Orderonthego-your-on-demand-food-ordering-solution Application

INTRODUCTION

Introducing SB Foods, the cutting-edge digital platform poised to revolutionize the way you order food online. With SB Foods, your food ordering experience will reach unparalleled levels of convenience and efficiency.

Our user-friendly web app empowers foodies to effortlessly explore, discover, and order dishes tailored to their unique tastes. Whether you're a seasoned food enthusiast or an occasional diner, finding the perfect meals has never been more straightforward.

Imagine having comprehensive details about each dish at your fingertips. From dish descriptions and customer reviews to pricing and available promotions, you'll have all the information you need to make well-informed choices. No more second-guessing or uncertainty – SB Foods ensures that every aspect of your online food ordering journey is crystal clear.

The ordering process is a breeze. Just provide your name, delivery address, and preferred payment method, along with your desired dishes. Once you place your order, you'll receive an instant confirmation. No more waiting in long queues or dealing with complicated ordering processes – SB Foods streamlines it, making it quick and hassle-free.

SCENARIO:

Late-Night Craving Resolution

Meet Lisa, a college student burning the midnight oil to finish her assignment. As the clock strikes midnight, her stomach grumbles, reminding her that she skipped dinner. Lisa doesn't want to interrupt her workflow by cooking, nor does she have the energy to venture outside in search of food.

Solution with Food Ordering App:

1. Lisa opens the Food Ordering App on her smartphone and navigates to the late-night delivery section, where she finds a variety of eateries still open for orders.

2. She scrolls through the options, browsing menus and checking reviews until she spots her favorite local diner offering comfort food classics.

3. Lisa selects a hearty bowl of chicken noodle soup and a side of garlic bread, craving warmth and satisfaction in each bite.

4. With a few taps, she adds the items to her cart, specifies her delivery address, and chooses her preferred payment method.

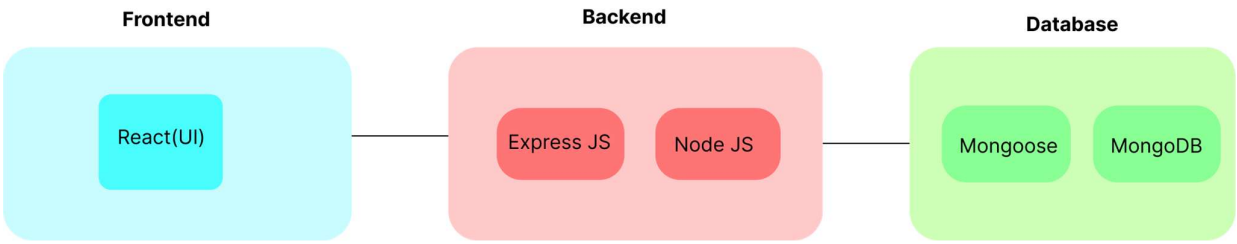
5. Lisa double-checks her order details on the confirmation page, ensuring everything looks correct, before tapping the "Place Order" button.

6. Within minutes, she receives a notification confirming her order and estimated delivery time, allowing her to continue working with peace of mind.

7. As promised, the delivery arrives promptly at her doorstep, and Lisa eagerly digs into her piping hot meal, grateful for the convenience and comfort provided by the Food Ordering App during her late-night study session.

This scenario illustrates how a Food Ordering App caters to users' needs, even during unconventional hours, by offering a seamless and convenient solution for satisfying late-night cravings without compromising on quality or convenience.

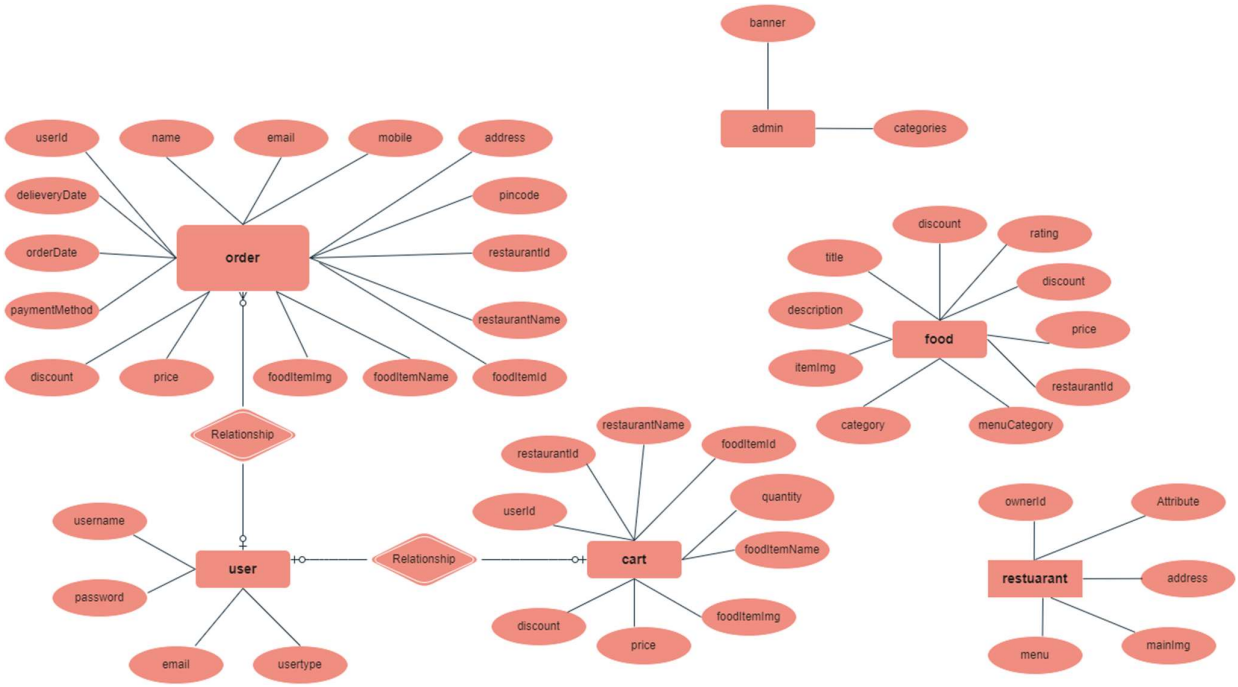
TECHNICAL ARCHITECTURE:

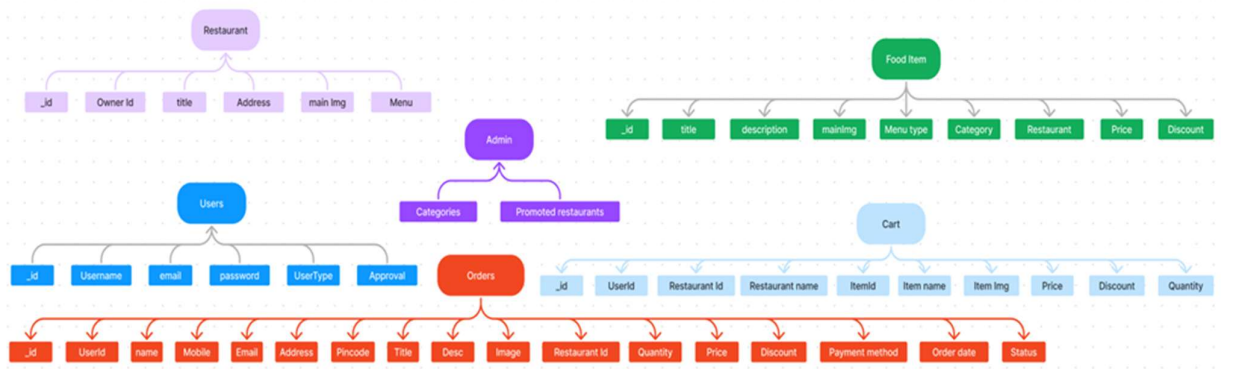


In this architecture diagram:

- 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 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.

FEATURES:

1. **Comprehensive Product Catalog:** SB Foods boasts an extensive catalog of food items from various restaurants, offering a diverse range of items and options for shoppers. You can effortlessly explore and discover various products, complete with detailed descriptions, customer reviews, pricing, and available discounts, to find the perfect food for your hunger.

2. **Order Details Page**: Upon clicking the "Shop Now" button, you will be directed to an order details page. Here, you can provide relevant information such as your shipping address, preferred payment method, and any specific product requirements.

3. **Secure and Efficient Checkout Process:** SB Foods guarantees a secure and efficient checkout process. Your personal information will be handled with the utmost security, and we strive to make the purchasing process as swift and trouble-free as possible.

4. **Order Confirmation and Details:** After successfully placing an order, you will receive a confirmation notification. Subsequently, you will be directed to an order details page, where you can review all pertinent information about your order, including shipping details, payment method, and any specific product requests you specified.

In addition to these user-centric features, SB Foods provides a robust restaurant dashboard, offering restaurants an array of functionalities to efficiently manage their products and sales. With the restaurant dashboard, restaurants can add and oversee multiple product listings, view order history, monitor customer activity, and access order details for all purchases.

SB Foods is designed to elevate your online food ordering experience by providing a seamless and user-friendly way to discover your desired foods. With our efficient checkout process, comprehensive product catalog, and robust restaurant dashboard, we ensure a convenient and enjoyable online shopping experience for both shoppers and restaurants alike.

PREREQUISITES:

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 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

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/

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/subbu-vishnu-86/orderonthego-your-on-demand-food-ordering-solution.git**

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.

USER & ADMIN FLOW:

1. User Flow:

• Users start by registering for an account.

• After registration, they can log in with their credentials.

• Once logged in, they can check for the available products in the platform. • Users can add the products they wish to their carts and order.

• They can then proceed by entering address and payment details. • After ordering, they can check them in the profile section.

2. Restaurant Flow:

• Restaurants start by authenticating with their credentials.
• They need to get approval from the admin to start listing the products. • They can add/edit the food items.

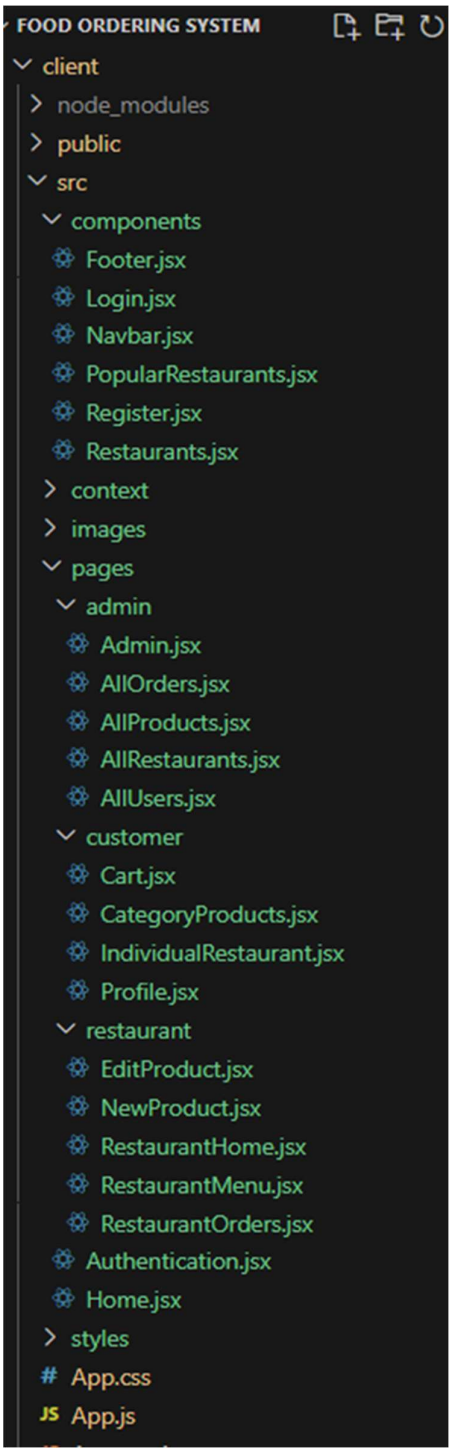
3. Admin Flow:

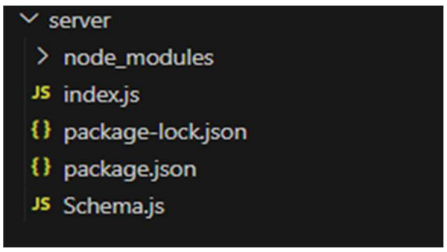
• Admins start by logging in with their credentials.

• Once logged in, they are directed to the Admin Dashboard.

• Admins can access the users list, products, orders, etc.

PROJECT STRUCTURE





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.

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

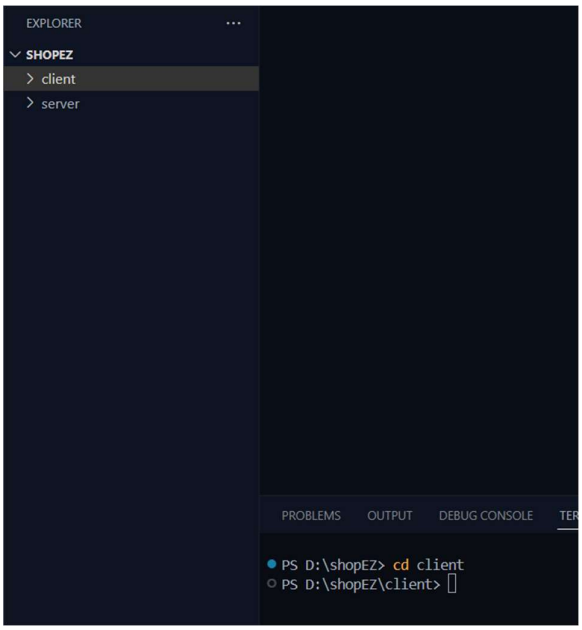
Create project folders and files:

- Client folders.
- Server folders

Referral Video Link:

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

Referral Image:



DATABASE DEVELOPMENT:

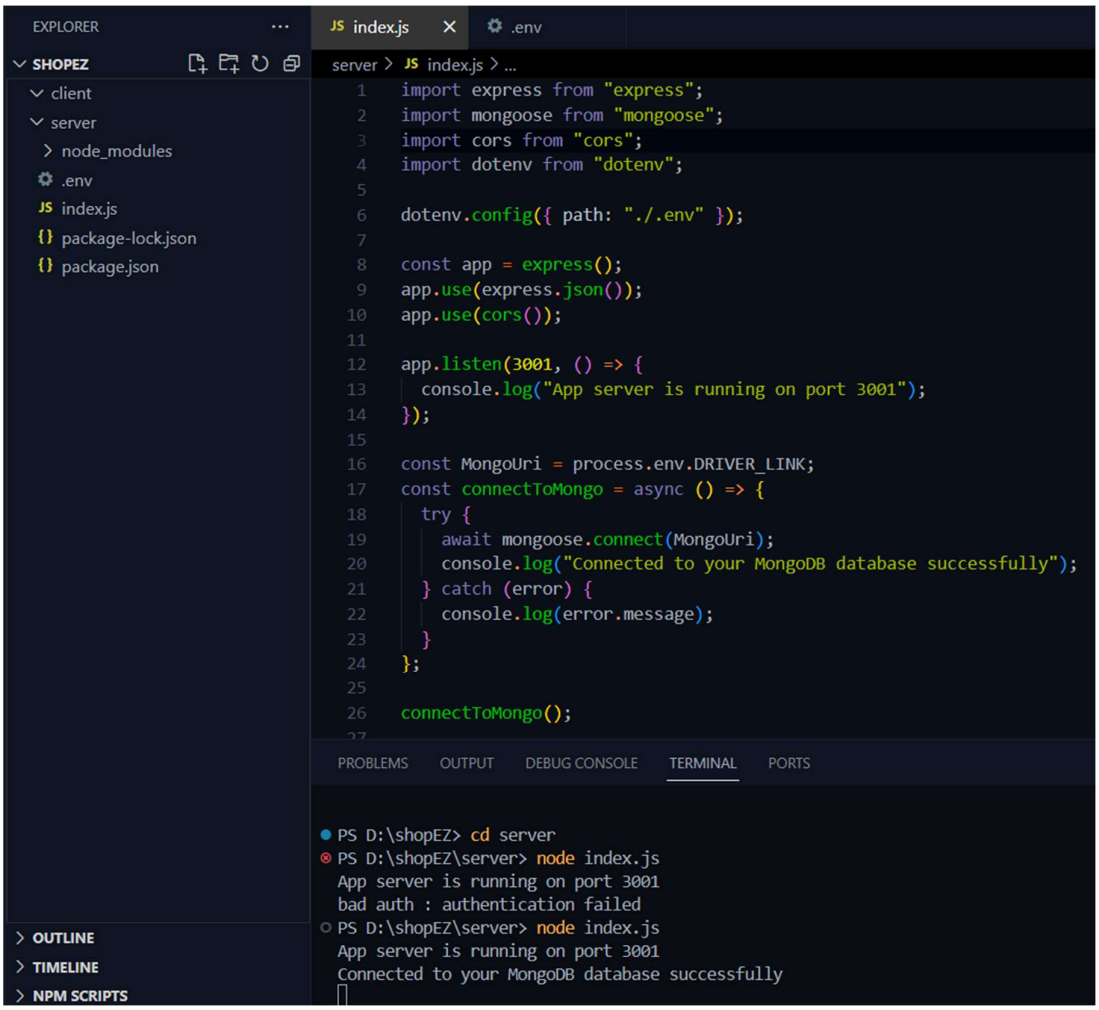
Create database in cloud video link:-
https://drive.google.com/file/d/1CQil5KzGnPvkVOPWTLP0h-Bu2bXhq7A3/view

- 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: userSchema
   - Model: ‘User’
   - The User schema represents the user data and includes fields such as username, email, and password.
2. **Product Schema:**
   - Schema: productSchema
   - 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: ordersSchema
- Model: ‘Orders’
- The Orders schema represents the orders data and includes fields such as userId, product Id, product name, quantity, size, order date, etc.,

4. Cart Schema:

- Schema: cartSchema
- Model: ‘Cart’
- The Cart schema represents the cart data and includes fields such as userId, 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: adminSchema
- Model: ‘Admin’
- The admin schema has essential data such as categories, promoted restaurants, etc.,

6. Restaurant Schema:

- Schema: restaurantSchema
- Model: ‘Restaurant’
- The restaurant schema has the info about the restaurant and it’s menu

Schemas: Now let us define the required schemas

Schema.js

server > Schema.js > orderSchema

```
import mongoose from "mongoose";

const userSchema = new mongoose.Schema({
    username: {type: String},
    password: {type: String},
    email: {type: String},
    usertype: {type: String},
    approval: {type: String}
});

const adminSchema = new mongoose.Schema({
    categories: {type: Array},
    promotedRestaurants: []
});

const restaurantSchema = new mongoose.Schema({
    ownerId: {type: String},
    title: {type: String},
    address: {type: String},
    mainImg: {type: String},
    menu: {type: Array, default: []}
})

const foodItemSchema = new mongoose.Schema({
    title: {type: String},
    description: {type: String},
    itemImg: {type: String},
    category: {type: String}, //veg or non-veg or beverage
    menuCategory: {type: String},
    restaurantId: {type: String},
    price: {type: Number},
    discount: {type: Number},
    rating: {type: Number}
})

```

Schema.js

server > Schema.js > ...

```
const orderSchema = new mongoose.Schema({
    userId: {type: String},
    name: {type: String},
    email: {type: String},
    mobile: {type: String},
    address: {type: String},
    pincode: {type: String},
    restaurantId: {type: String},
    restaurantName: {type: String},
    foodItemId: {type: String},
    foodItemName: {type: String},
    foodItemImg: {type: String},
    quantity: {type: Number},
    price: {type: Number},
    discount: {type: Number},
    paymentMethod: {type: String},
    orderDate: {type: String},
    orderStatus: {type: String, default: 'order placed'}
})

const cartSchema = new mongoose.Schema({
    userId: {type: String},
    restaurantId: {type: String},
    restaurantName: {type: String},
    foodItemId: {type: String},
    foodItemName: {type: String},
    foodItemImg: {type: String},
    quantity: {type: Number},
    price: {type: Number},
    discount: {type: Number}
})

export const User = mongoose.model('users', userSchema);
export const Admin = mongoose.model('admin', adminSchema);
export const Restaurant = mongoose.model('restaurant', restaurantSchema);
export const FoodItem = mongoose.model('foodItem', foodItemSchema);
export const Orders = mongoose.model('orders', orderSchema);
export const Cart = mongoose.model('cart', cartSchema);

```

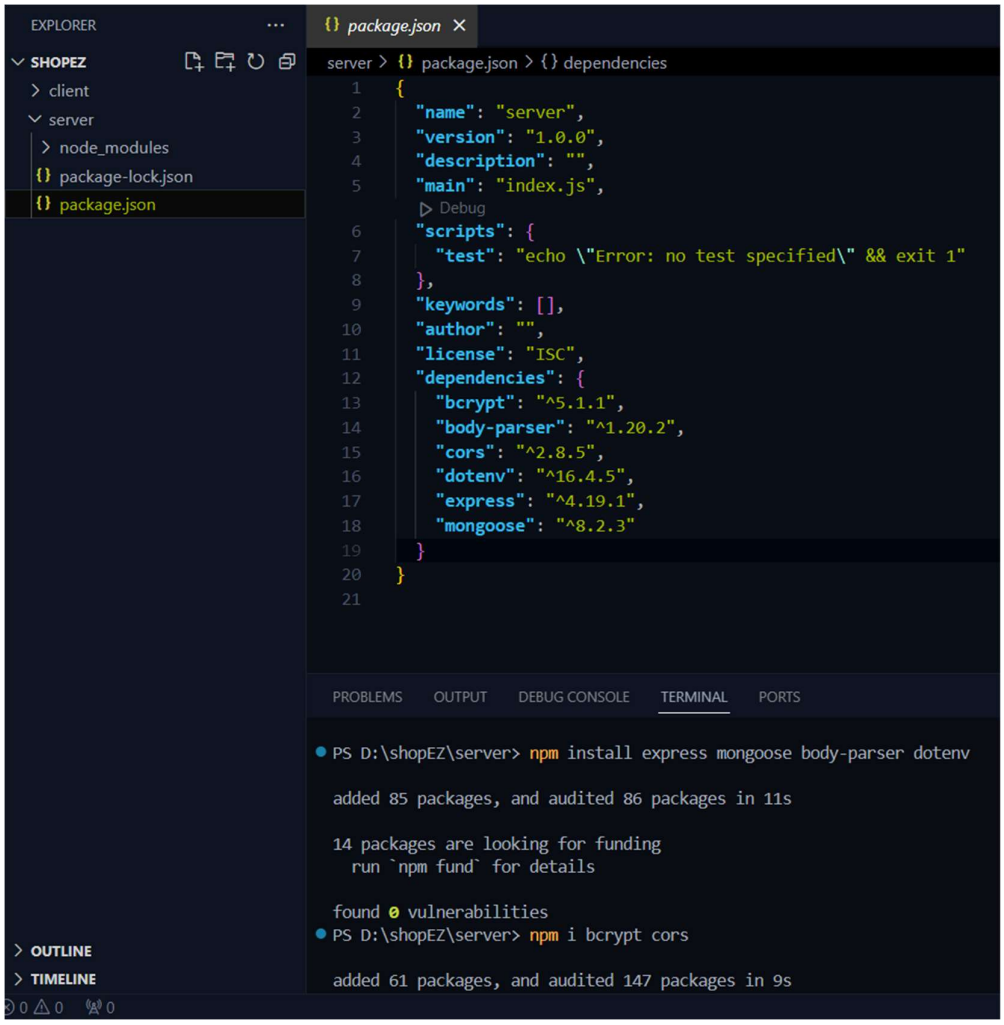
BACKEND DEVELOPMENT:

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: https://drive.google.com/file/d/19df7NU-gQK3DO6wr7ooAfJYIQwnemZoF/view?usp=sharing

Reference Image:



1. Setup express server:

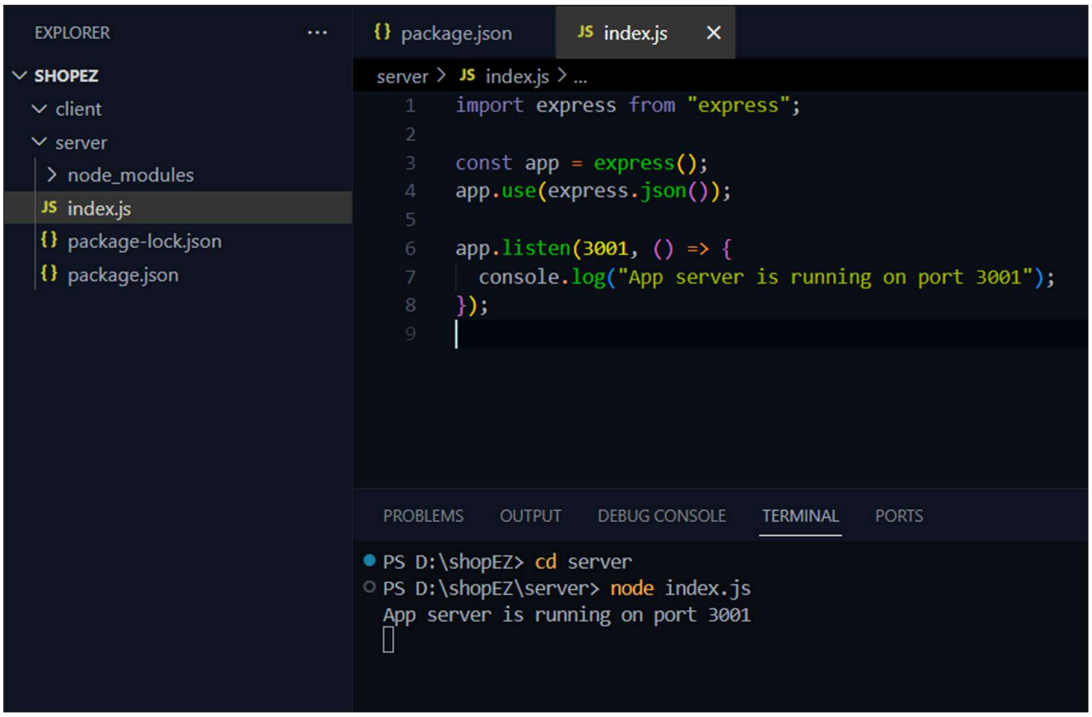
• Create index.js file.

• Create an express server on your desired port number.

• Define API's

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

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.

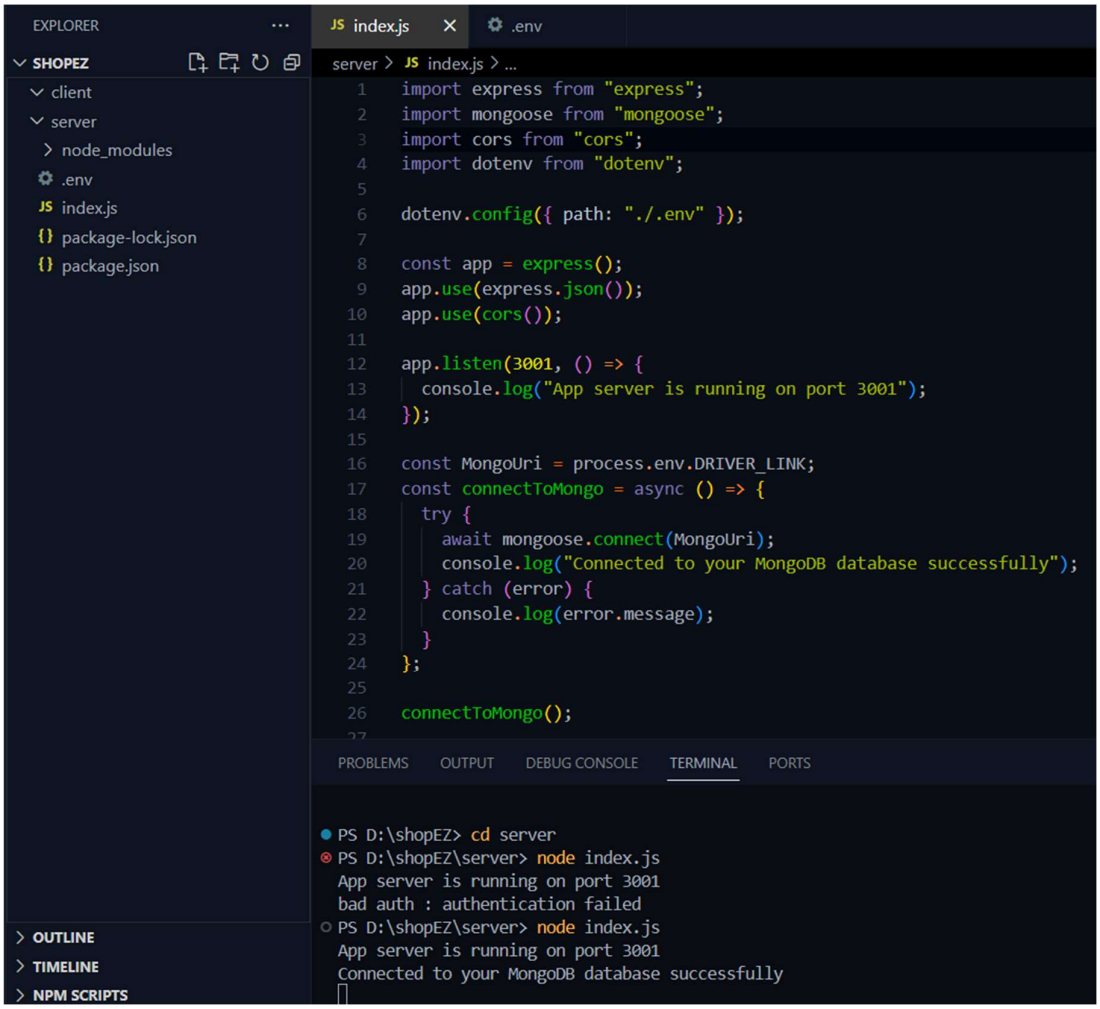
• Create a database and define the necessary collections for admin, users, restaurants, food products, orders,and other relevant data.

Reference 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:



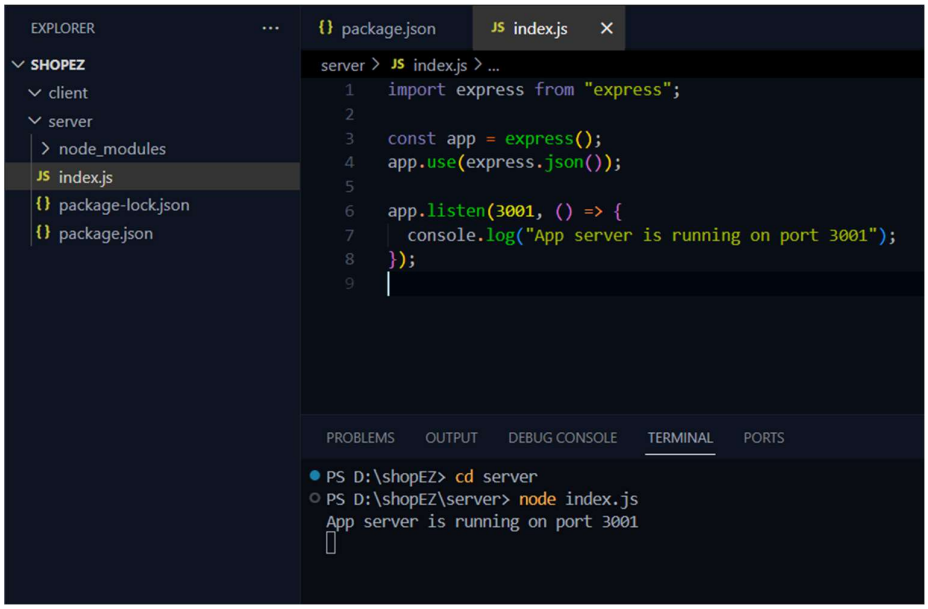
3. Create Express.js Server:

• 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/1-uKMIcrok_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.

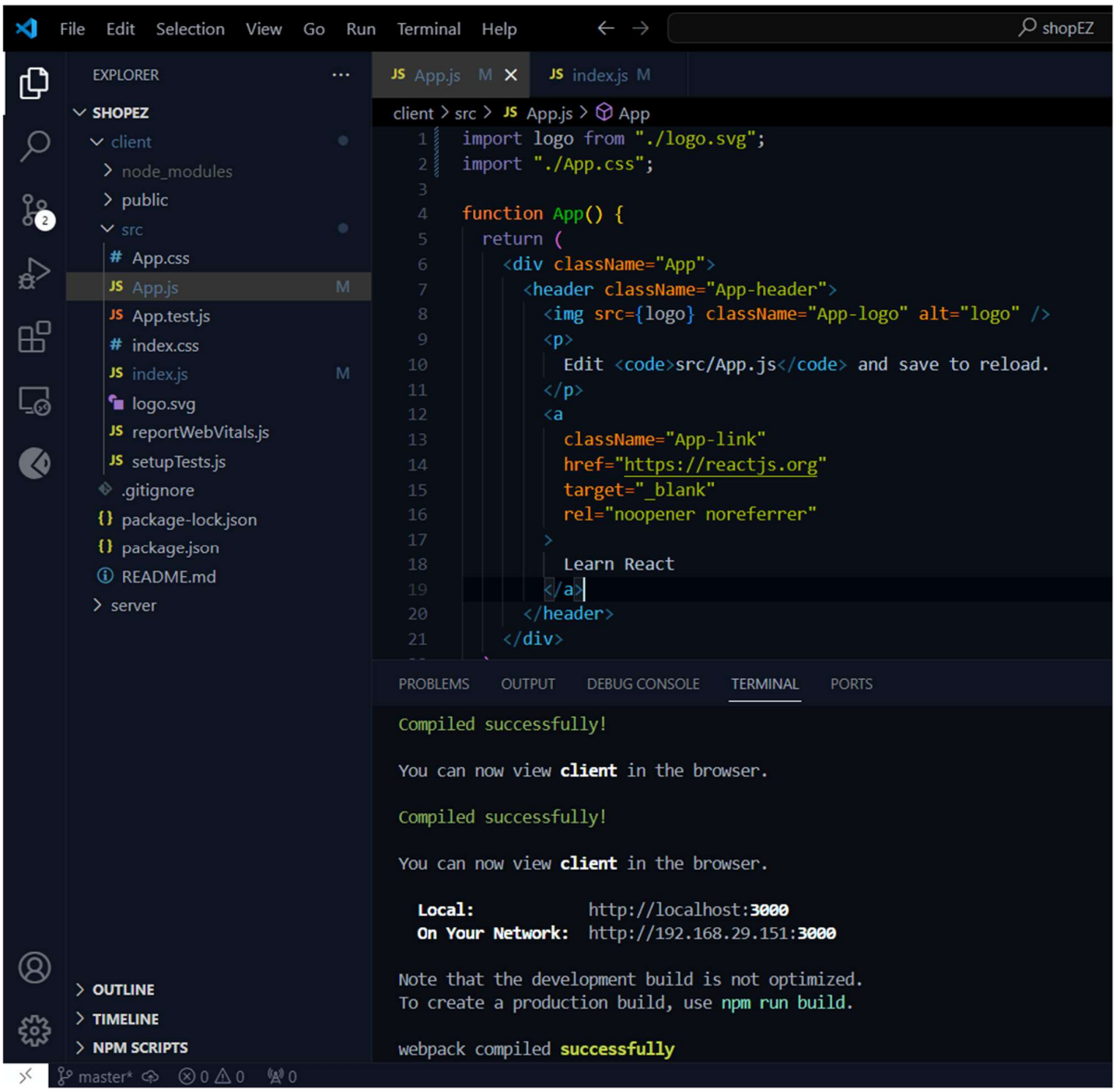
FRONTEND DEVELOPMENT:

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.

index.js

server > index.js > ...

```
import express from 'express'
import bodyParser from 'body-parser';
import mongoose from 'mongoose';
import cors from 'cors';
import bcrypt from 'bcrypt';
import {Admin, Cart, FoodItem, Orders, Restaurant, User } from './Schema.js'


const app = express();

app.use(express.json());
app.use(bodyParser.json({limit: "30mb", extended: true}))
app.use(bodyParser.urlencoded({limit: "30mb", extended: true}));
app.use(cors());

const PORT = 6001;

mongoose.connect('mongodb://localhost:27017/foodDelivery',{
    useNewUrlParser: true,
    useUnifiedTopology: true
}).then(()=>{
```

User Authentication:

- **Backend**

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

index.js

server > index.js > then() callback

```
app.post('/login', async (req, res) => {
    const { email, password } = req.body;
    try {
        const user = await User.findOne({ email });

        if (!user) {
            return res.status(401).json({ message: 'Invalid email or password' });
        }
        const isMatch = await bcrypt.compare(password, user.password);
        if (!isMatch) {
            return res.status(401).json({ message: 'Invalid email or password' });
        } else{
            return res.json(user);
        }
    } catch (error) {
      console.log(error);
      return res.status(500).json({ message: 'Server Error' });
    }
});
```

index.js

server > index.js > then() callback > app.post('/login') callback

```
app.post('/register', async (req, res) => {
    const { username, email, usertype, password , restaurantAddress, restaurantImage} = req.body;
    try {
        const existingUser = await User.findOne({ email });
        if (existingUser) {
            return res.status(400).json({ message: 'User already exists' });
        }
        const hashedPassword = await bcrypt.hash(password, 10);
        if(usertype === 'restaurant'){
            const newUser = new User({
                username, email, usertype, password: hashedPassword, approval: 'pending'
            });
            const user =  await newUser.save();
            console.log(user._id);
            const restaurant = new Restaurant({ownerId: user._id ,title: username,
                        address: restaurantAddress, mainImg: restaurantImage, menu: []});
            await restaurant.save();
            return res.status(201).json(user);
        } else{
            const newUser = new User({
                username, email, usertype, password: hashedPassword, approval: 'approved'
            });
            const userCreated = await newUser.save();
            return res.status(201).json(userCreated);
        }
    } catch (error) {
      console.log(error);
      return res.status(500).json({ message: 'Server Error' });
    }
});
```

- **Frontend**

  Login:

GeneralContext.js U

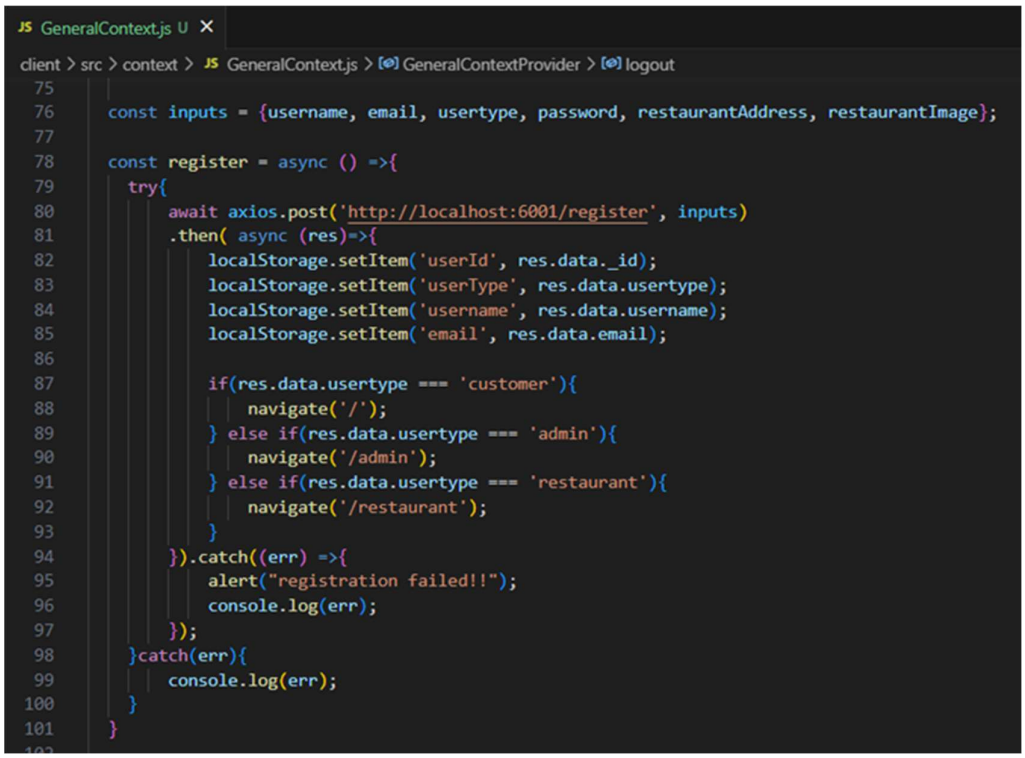
client > src > context > GeneralContext.js > GeneralContextProvider > register > then() callback

```
const login = async () =>{
  try{
    const loginInputs = {email, password}
      await axios.post('http://localhost:6001/login', loginInputs)
      .then( async (res)=>{

        localStorage.setItem('userId', res.data._id);
          localStorage.setItem('userType', res.data.usertype);
          localStorage.setItem('username', res.data.username);
          localStorage.setItem('email', res.data.email);
          if(res.data.usertype === 'customer'){
              navigate('/');
          } else if(res.data.usertype === 'admin'){
              navigate('/admin');
          }
        }).catch((err) =>{
          alert("login failed!!");
          console.log(err);
        });
      }catch(err){
        console.log(err);
      }
    }
```

Logout:

Register:

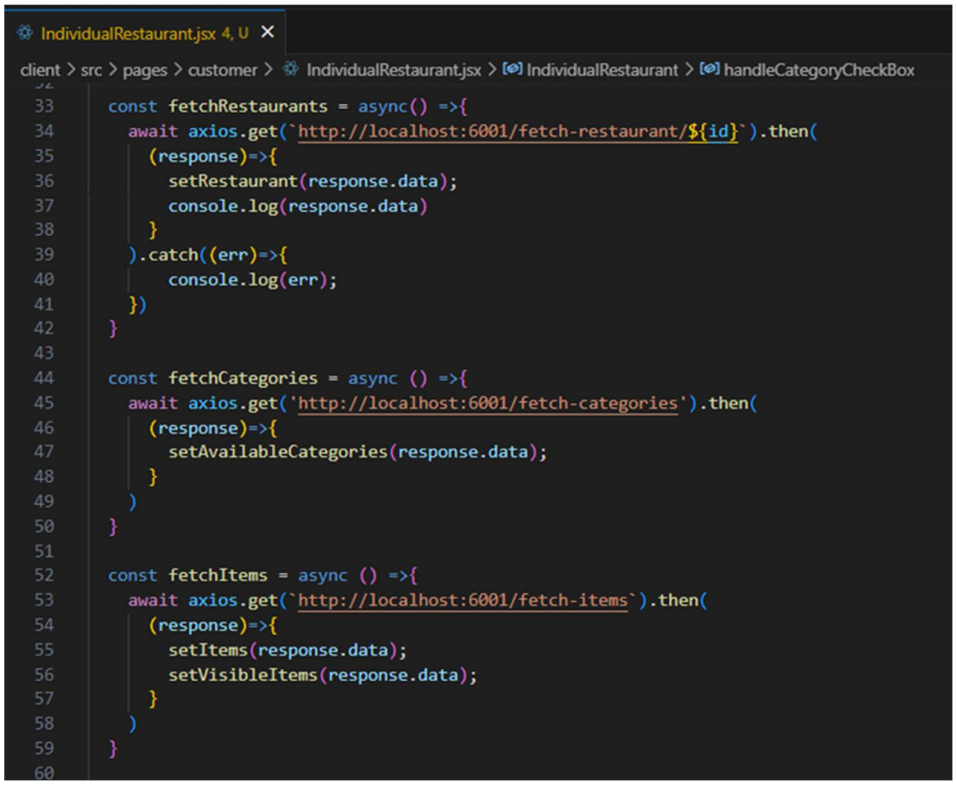


All Products (User):

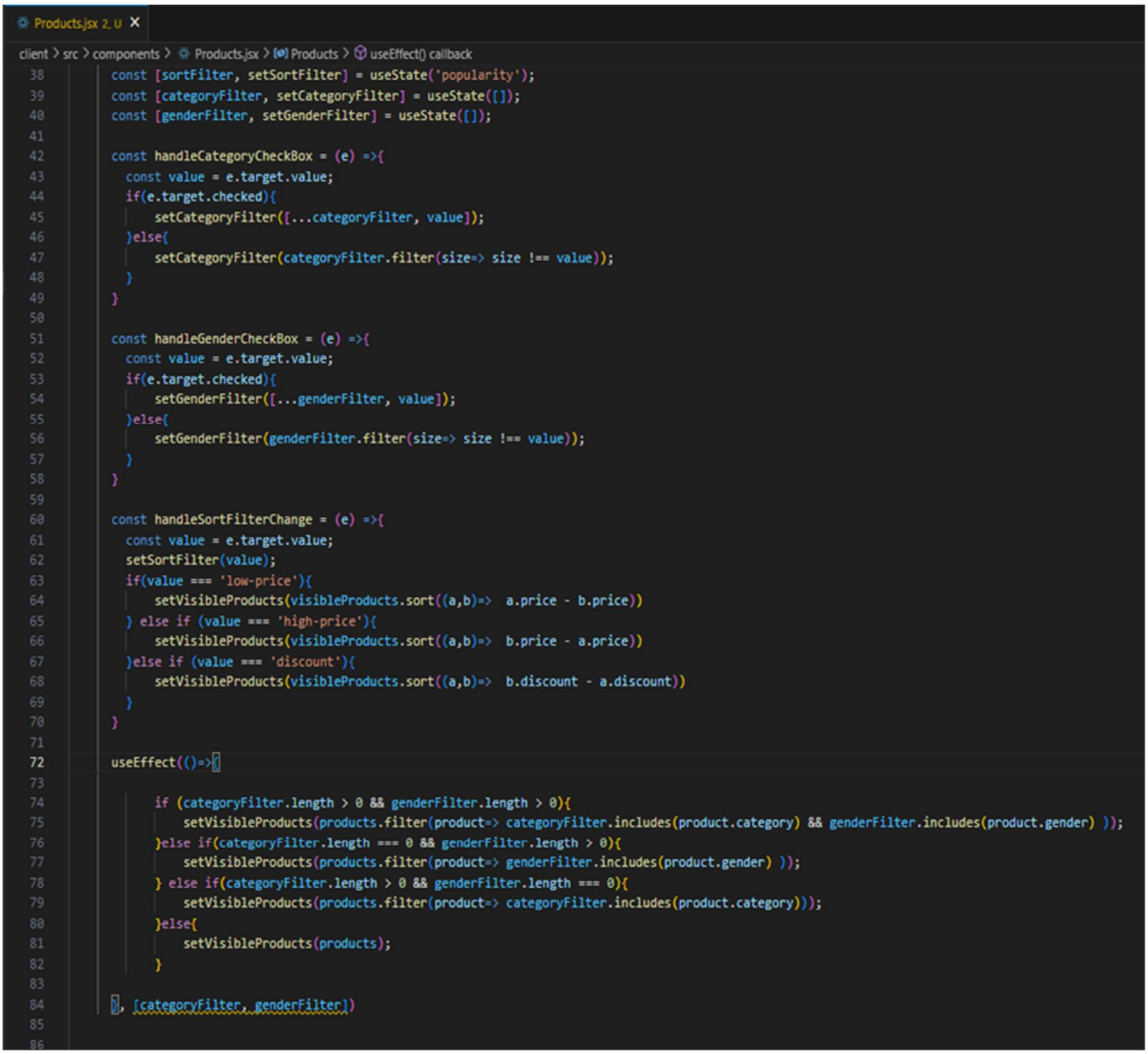
· Frontend

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

Fetching food items:



Filtering products:



- **Backend**

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

index.js

server > 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
108         }catch(err){
109             res.status(500).json({ message: 'Error occured' });
110         }
111     })
```

Add product to cart:

- **Frontend**

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

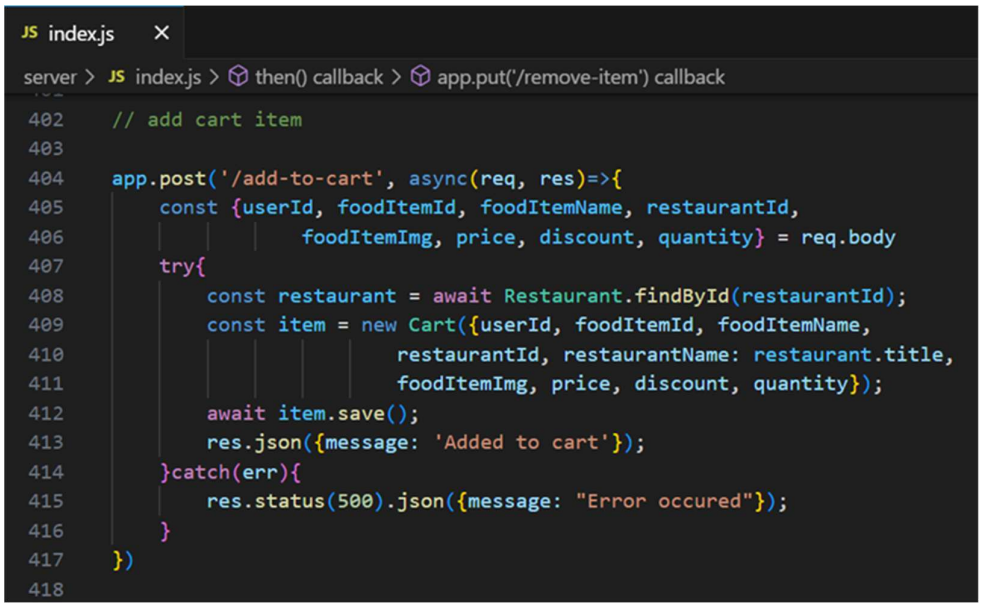
IndividualRestaurant.jsx U

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**

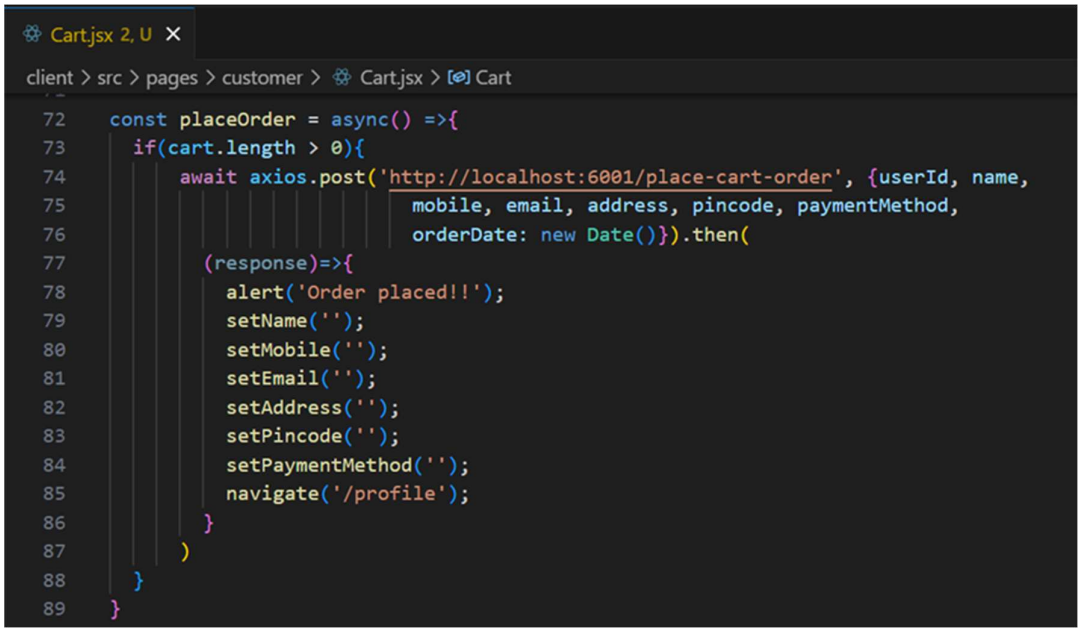
Add product to cart:



Order products:

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

- Frontend



- **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.

index.js

server > 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   })
```

Add new product:

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

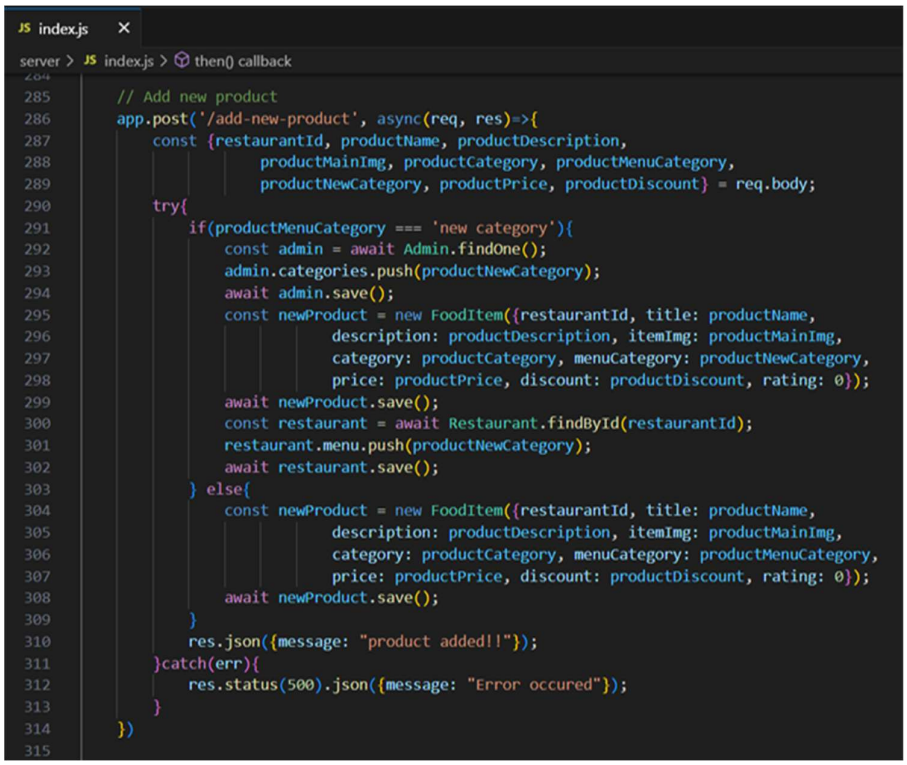
- Frontend:

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
```

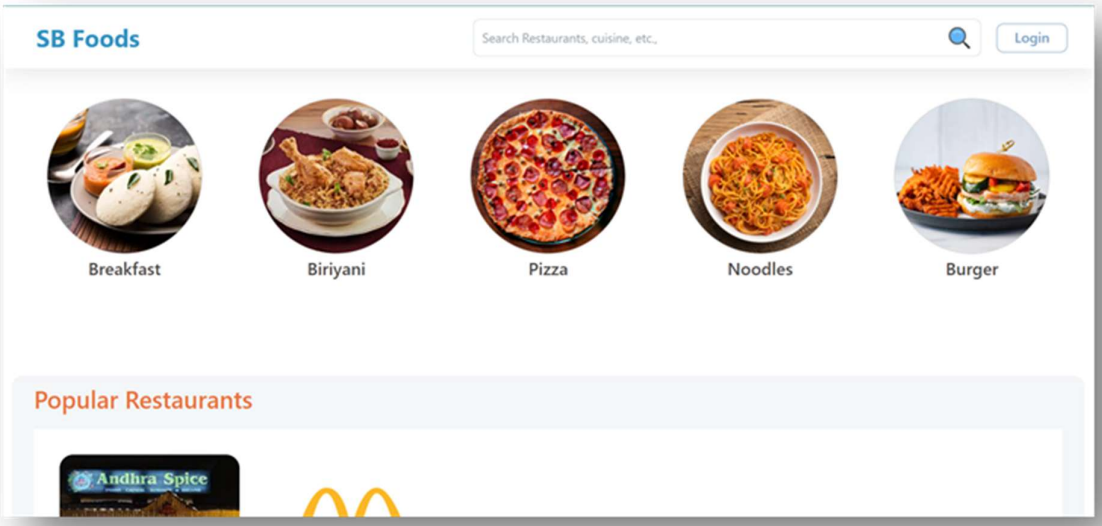
- **Backend:**



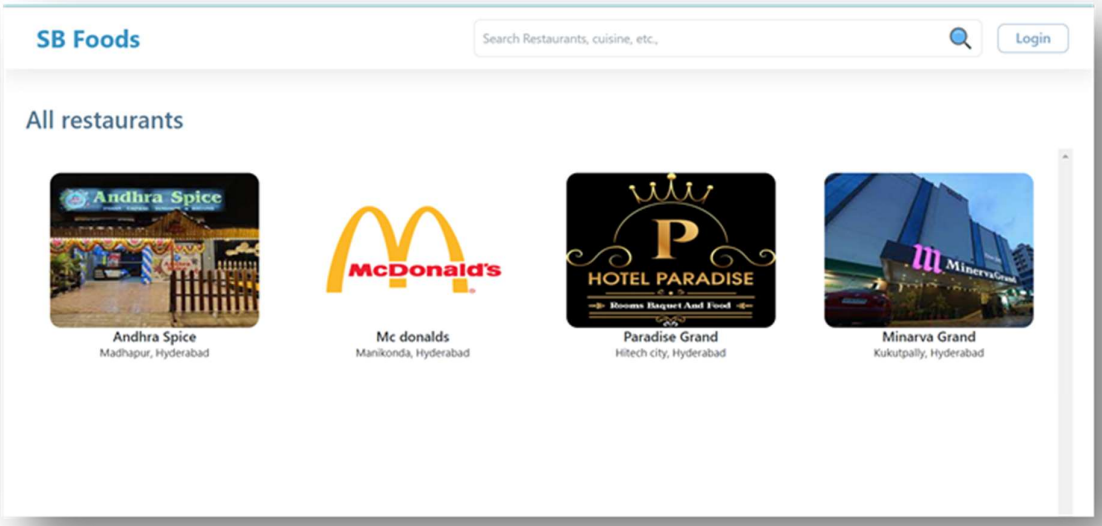
Along with this, implement additional features to view all orders, products, etc., in the admin dashboard.

Demo UI images:

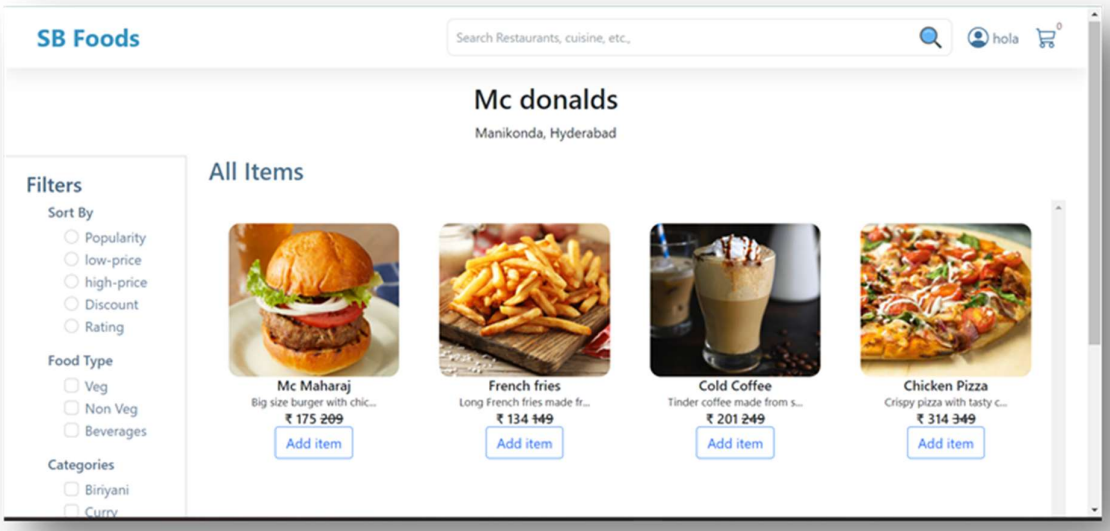
- **Landing page**



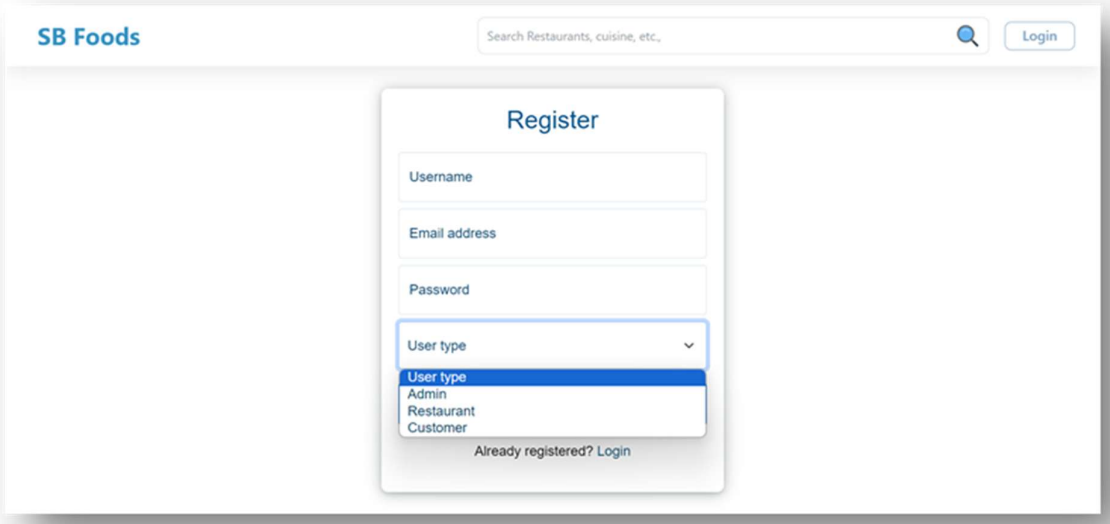
- **Restaurants**



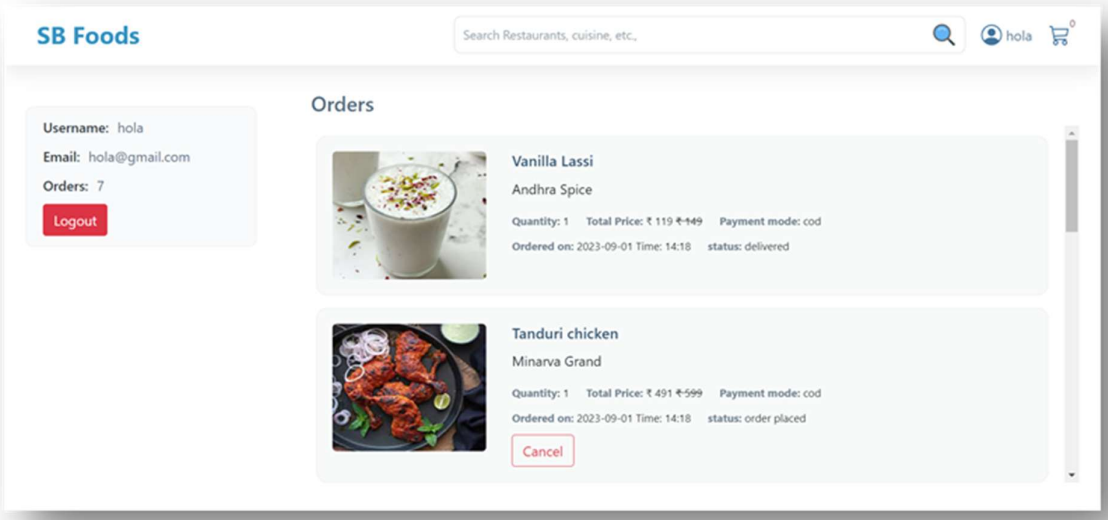
- **Restaurant Menu**



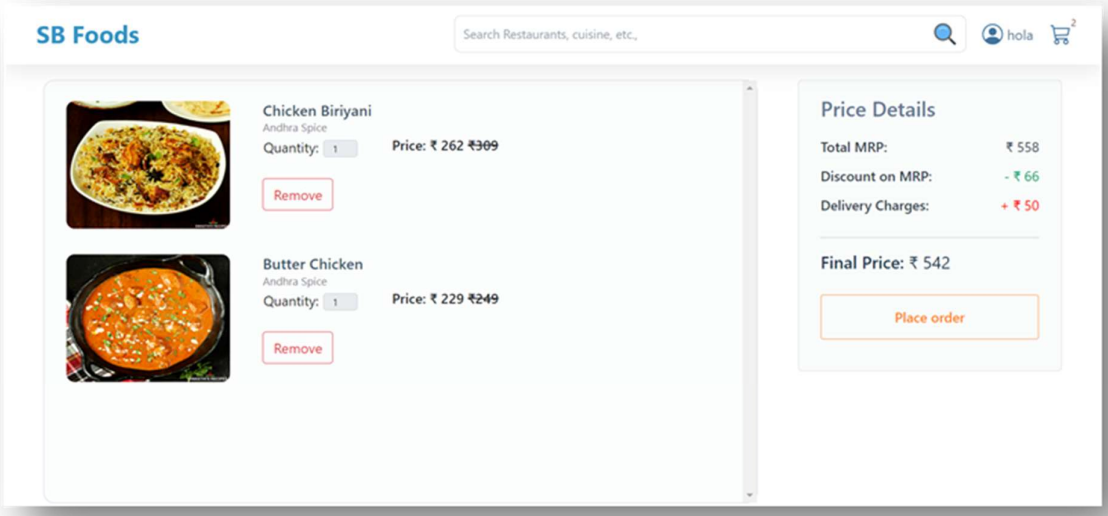
- **Authentication**



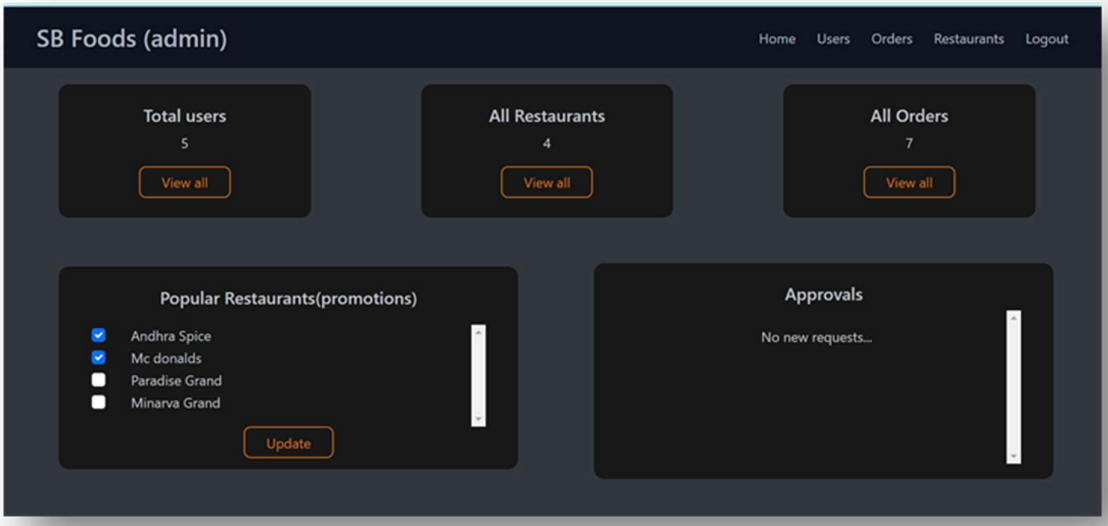
- **User Profile**



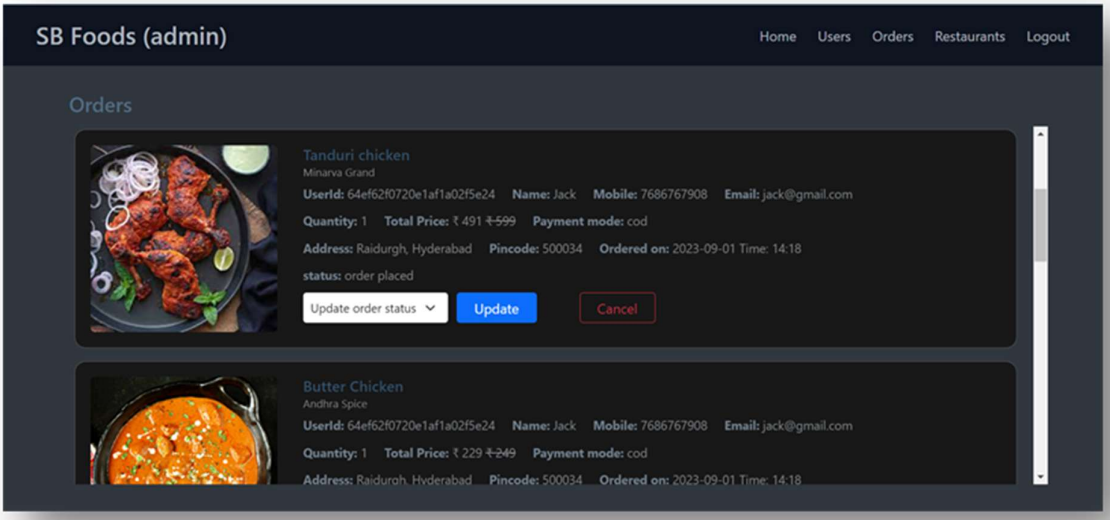
- **Cart**



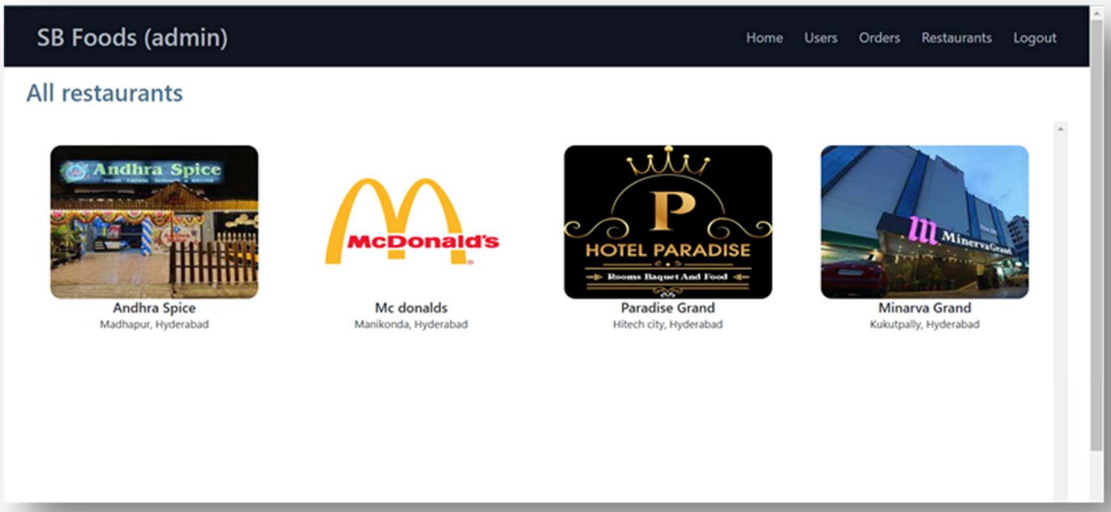
- **Admin dashboard**



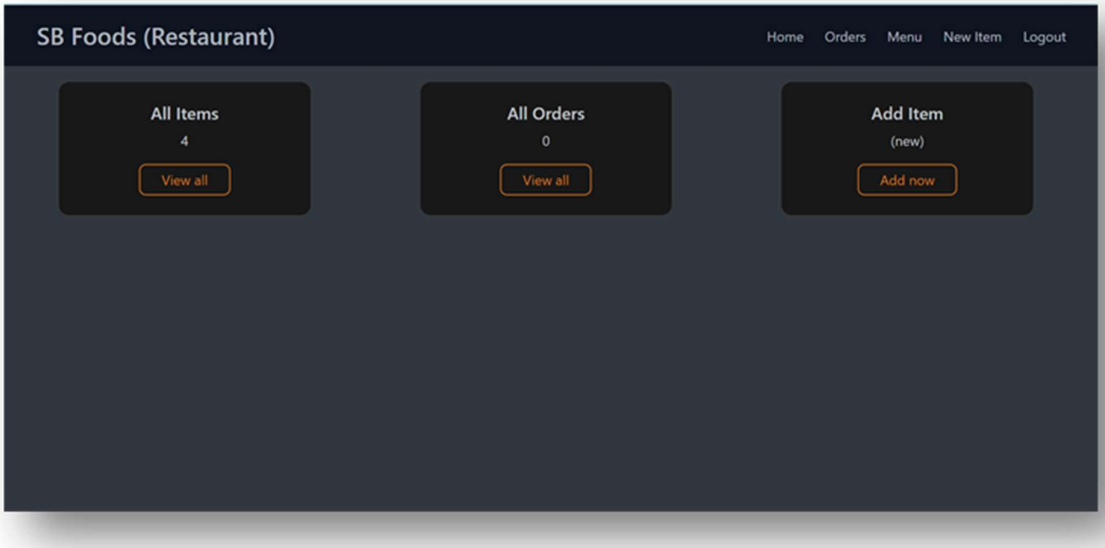
- **All Orders**



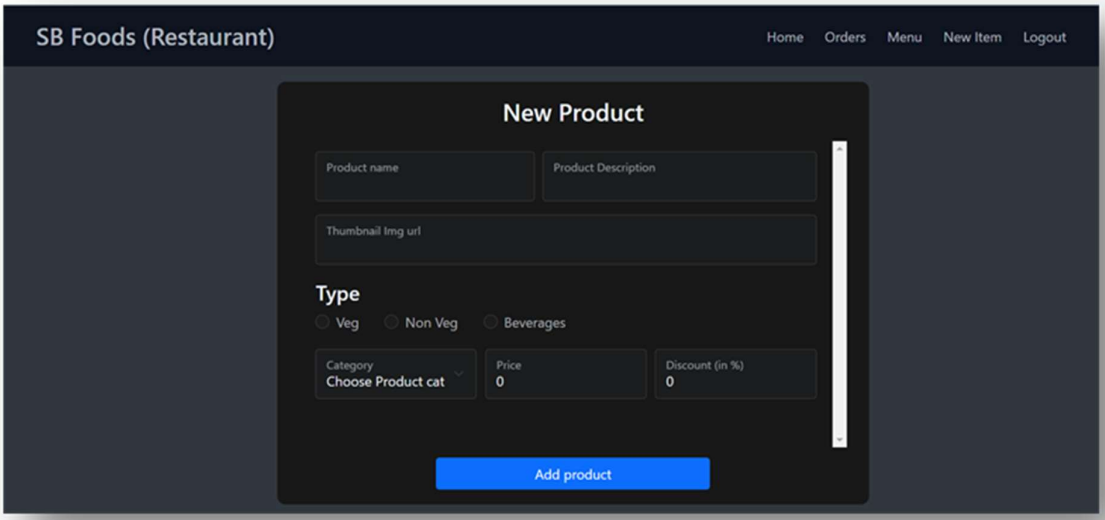
- **All restaurants**



- **Restaurant Dashboard**



- **New Item**



For any further doubts or help, please consider the GitHub repo, https://github.com/subbu-vishnu-86/orderonthego-your-on-demand-food-ordering-solution.git

The demo of the app is available at: https://drive.google.com/drive/folders/1NYmJvWeBi5gpn0oTtMzSCFOrFJOYe-ZZ

** Happy Coding **