OrderOnTheGo: Your On-Demand Food Ordering Solution

Team Members:

1. **Kontham Yaswanthi** - konthamyaswanthi@gmail.com
2. **Togaru Devi Naga Durga Lakshmi** - **devithogaru2@gmail.com**
3. **V Ashmitha** - ashmithavankayala@gmail.com
4. **Vegesna Varshitha** - varshithavarmavegesna@gmail.com

INTRODUCTION:

OrderOnTheGo: Your On-Demand Food Ordering Solution is a full-stack web application developed to streamline and modernize the food ordering process. The platform bridges the gap between customers and restaurants by offering an intuitive interface for users to browse, select, and order their favourite meals with ease. Whether you're ordering lunch from your desk or managing orders from a restaurant dashboard, this application provides a seamless experience for both users and administrators.

Built using the MERN stack (MongoDB, Express.js, React.js, and Node.js), the application supports robust performance, secure data management, and real-time interaction. The user-facing side of the platform features a clean and responsive UI where users can register, log in, browse menus, add items to their cart, and place orders. Each order is tracked and updated to ensure transparency and timely delivery.

On the administrative side, the system offers a powerful dashboard where restaurant owners or admins can manage food listings, view order details, update inventory, and monitor user activity. Role-based access ensures that users only see what's relevant to them.

The project was designed with scalability and usability in mind, making it suitable for both small eateries and larger food delivery businesses. In addition to its core functionality, the application emphasizes responsive design, making it accessible across devices, from mobile phones to desktops.

OrderOnTheGo demonstrates effective integration of frontend and backend technologies, user authentication, database management, and practical UI/UX design principles. It represents a real-world solution in the domain of online food delivery and serves as a hands-on learning experience in full-stack development.

This documentation outlines the system's features, technologies, setup instructions, and key design considerations, providing a comprehensive overview of how the application was developed and how it functions.

PROJECT OVERVIW

PURPOSE:

The primary purpose of OrderOnTheGo is to provide a convenient and efficient platform for users to order food online and for restaurant administrators to manage their offerings and orders digitally. In today's fast-paced world, customers expect quick and seamless access to food services. This project aims to meet that demand by digitizing the food ordering process with a user-friendly interface and robust backend support.

From a user perspective, the application simplifies the process of discovering, selecting, and ordering food from a variety of options. Users can create accounts, explore menus, manage their cart, and track orders—all from a single platform. The design focuses on reducing the time and friction involved in placing an order, thereby improving customer satisfaction.

From an admin perspective, the platform serves as a centralized system to manage restaurant data, food items, and incoming orders. Admins can add, update, or delete food listings, monitor order statuses, and maintain overall control of the system. This not only streamlines daily operations but also minimizes manual errors and enhances business efficiency.

Additionally, the project serves an educational purpose. It provides hands-on experience in full-stack web development using modern technologies such as React, Node.js, Express, and MongoDB. It showcases real-world implementation of concepts like RESTful APIs, user authentication, role-based access control, responsive design, and CRUD operations.

In summary, the purpose of this project is twofold:

1. To build a real-time, full-stack food ordering solution that meets the practical needs of users and administrators.

2. To serve as a comprehensive learning project that demonstrates the integration of frontend and backend technologies in a real-world context.

FEATURES:

OrderOnTheGo offers a comprehensive set of features for both users and administrators, making the food ordering experience smooth, efficient, and intuitive.

On the **user side**, the application provides a secure authentication system that allows customers to register and log in using their credentials. Once logged in, users can browse a dynamic list of food items, each displaying details such as name, description, image, and price. The platform includes convenient search and filter functionalities, enabling users to easily find their preferred dishes by category or keyword. After selecting their desired items, users can add them to their cart and proceed to a streamlined checkout process. Once the order is placed, users can track the real-time status of their order, from confirmation to delivery, all within the same interface.

For the **admin side**, the platform includes a separate login system with role-based access control to ensure security and separation of privileges. Administrators gain access to a dedicated dashboard where they can manage the menu using full CRUD operations— creating, updating, or deleting food items as needed. The system also allows admins to view and manage all incoming customer orders, including the ability to update order statuses (such as "Pending," "Preparing," or "Delivered"). This centralized order and menu management enhances operational efficiency and allows admins to maintain full control of the platform.

In terms of **core functionality**, OrderOnTheGo is built using the MERN stack—MongoDB, Express.js, React.js, and Node.js—ensuring scalability, performance, and a modern development architecture. It features RESTful APIs for seamless communication between the frontend and backend, and a fully responsive UI that ensures compatibility across devices, including smartphones, tablets, and desktops. The implementation of role-based access control further strengthens the platform's security and usability for both users and admins.

ARCHITECTURE:
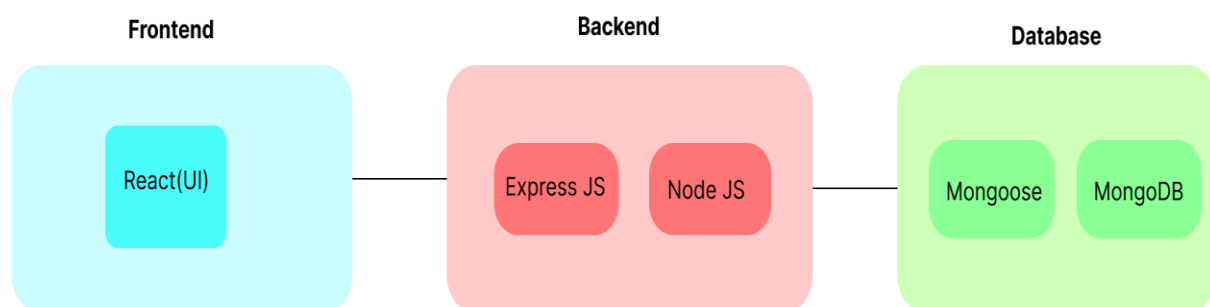
In this architecture diagram:

• The frontend is represented by the "Frontend" section, including user interface components such as User Authentication, Cart, Products, Profile, Admin dashboard, etc.,

• The backend is represented by the "Backend" section, consisting of API endpoints for

Users, Orders, Products, etc., It also includes Admin Authentication and an Admin Dashboard.

• The Database section represents the database that stores collections for Users, Admin, Cart, Orders, and products.

ER DIAGRAM:

The database schema of OrderOnTheGo is designed to efficiently handle all operations involved in a food ordering platform. The ER diagram outlines the key entities—User, Order, Food, Cart, Restaurant, and Admin—and their relationships.

1. User

The user entity stores information about platform users. It contains fields such as:

- username, password, email, and user type (e.g., customer or admin)

Users are related to both orders and cart, meaning each user can place multiple orders and maintain a cart.

2. Order

The order entity manages all customer orders. Key fields include:

- Customer details: user Id, name, email, mobile, address, pin code

- Order details: order Date, delivery Date, payment Method, discount, price

- Food details: food Item Id, food Item Name, food Item Image

- Restaurant info: restaurant Id, restaurant Name

Each order is associated with a user and can contain one or more food items.

3. Cart

The cart entity holds the current items selected by a user before placing an order. It includes:

- User Id, restaurant Id, restaurant Name

- Food details: food Item Id, food Item Name, food Item Image

quantity, discount, price

4. Food

The food entity stores menu items. It contains:

- Item info: title, description, item Image

- Meta info: category, menu Category, rating, discount, price, restaurant Id

This allows multiple restaurants to manage their menus independently.

5. Restaurant

This entity captures restaurant-specific data:

- Owner Id, menu, address, and mailing info

Each restaurant has its own menu and is associated with multiple food items.

6. Admin

The admin entity includes minimal fields like:

- banner, categories

It is responsible for managing global content like promotional banners or food categories.

This ER model ensures normalized data handling, supports scalability, and aligns with the application's logic of managing users, restaurants, orders, and carts effectively. It enables smooth interactions between frontend components and backend APIs.

PRE REQUISITES:

To develop a full-stack food ordering app using React JS, Node.js, and MongoDB, there are several prerequisites you should consider. Here are the key prerequisites for developing such an application:

Node.js and npm: Install Node.js, which includes npm (Node Package Manager), on your development machine. Node.js is required to run JavaScript on the server side. • Download: https://nodejs.org/en/download/

• Installation instructions: https://nodejs.org/en/download/package-manager/

MongoDB: Set up a MongoDB database to store hotel and booking information. Install MongoDB locally or use a cloud-based MongoDB service.

• Download: https://www.mongodb.com/try/download/community

• Installation instructions: https://docs.mongodb.com/manual/installation/

Express.js: Express.js is a web application framework for Node.js. Install Express.js to handle the server-side routing, middleware, and API development.

• Installation: Open your command prompt or terminal and run the following command: **npm install express**

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. To install React.js, a JavaScript library for building user interfaces, follow the installation guide: https://reactjs.org/docs/create-a-new-react-app.html

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 Angular to build the user-facing part of the application, including product listings, booking forms, and user interfaces for the

admin dashboard.

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.

• Git: Download and installation instructions can be found at: [https://git scm.com/downloads](https://gitscm.com/downloads)

Development Environment: Choose a code editor or Integrated Development Environment (IDE) that suits your preferences, such as Visual Studio Code, Sublime Text, or WebStorm.

• Visual Studio Code: Download from https://code.visualstudio.com/download

• Sublime Text: Download from https://www.sublimetext.com/download

• WebStorm: Download from https://www.jetbrains.com/webstorm/download

To Connect the Database with Node JS go through the below provided link:

Link: [https://www.section.io/engineering-education/nodejs- mongoosejs-mongodb/](https://www.section.io/engineering-education/nodejs-mongoosejs-mongodb/)

To run the existing SB Foods App project downloaded from GitHub:
Follow below steps:
Clone the repository:

• Open your terminal or command prompt.

• Navigate to the directory where you want to store the e-commerce app.

• Execute the following command to clone the repository:

Git clone: https://github.com/harsha-vardhan-reddy-07/Food-Ordering-App-MERN Install Dependencies:

• Navigate into the cloned repository directory:

    cd Food-Ordering-App-MERN

- Install the required dependencies by running the following command:

  npm install

Start the Development Server:

- To start the development server, execute the following command:

  npm run dev or npm run start

- The e-commerce app will be accessible at http://localhost:3000 by default. You can change the port configuration in the .env file if needed.

Access the App:

- Open your web browser and navigate to http://localhost:3000.

- You should see the flight booking app's homepage, indicating that the installation and setup were successful.

You have successfully installed and set up the SB Foods app on your local machine. You can now proceed with further customization, development, and testing as needed.

Application Flow:

The flow of the OrderOnTheGo application is designed to provide seamless interactions for users, restaurant owners, and administrators. Each type of user follows a specific path tailored to their role in the system.

1.User Flow

Users begin their journey by registering for an account using basic credentials such as name, email, and password. Once registered, they can log in securely and access the platform. After logging in, users can browse the available food items listed on the platform. They have the option to search and filter through products, view detailed descriptions, and select the items they wish to order. These selected items are added to the user's cart, from where they can proceed to checkout. During checkout, users provide their delivery address and choose a preferred payment method. Once the order is placed, users can track their order status and view order history within the profile section.
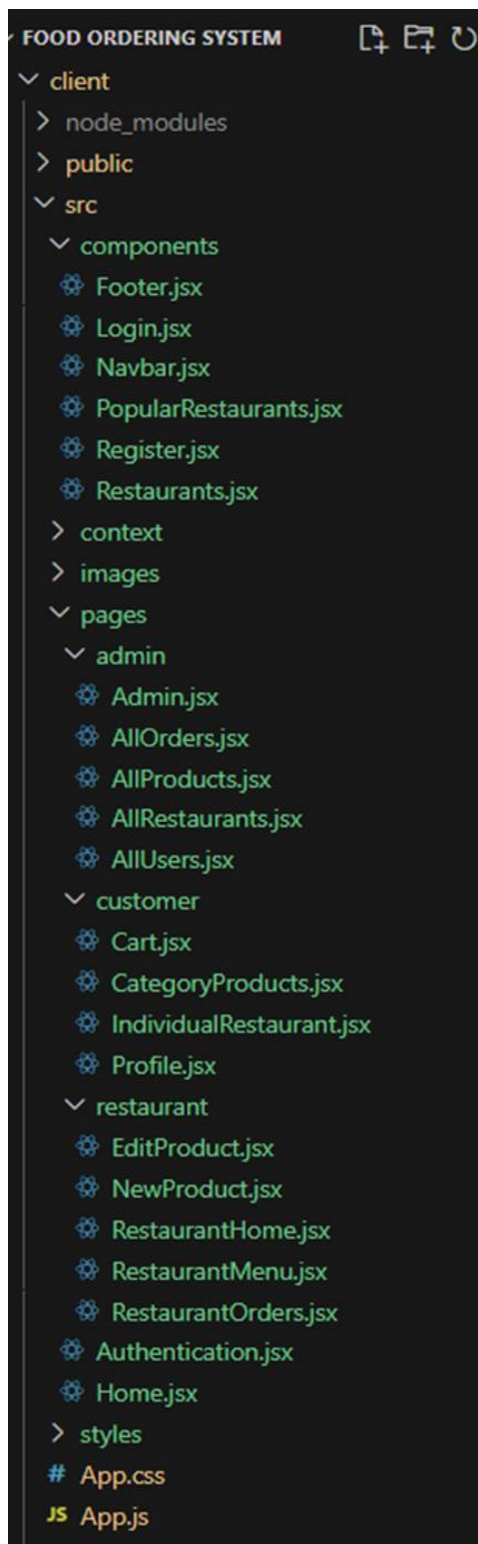
2. Restaurant Flow

Restaurant owners start by logging into the platform with their credentials. However, before they can begin listing food items, they must receive approval from the admin. Once approved, restaurants gain access to functionalities that allow them to manage their menu. They can add new food items, update existing listings, or delete items that are no longer available. This ensures that the food menu remains current and reflects the restaurant's offerings in real-time.
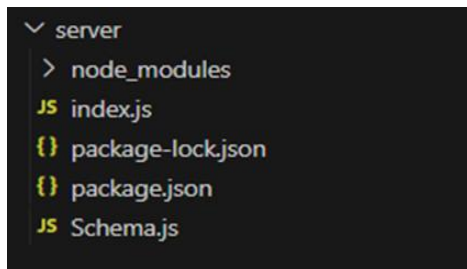
3. Admin Flow

Admins access the system by logging in with their administrator credentials. Upon successful login, they are redirected to the Admin Dashboard, which serves as a central control panel for the entire platform. From the dashboard, admins can manage and monitor various aspects of the application, including the list of users, restaurants, available food items, and placed orders. They also oversee restaurant approvals and maintain platform-wide consistency and quality.

PROJECT STRUCTURE:

Client:

FOOD ORDERING SYSTEM

- client
  - node_modules
  - public
  - src
    - components
      - Footer.jsx
      - Login.jsx
      - Navbar.jsx
      - PopularRestaurants.jsx
      - Register.jsx
      - Restaurants.jsx
    - context
    - images
    - pages
      - admin
        - Admin.jsx
        - AllOrders.jsx
        - AllProducts.jsx
        - AllRestaurants.jsx
        - AllUsers.jsx
      - customer
        - Cart.jsx
        - CategoryProducts.jsx
        - IndividualRestaurant.jsx
        - Profile.jsx
      - restaurant
        - EditProduct.jsx
        - NewProduct.jsx
        - RestaurantHome.jsx
        - RestaurantMenu.jsx
        - RestaurantOrders.jsx
      - Authentication.jsx
      - Home.jsx
    - styles
    - App.css
    - App.js

Server:

The backend of the project is organized under the server folder, which contains the core files needed to run the Node.js application. The index.js file acts as the main entry point, where the Express server is configured along with routes and database connections. The Schema.js file defines Mongoose schemas for handling various collections like users, orders, and food items in MongoDB. The package.json and package-lock.json files manage the project's metadata and dependencies. The node modules folder stores all the installed packages required for the server to function. This minimal and clean structure provides a solid foundation for managing and scaling the backend effectively.

Running the Application:

- **Frontend running**
  1. Open the command prompt
  2. Navigate to directory where your front end files are located
  3. Give the command 'npm install' install all dependencies
  4. Give the command 'npn run'
  5. The application is run on local host 3000

- **Backend running**
  1. Open the another window on the same command prompt
  2. Navigate to the directory where your backend files are located
  3. Give the same command 'npm install' and install all dependencies
  4. Connect the current Ip address in your database
  5. Give the command 'npm run' and run the backend files
  6. The application run on the browser with out any error

PROJECT SETUP AND CONFIGURATION:

Install required tools and software:
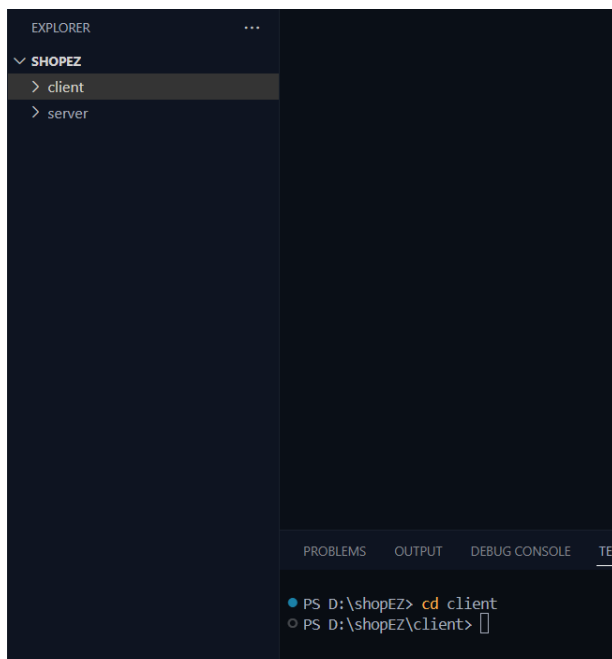
• Node.js.

Reference Article: https://www.geeksforgeeks.org/installation-of-node-js-on-windows/

• Git.

Reference Article: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Create project folders and files:

• Client folders.

• Server folders

Referral Video Link: https://drive.google.com/file/d/1uSMbPIAR6rfAEMcb_nLZAZd5QIjTpnYQ/view?usp=sharing

Referral Image:

DATABASE DEVELOPMENT:

Create database in cloud video link:https://drive.google.com/file/d/1CQil5KzGnPvkVOPWTLP0h-Bu2bXhq7A3/view

• Install Mongoose.

• Create database connection.

Reference Video of connect node with MongoDB database: https://drive.google.com/file/d/1cTS3_-EOAAvDctkibG5zVikrTdmoY2Ag/view?usp=sharing

Reference Article: https://www.mongodb.com/docs/atlas/tutorial/connect-to-your-cluster/

Reference Image:

Schema use-case:

1. User Schema:

• Schema: user Schema

• Model: 'User'

• The User schema represents the user data and includes fields such as username, email, and password.

2. Product Schema:

• Schema: product Schema

• Model: 'Product'

• The Product schema represents the data of all the products in the platform.

• It is used to store information about the product details, which will later be useful for ordering.

3. Orders Schema:

• Schema: orders Schema

• Model: 'Orders'

• The Orders schema represents the orders data and includes fields such as user Id, product Id, product name, quantity, size, order date, etc.,

4. Cart Schema:

• Schema: cart Schema

• Model: 'Cart'

• The Cart schema represents the cart data and includes fields such as user Id, product Id, product name, quantity, size, order date, etc.,

• The user Id field is a reference to the user who has the product in cart.

5. Admin Schema:

• Schema: admin Schema

• Model: 'Admin'

• The admin schema has essential data such as categories, promoted restaurants,

etc.,

6. Restaurant Schema:

- Schema: restaurant Schema

- Model: 'Restaurant'

- The restaurant schema has the info about the restaurant and it's menu

Schemas: Now let us define the required schemas

```javascript
JS Schema.js ×
server > JS Schema.js > [@] orderSchema
  1   import mongoose from "mongoose";
  2
  3   const userSchema = new mongoose.Schema({
  4       username: {type: String},
  5       password: {type: String},
  6       email: {type: String},
  7       usertype: {type: String},
  8       approval: {type: String}
  9   });
 10
 11   const adminSchema = new mongoose.Schema({
 12       categories: {type: Array},
 13       promotedRestaurants: []
 14   });
 15
 16   const restaurantSchema = new mongoose.Schema({
 17       ownerId: {type: String},
 18       title: {type: String},
 19       address: {type: String},
 20       mainImg: {type: String},
 21       menu: {type: Array, default: []}
 22   })
 23
 24   const foodItemSchema = new mongoose.Schema({
 25       title: {type: String},
 26       description: {type: String},
 27       itemImg: {type: String},
 28       category: {type: String}, //veg or non-veg or beverage
 29       menuCategory: {type: String},
 30       restaurantId: {type: String},
 31       price: {type: Number},
 32       discount: {type: Number},
 33       rating: {type: Number}
 34   })
 35
```
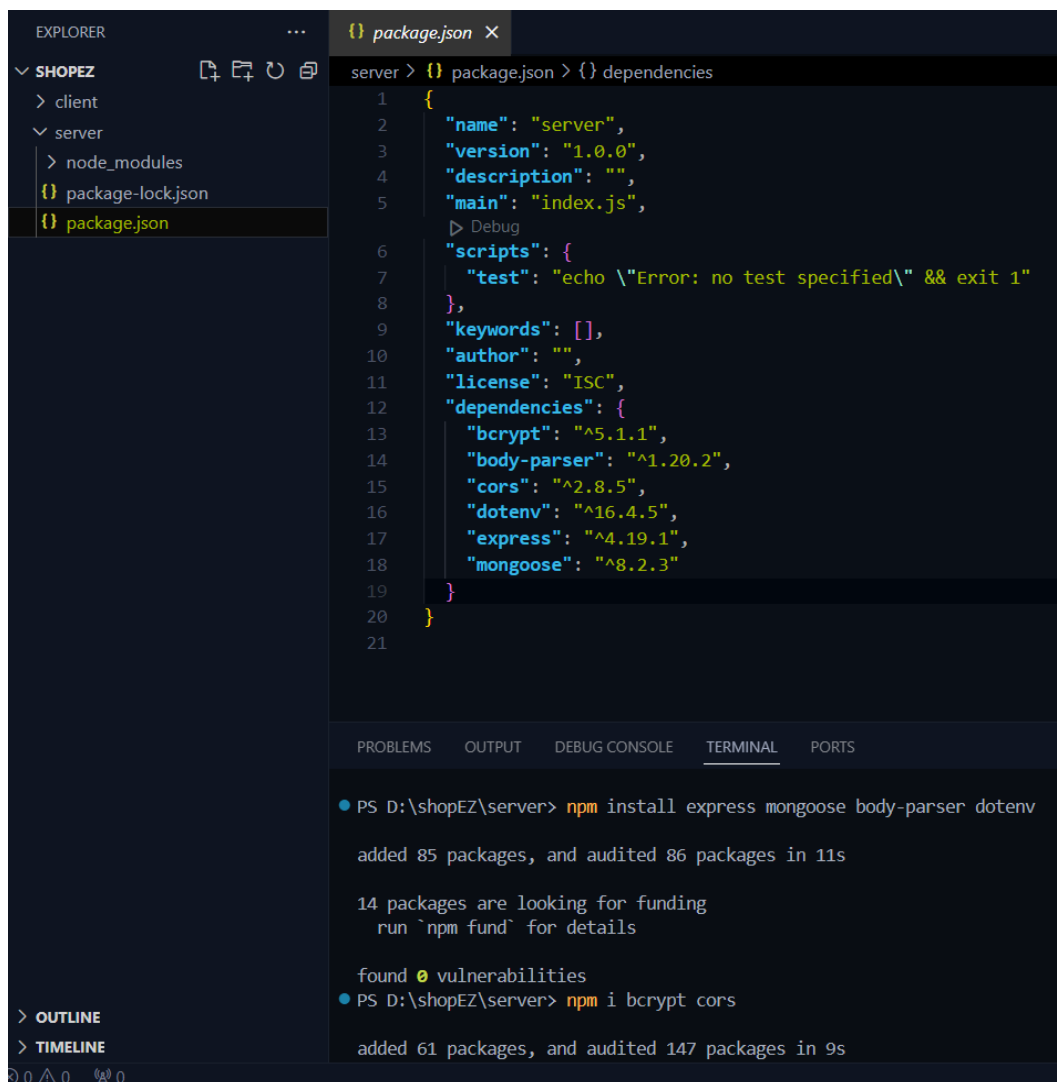
BACKEND DEVELOPMENT:

Set Up Project Structure:

- Create a new directory for your project and set up a package.json file using
  the npm init command.

• Install necessary dependencies such as Express.js, Mongoose, and other
   required packages.

Reference Video:

Reference Image:

1. Setup express server:

• Create index.js file.

• Create an express server on your desired port number.

• Define API's

Reference Image:

2. Database Configuration:

• Set up a MongoDB database either locally or using a cloud-based MongoDB
service like MongoDB Atlas or use locally with MongoDB compass.

• Create a database and define the necessary collections for admin, users,
restaurants, food products, orders, and other relevant data.

Reference Image:

3. Create Express.js Server:

• Set up an Express.js server to handle HTTP requests and serve API endpoints.

• Configure middleware such as body-parser for parsing request bodies and cors for handling cross-origin requests.

Reference Video:
https://drive.google.com/file/d/1uKMIcrok_ROHyZl2vRORggrYRio2qXS/view?usp=sharing

Reference Image:

4. Define API Routes:

- Create separate route files for different API functionalities such as users, orders, and authentication.

- Define the necessary routes for listing products, handling user registration and login managing orders, etc.

- Implement route handlers using Express.js to handle requests and interact with the database.

5. Implement Data Models:

- Define Mongoose schemas for the different data entities like products, users, and orders.

- Create corresponding Mongoose models to interact with the MongoDB database.

- Implement CRUD operations (Create, Read, Update, Delete) for each model to perform database operations.

6. User Authentication:

- Create routes and middleware for user registration, login, and logout.

• Set up authentication middleware to protect routes that require user authentication.

7. Handle new products and Orders:

- • Create routes and controllers to handle new product listings, including fetching products data from the database and sending it as a response.

- • Implement ordering(buy) functionality by creating routes and controllers to handle order requests, including validation and database updates.

8. Admin Functionality:

- • Implement routes and controllers specific to admin functionalities such as adding products, managing user orders, etc.

- • Add necessary authentication and authorization checks to ensure only authorized admins can access these routes.

9. Error Handling:

- • Implement error handling middleware to catch and handle any errors that occur during the API requests.

- • Return appropriate error responses with relevant error messages and HTTP status codes.

FRONTEND DEVELOPMENT:

1. Setup React Application:

- • Create a React app in the client folder.
- • Install required libraries
- • Create required pages and components and add routes.

2.Design UI components:

- • Create Components.
- • Implement layout and styling.
- • Add navigation.

3.Implement frontend logic:

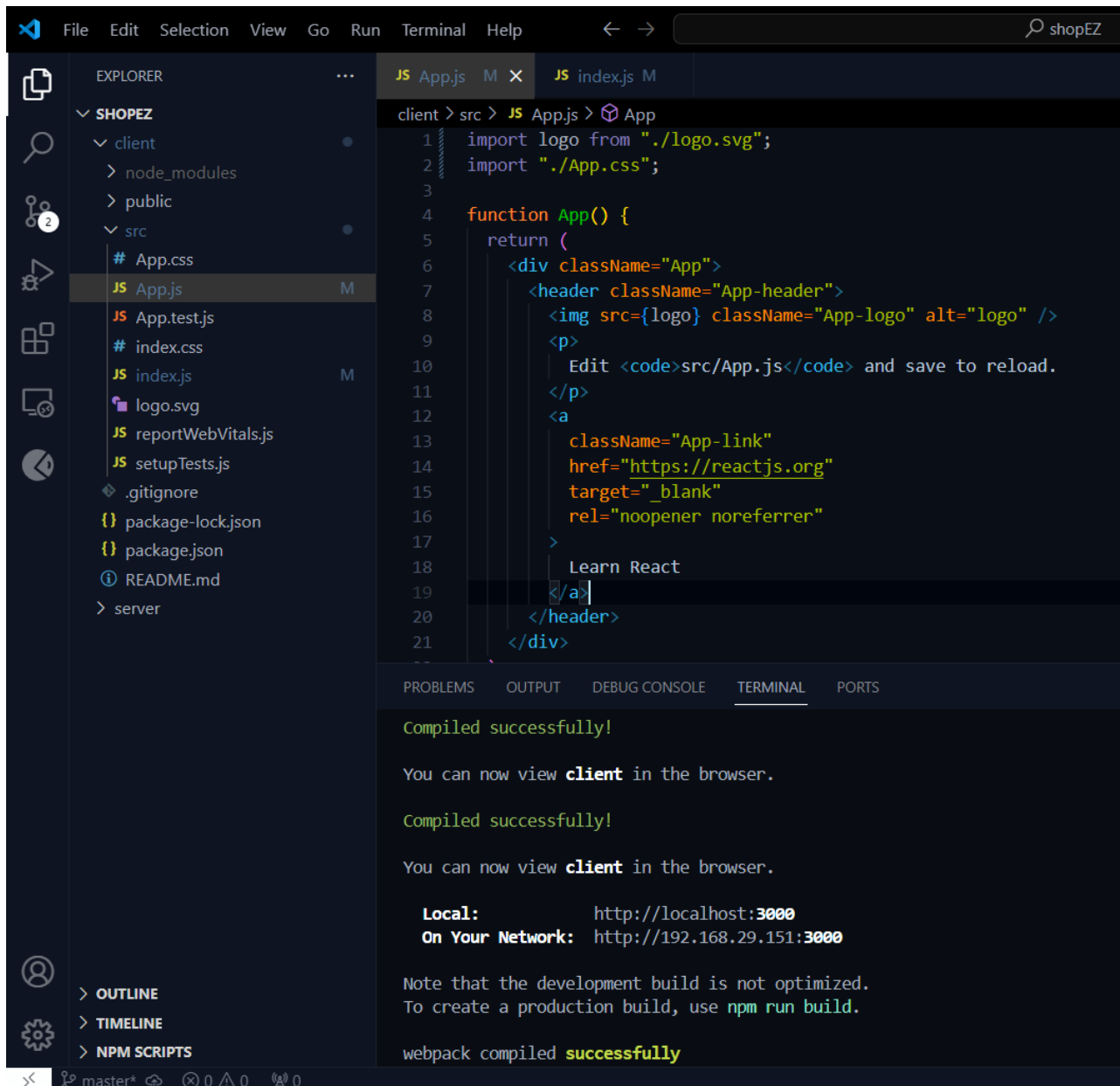- Integration with API endpoints.
- Implement data binding.

Reference Video Link:

https://drive.google.com/file/d/1EokogagcLMUGiIluwHGYQo65x8GRpDcP/view?usp=sharing

Reference Article Link:

https://www.w3schools.com/react/react_getstarted.asp

Reference Image:

CODE EXPLANATION:

Server setup:

Let us import all the required tools/libraries and connect the database.

```js
JS index.js    ×

server > JS index.js > ...
    1    import express from 'express'
    2    import bodyParser from 'body-parser';
    3    import mongoose from 'mongoose';
    4    import cors from 'cors';
    5    import bcrypt from 'bcrypt';
    6    import {Admin, Cart, FoodItem, Orders, Restaurant, User } from './Schema.js'
    7
    8
    9    const app = express();
   10
   11    app.use(express.json());
   12    app.use(bodyParser.json({limit: "30mb", extended: true}))
   13    app.use(bodyParser.urlencoded({limit: "30mb", extended: true}));
   14    app.use(cors());
   15
   16    const PORT = 6001;
   17
   18    mongoose.connect('mongodb://localhost:27017/foodDelivery',{
   19        useNewUrlParser: true,
   20        useUnifiedTopology: true
   21    }).then(()=>{
   22
```

User Authentication:

Backend

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

```js
JS index.js    ×

server > JS index.js > 🔾 then() callback
   55
   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 ✕

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 ✕

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 ✕

client > src > context > GeneralContext.jsx > [∅] GeneralContextProvider > [∅] 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 ✕

client > src > context > JS GeneralContext.js > [∅] GeneralContextProvider > [∅] 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      }
10?
```

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:

```
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:

```jsx
  const [sortFilter, setSortFilter] = useState('popularity');
  const [categoryFilter, setCategoryFilter] = useState([]);
  const [genderFilter, setGenderFilter] = useState([]);

  const handleCategoryCheckBox = (e) =>{
    const value = e.target.value;
    if(e.target.checked){
      setCategoryFilter([...categoryFilter, value]);
    }else{
      setCategoryFilter(categoryFilter.filter(size=> size !== value));
    }
  }

  const handleGenderCheckBox = (e) =>{
    const value = e.target.value;
    if(e.target.checked){
      setGenderFilter([...genderFilter, value]);
    }else{
      setGenderFilter(genderFilter.filter(size=> size !== value));
    }
  }

  const handleSortFilterChange = (e) =>{
    const value = e.target.value;
    setSortFilter(value);
    if(value === 'low-price'){
      setVisibleProducts(visibleProducts.sort((a,b)=> a.price - b.price))
    } else if (value === 'high-price'){
      setVisibleProducts(visibleProducts.sort((a,b)=> b.price - a.price))
    }else if (value === 'discount'){
      setVisibleProducts(visibleProducts.sort((a,b)=> b.discount - a.discount))
    }
  }

  useEffect(()=>{

      if (categoryFilter.length > 0 && genderFilter.length > 0){
        setVisibleProducts(products.filter(product=> categoryFilter.includes(product.category) && genderFilter.includes(product.gender) ));
      }else if(categoryFilter.length === 0 && genderFilter.length > 0){
        setVisibleProducts(products.filter(product=> genderFilter.includes(product.gender) ));
      } else if(categoryFilter.length > 0 && genderFilter.length === 0){
        setVisibleProducts(products.filter(product=> categoryFilter.includes(product.category)));
      }else{
        setVisibleProducts(products);
      }

  }, [categoryFilter, genderFilter])
```

· **Backend**

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

```js
    // fetch products

    app.get('/fetch-products', async(req, res)=>{
        try{
            const products = await Product.find();
            res.json(products);

        }catch(err){
            res.status(500).json({ message: 'Error occured' });
        }
    })
```

Add product to cart:

- **Frontend**

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

```jsx
IndividualRestaurant.jsx U ×

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**
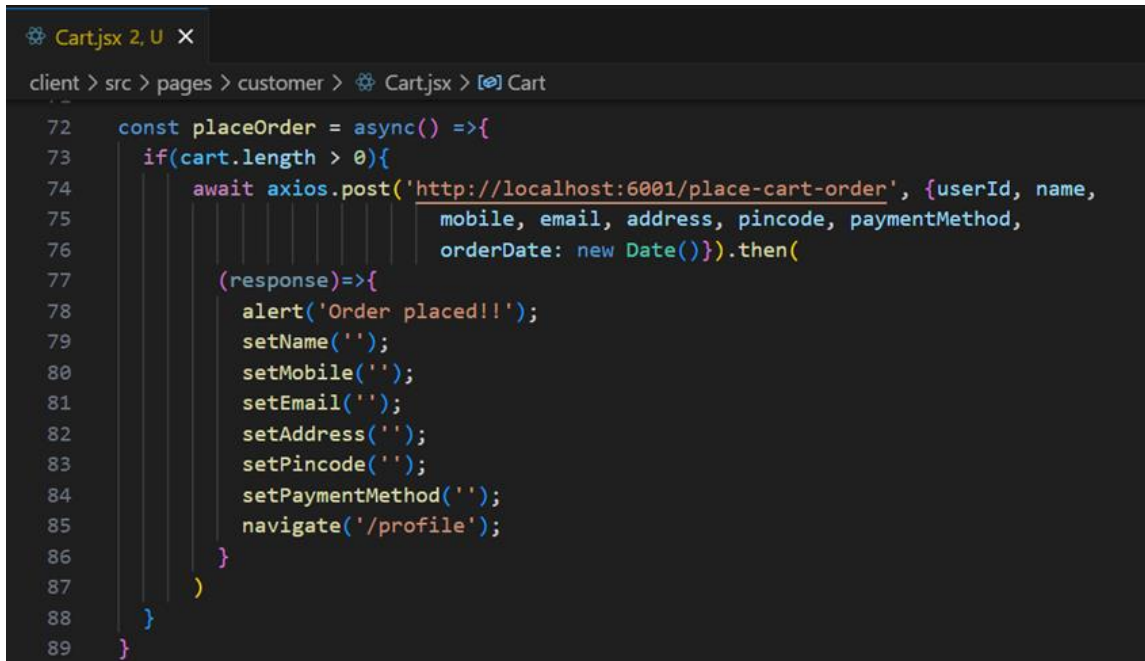
Add product to cart:

```js
JS index.js    ×

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 occured"});
416        }
417    })
418
```

Order products:

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

· Frontend

```jsx
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 use

```
JS index.js    ✕

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 occured"});
456          }
457      })
458
```

Add new product:

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

· Frontend:

```
❖ NewProduct.jsx 1, U ✕

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
```
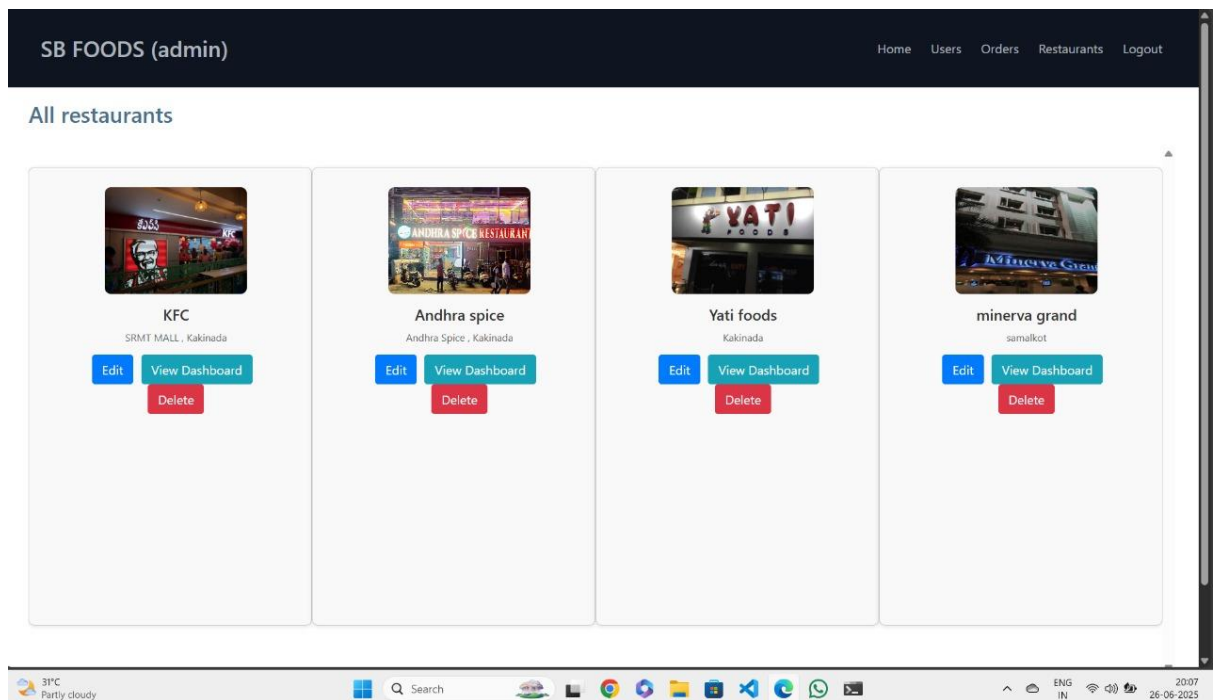
· **Backend:**


```js
// Add new product
app.post('/add-new-product', async(req, res)=>{
    const {restaurantId, productName, productDescription,
            productMainImg, productCategory, productMenuCategory,
            productNewCategory, productPrice, productDiscount} = req.body;
    try{
        if(productMenuCategory === 'new category'){
            const admin = await Admin.findOne();
            admin.categories.push(productNewCategory);
            await admin.save();
            const newProduct = new FoodItem({restaurantId, title: productName,
                        description: productDescription, itemImg: productMainImg,
                        category: productCategory, menuCategory: productNewCategory,
                        price: productPrice, discount: productDiscount, rating: 0});
            await newProduct.save();
            const restaurant = await Restaurant.findById(restaurantId);
            restaurant.menu.push(productNewCategory);
            await restaurant.save();
        } else{
            const newProduct = new FoodItem({restaurantId, title: productName,
                        description: productDescription, itemImg: productMainImg,
                        category: productCategory, menuCategory: productMenuCategory,
                        price: productPrice, discount: productDiscount, rating: 0});
            await newProduct.save();
        }
        res.json({message: "product added!!"});
    }catch(err){
        res.status(500).json({message: "Error occured"});
    }
})
```

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

Demo UI images:
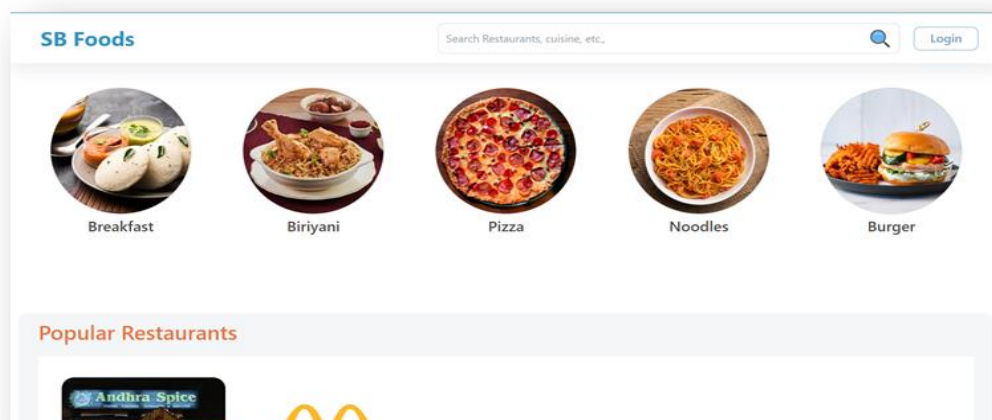
Restaurants:

Restaurant Menu:
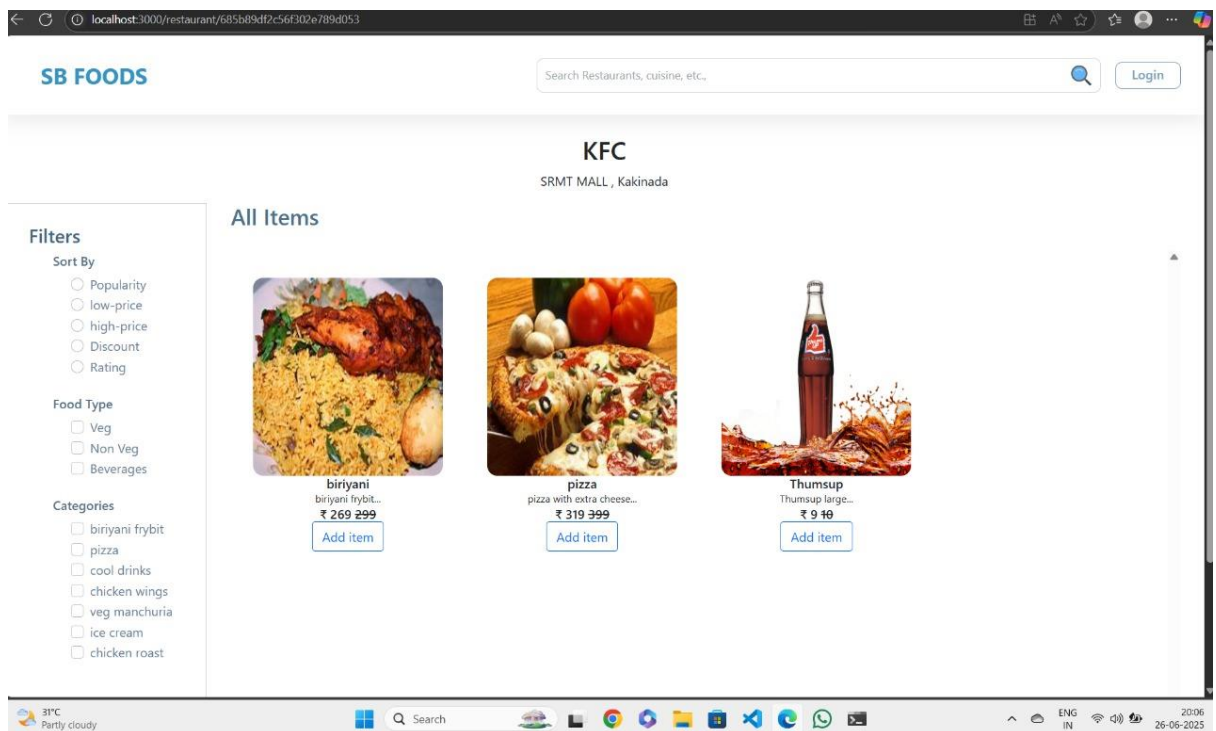
Authentication:

User Profile:

Cart:
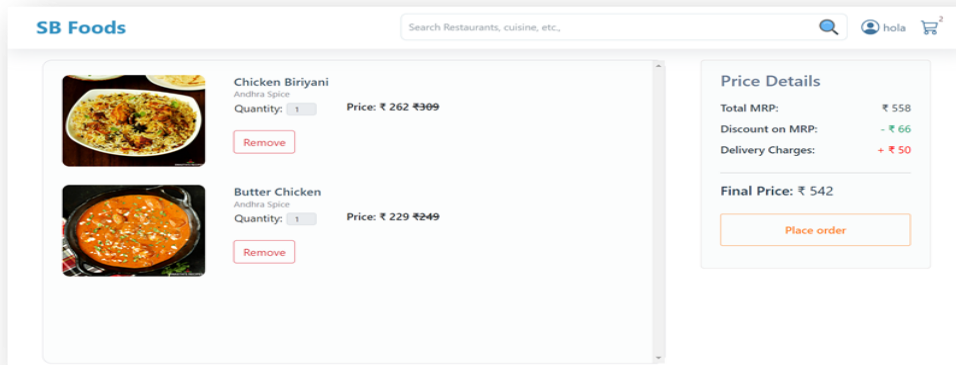
Admin Dashboard:
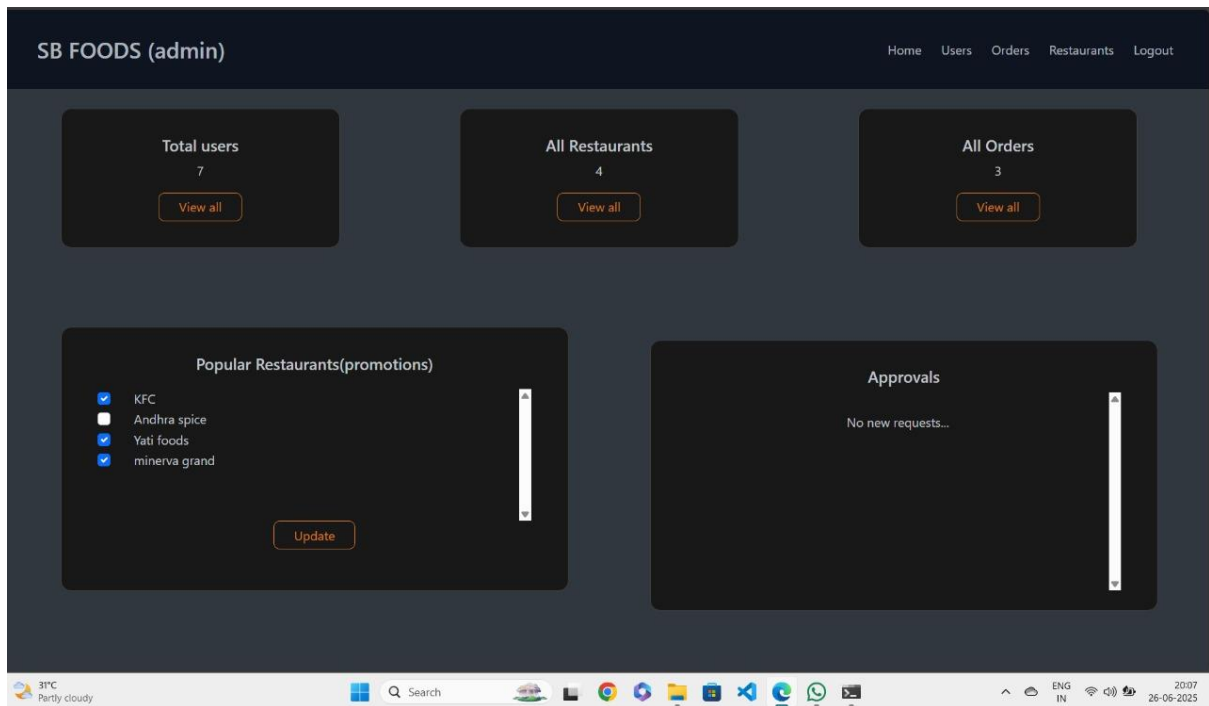
All orders:
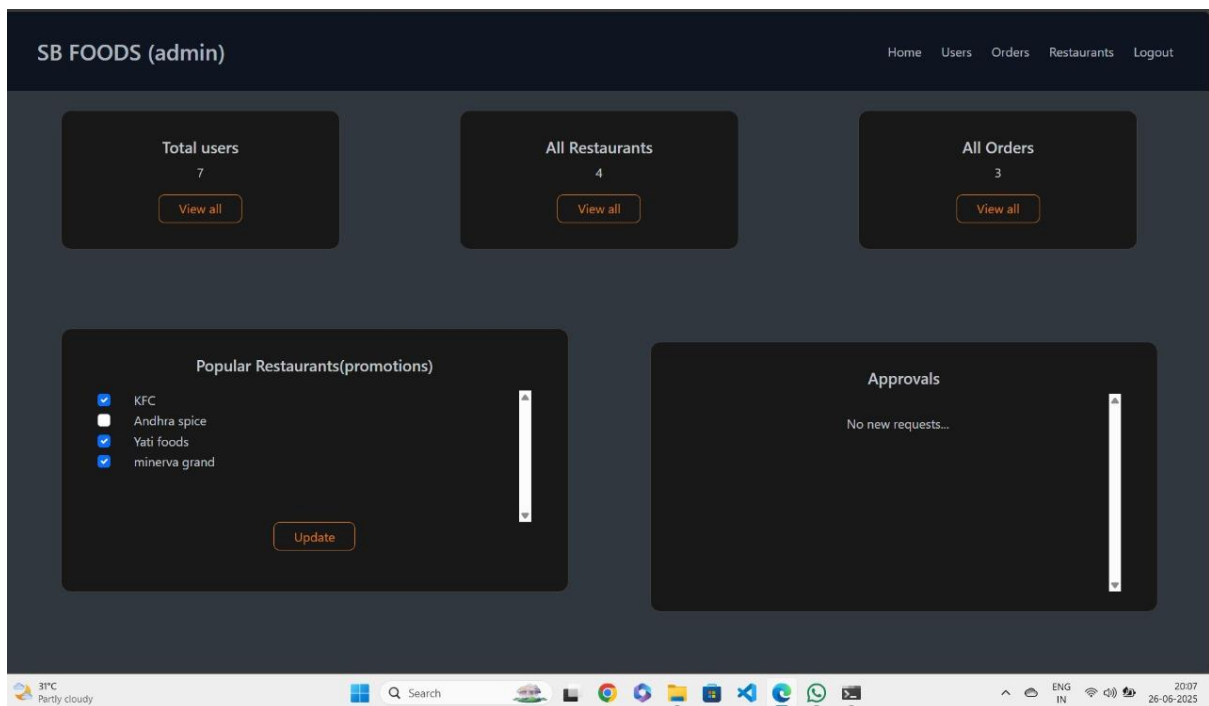
All Restaurants:
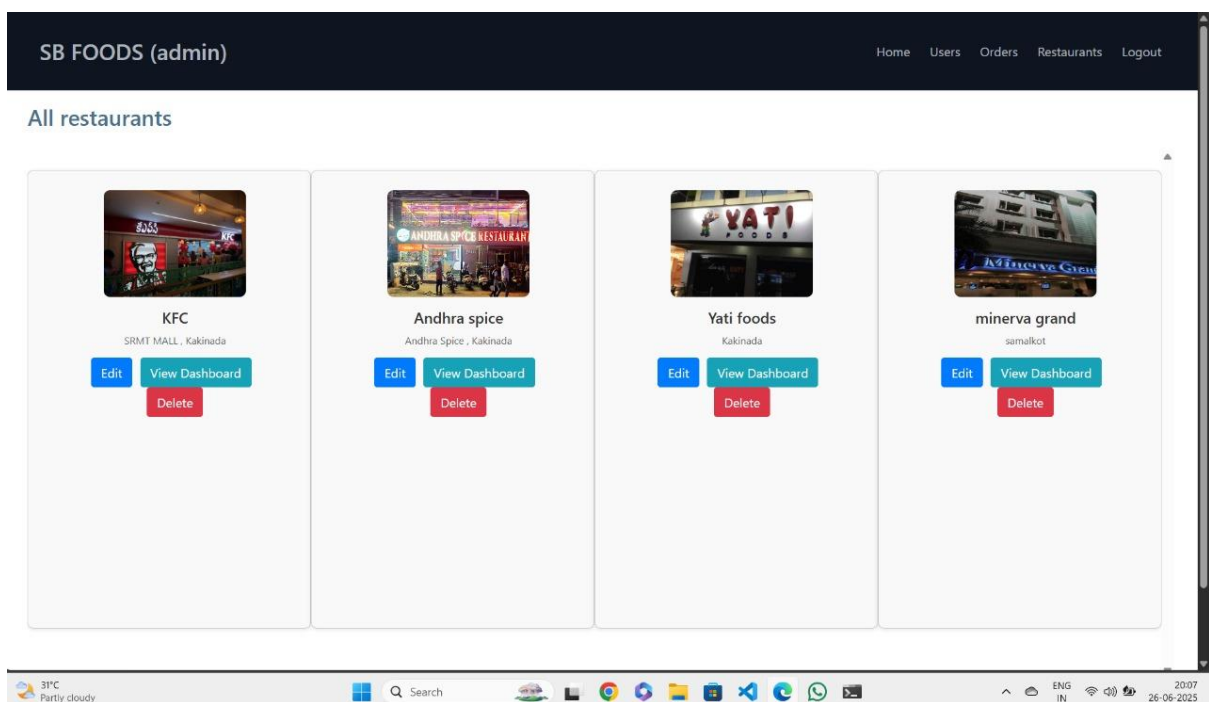
Restaurant Dashboard:
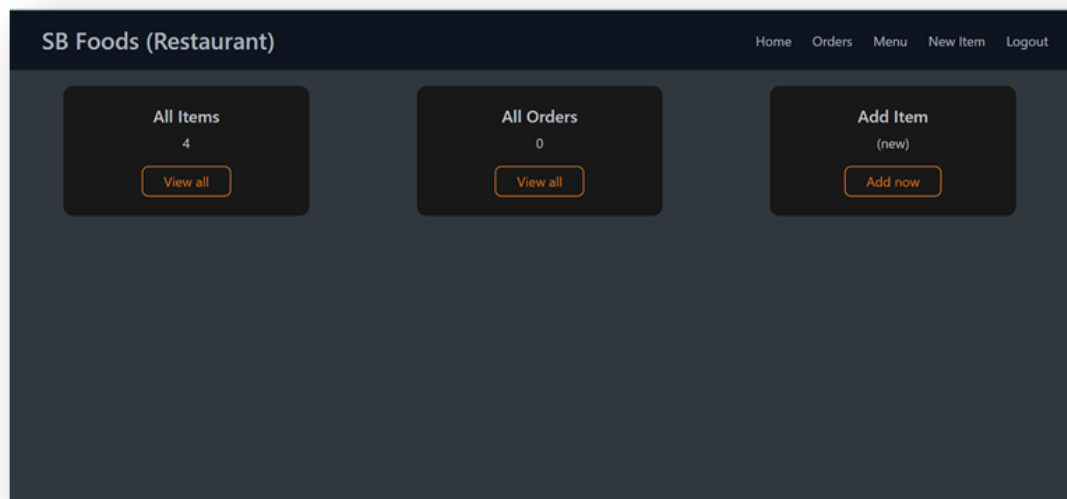
New Item:

Future Enhancement:

Future Scope:

 The following section describes the work that will be implemented with future releases of the software.

 • Customize orders: Allow customers to customize food orders

• Enhance User Interface by adding more user interactive features. Provide Deals and

promotional Offer details to home page. Provide Recipes of the Week/Day to Home Page

 • Payment Options: Add different payment options such as PayPal, Cash, Gift Cards etc. Allow to save payment details for future use.

 • Allow to process an order as a Guest

 • Delivery Options: Add delivery option

 • Order Process Estimate: Provide customer a visual graphical order status bar

 • Order Status: Show only Active orders to Restaurant Employees.

 • Order Ready notification: Send an Order Ready notification to the customer

• Restaurant Locator: Allow to find and choose a nearby restaurant

 • Integrate with In store touch screen devices like iPad