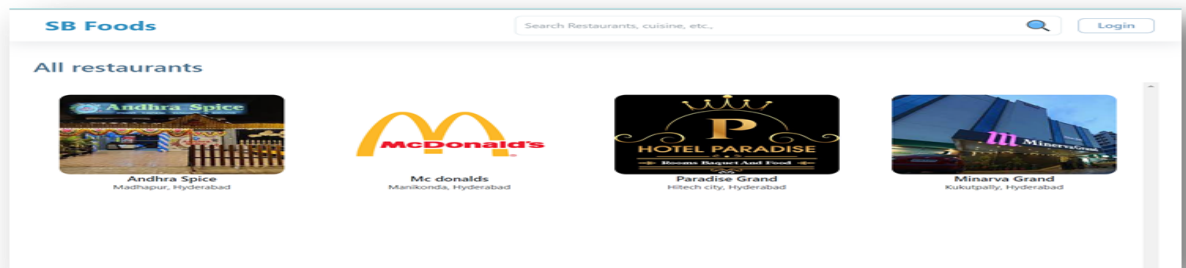
- **Challenge:** Managing deployment processes and maintaining uptime can be complex, especially with frequent updates.
- **Solution:**
  - Use **containerization** (e.g., Docker) to standardize the deployment environment and simplify deployment processes.
  - Implement **CI/CD pipelines** to automate testing and deployment, ensuring that changes are quickly integrated and delivered to users.
  - Monitor application performance and health using tools like New Relic or Grafana to quickly identify and resolve issues.

## PROJECT EXECUTION (Screenshots or Demo)

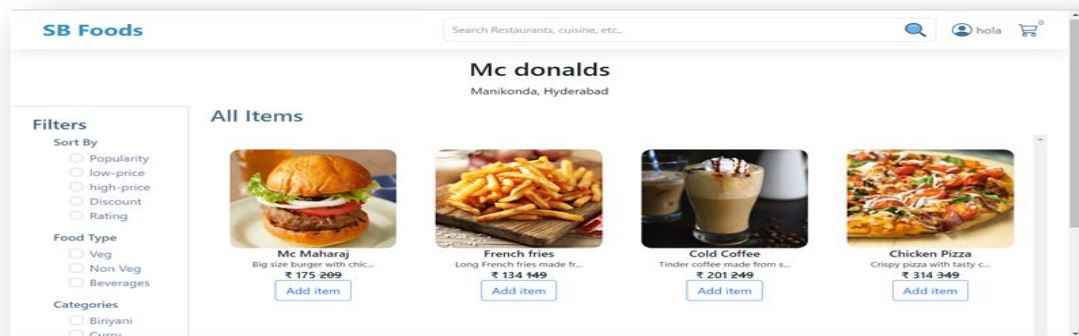
### Landing page



### Restaurants

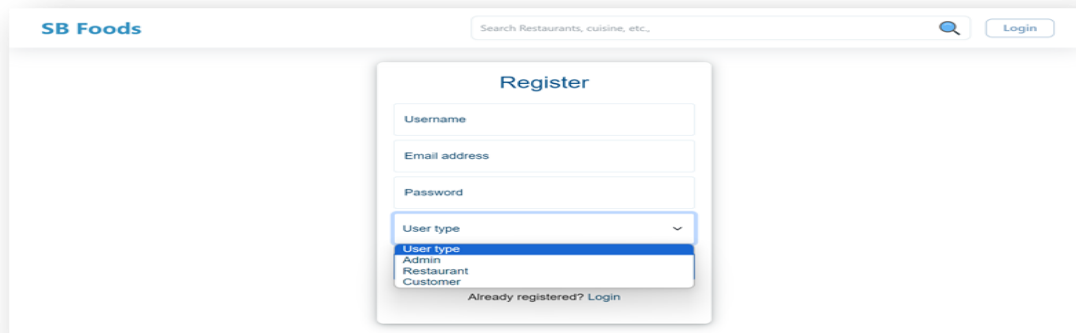


### Restaurant Menu





## Authentication



The image shows the 'Register' form on the SB Foods website. The form is centered on the page and includes input fields for Username, Email address, and Password. Below these is a dropdown menu for 'User type' with options: User type, Admin, Restaurant, and Customer. A link 'Already registered? Login' is located at the bottom of the form. The website header shows the SB Foods logo, a search bar, and a Login button.

SB Foods

Search Restaurants, cuisine, etc.,

Login

### Register

Username

Email address

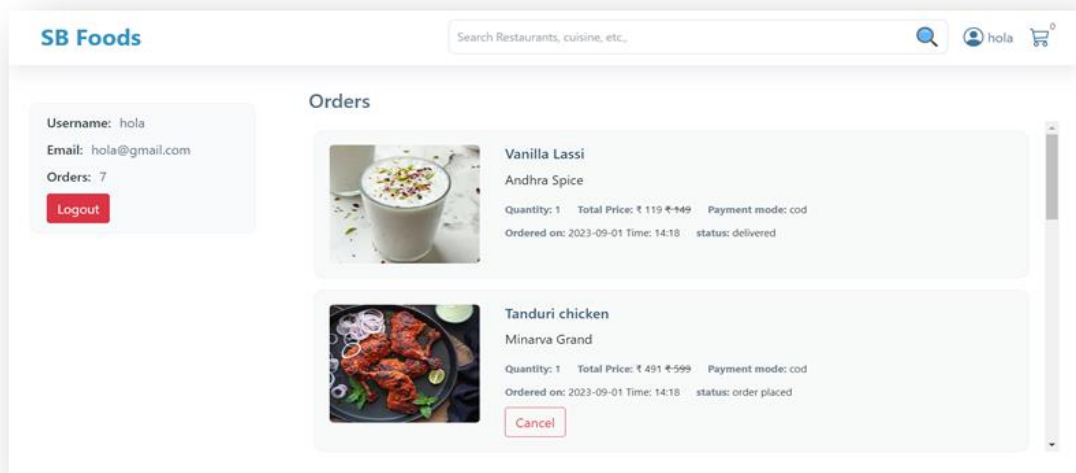
Password

User type

- User type
- Admin
- Restaurant
- Customer

Already registered? Login

## User Profile



The image shows the 'User Profile' and 'Orders' page on the SB Foods website. The user is logged in as 'hola'. The profile section on the left shows the username 'hola', email 'hola@gmail.com', and 'Orders: 7', with a 'Logout' button. The 'Orders' section on the right lists two orders: 'Vanilla Lassi' from 'Andhra Spice' and 'Tanduri chicken' from 'Minarva Grand'. Each order entry includes a quantity of 1, total price, payment mode, and order status. A 'Cancel' button is visible for the 'Tanduri chicken' order. The website header shows the SB Foods logo, a search bar, and user profile icons.

SB Foods

Search Restaurants, cuisine, etc.,

hola

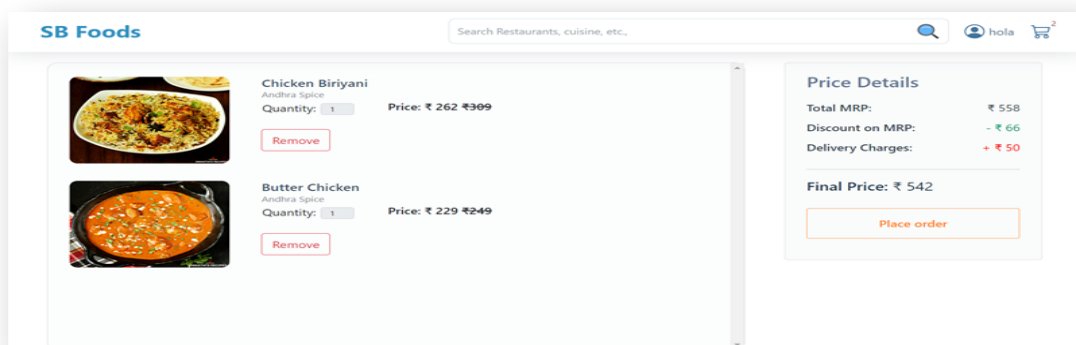
Username: hola  
Email: hola@gmail.com  
Orders: 7  
Logout

### Orders

**Vanilla Lassi**  
Andhra Spice  
Quantity: 1 Total Price: ₹ 119 ₹149 Payment mode: cod  
Ordered on: 2023-09-01 Time: 14:18 status: delivered

**Tanduri chicken**  
Minarva Grand  
Quantity: 1 Total Price: ₹ 491 ₹599 Payment mode: cod  
Ordered on: 2023-09-01 Time: 14:18 status: order placed  
Cancel

## Cart



The image shows the 'Cart' page on the SB Foods website. The cart contains two items: 'Chicken Biryani' from 'Andhra Spice' and 'Butter Chicken' from 'Andhra Spice'. Each item has a quantity of 1 and a 'Remove' button. The 'Price Details' section on the right shows the total MRP of ₹ 558, a discount of ₹ 66, and delivery charges of ₹ 50, resulting in a final price of ₹ 542. A 'Place order' button is at the bottom of the price details section. The website header shows the SB Foods logo, a search bar, and user profile icons.

SB Foods

Search Restaurants, cuisine, etc.,

hola

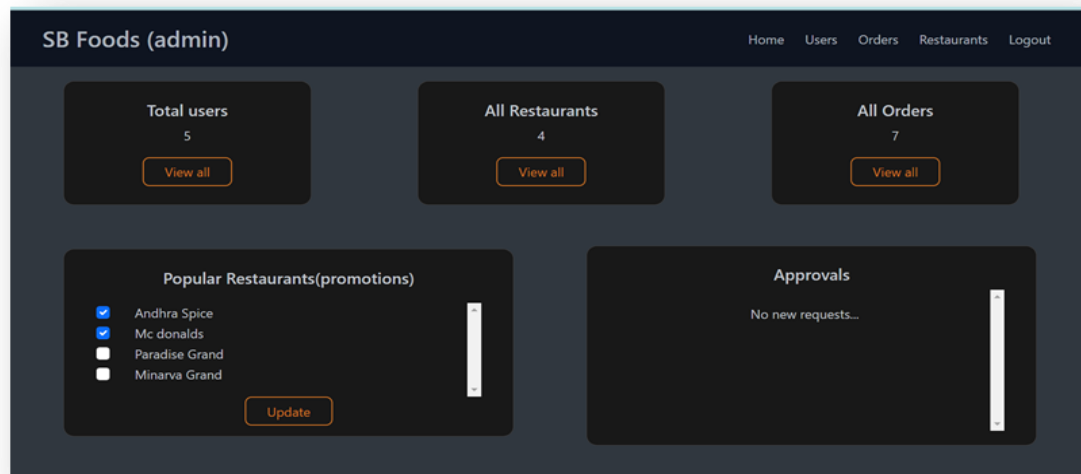
**Chicken Biryani**  
Andhra Spice  
Quantity: 1 Price: ₹ 262 ₹309  
Remove

**Butter Chicken**  
Andhra Spice  
Quantity: 1 Price: ₹ 229 ₹249  
Remove

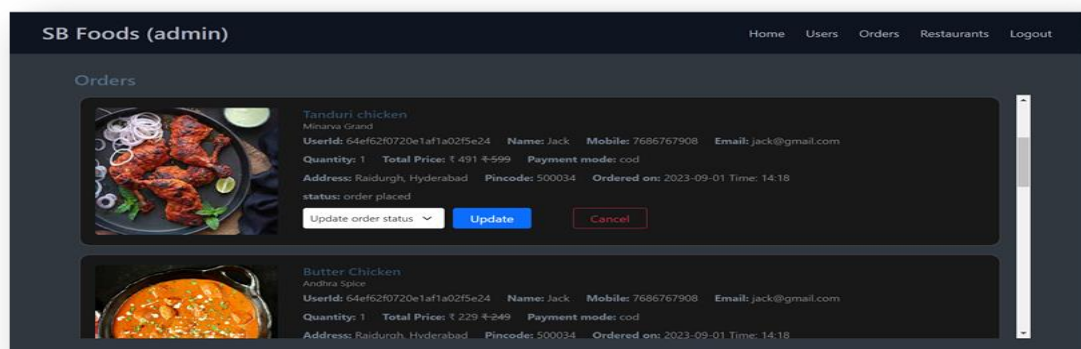
### Price Details

Total MRP: ₹ 558  
Discount on MRP: - ₹ 66  
Delivery Charges: + ₹ 50  
Final Price: ₹ 542  
Place order

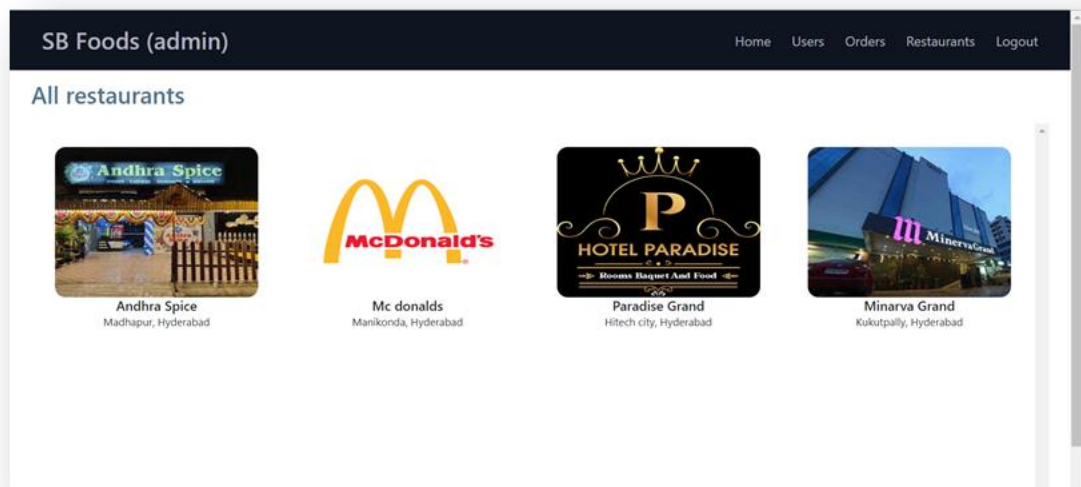
- **Admin dashboard**



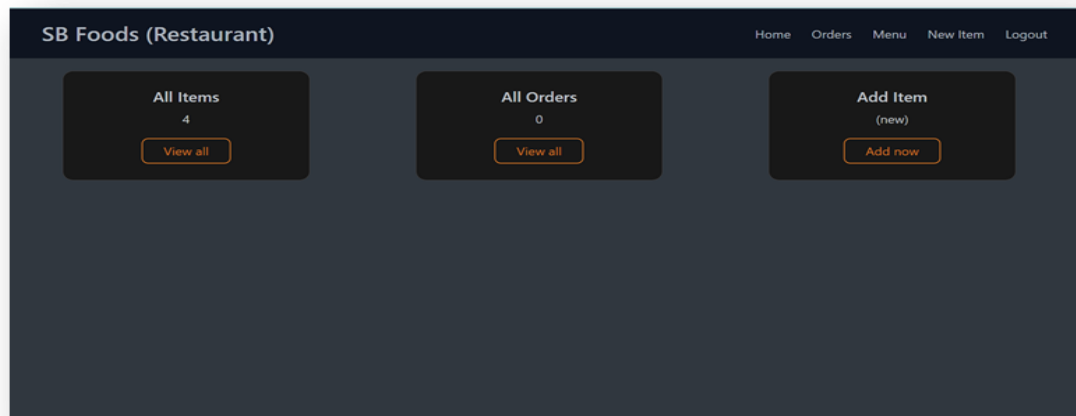
- **All Orders**



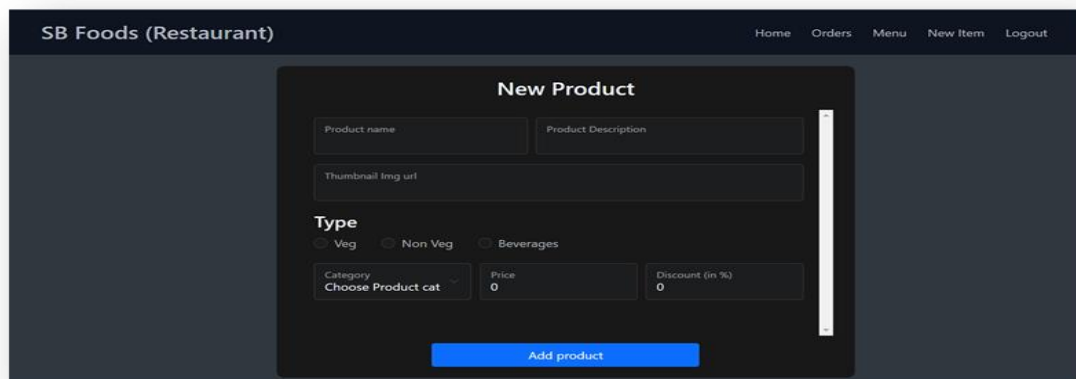
- **All restaurants**



- **Restaurant Dashboard**



- **New Item**



## KNOWN ISSUES:

- **Order Duplication:** In some cases, users might experience duplicate orders being placed when they click the "Place Order" button multiple times due to slow network response.
- **Delayed Notifications:** There may be a delay in receiving order confirmation notifications via email or SMS due to server load.
- **Profile Update Glitches:** Users might encounter issues when updating their profile information, where changes are not reflected immediately or require multiple attempts.
- **Search Inconsistencies:** Occasionally, the search functionality might return inaccurate or incomplete results, especially during high traffic times.

- **Payment Processing Errors:** Some users have reported issues with payment processing, including failed transactions and incorrect payment status updates.
- **Responsive Design Issues:** On certain devices or screen sizes, the app's layout may not adjust properly, leading to a suboptimal user experience.
- **Session Timeout Problems:** Users may sometimes be logged out unexpectedly, leading to session timeout issues while browsing menus or placing orders.
- **Image Upload Limitations:** Restaurant owners might face difficulties uploading multiple images at once, especially if the images are large in size.

## FUTURE ENHANCEMENT :

- Improvement with design and layout to modern and accurate web page .
- Providing a better user experience with Artificial Intelligence and Machine Learning .
- More enhancement in number of user active accounts .

## CONCLUSION :

- The development of a house rental app utilizing the MERN stack—MongoDB, Express.js, React, and Node.js—offers a comprehensive and robust solution for connecting renters with property owners in an efficient and user-friendly manner. The MERN stack is well-suited for this application due to its flexibility, scalability, and ability to handle real-time data interactions seamlessly.
- By leveraging MongoDB as a NoSQL database, the app can effectively manage large volumes of property listings and user information while ensuring data integrity and performance. Express.js provides a lightweight and modular framework for building a scalable backend API that can handle complex requests efficiently. Node.js enhances the application's performance with its non-blocking architecture, allowing for rapid handling of multiple user requests.
- On the frontend, React empowers developers to create an interactive and dynamic user interface, ensuring a smooth user experience. The component-based architecture of React facilitates efficient updates and state management,

which is crucial for features such as real-time property listings and user notifications.

- Moreover, implementing robust security measures, intuitive navigation, and effective search and filter functionalities ensures that both renters and property owners have a seamless experience. Continuous integration and deployment practices, combined with thorough testing and quality assurance processes, further enhance the application's reliability and performance.
- In summary, the house rental app built with the MERN stack not only addresses the current needs of the rental market but also positions itself for future growth and adaptation. By prioritizing user experience, security, and maintainability, this application stands to provide significant value to its users while fostering a vibrant community of renters and property owners.

## REFERENCE :

### 1. Technical Documentation

- Frameworks and Libraries:
  - Official documentation for React (for styling).
  - Examples:
    - ◆ React installation guide: <https://reactjs.org/docs/create-a-new-react-app.html>
    - ◆ Ant Design React UI library :<https://ant.design/docs/react/introduce>
    - ◆ Node.js Download: <https://nodejs.org/en/download/>  
→ Installation instructions: <https://nodejs.org/en/download/package-manager/>
- Database Management:
  - References for databases used (MongoDB).
  - Examples:
    - ◆ MongoDBDownload:<https://www.mongodb.com/try/download/community>
    - ◆ Installation instructions: <https://docs.mongodb.com/manual/installation/>

### 2. Books, Articles, and Tutorials

- Books: Include books that helped guide the architectural design or backend/front-end development practices. For example:
  - "JavaScript: The Good Parts" by Douglas Crockford for JavaScript best practices.
  - "Designing Data-Intensive Applications" by Martin Kleppmann for database and scalability strategies.
- Online Articles:
  - Articles or blog posts on implementing performance optimization, scaling e-commerce platforms, or best practices in UX design for web applications.

### 3. Industry Standards and Best Practices

- User Experience:
  - Guidelines on UX best practices, such as Nielsen Norman Group or Material Design by Google, can be cited if they influenced your design.
  - Example:
    - Nielsen Norman Group: <https://www.nngroup.com/articles/>
- Performance Optimization and CDN Use:
  - Guides and industry benchmarks from Google Lighthouse or Web.dev for front-end performance and mobile responsiveness.
  - Example:
    - Web.dev (Google): <https://web.dev/>

THANK YOU...