PROJECT REPORT

OrderOnTheGo:

Your On-Demand Food Ordering Solution

Submitted by

Team ID : LTVIP2025TMID55809

TEAM LEADER : SRIPRIYA AKULA

TEAM MEAMBER : POOJITHA PASUPULETI

TEAM MEAMBER : SARVA SREE LAKSHMI MANASWINI

TEAM MEAMBER : SUNDAR SINGH PARASA

Department of Computer Science and Engineering DMS SVH College of Engineering Machilipatnam, Krishna District, Andhra Pradesh – 521 002, India

Affiliated to: Jawaharlal Nehru Technological University, Kakinada (JNTU-K)

Submitted to

Academic Year

2022 - 2026

ABSTRACT

The Online Food Ordering System is designed to streamline and simplify the process of ordering food for both customers and restaurant staff. This system provides a user-friendly interface that displays an up-to-date menu with all available items, allowing customers to conveniently select multiple food items to place an order. Before completing the transaction, customers can review their order details, and upon confirmation, the system notifies them and records the order in real-time.

The order is then added to a processing queue and stored in the database, enabling restaurant staff to monitor and fulfill orders efficiently. This real-time synchronization enhances the workflow within the restaurant, improving order accuracy and service speed.

Primarily developed for the food delivery industry, the system supports restaurants and hotels in boosting their online food ordering capabilities. With just a few clicks, customers can choose their meals, and the system ensures timely and accurate delivery to their specified location. Restaurant staff benefit from an intuitive, graphical interface that simplifies order tracking and management, ultimately improving customer satisfaction and operational efficiency.

INTRODUCTION

The Online Food Ordering System can be defined as a simple and convenient way for customers to purchase food online without physically visiting a restaurant. This system is powered by the internet, which connects restaurants or food service providers directly with customers. Through this system, customers can access the restaurant’s website, browse available food items, and select and purchase the items of their choice. The selected food is then delivered to the customer’s location at their preferred time by a delivery person.

Payments for online orders can be made using various methods, including debit or credit cards, cash on delivery, or digital wallets. This system offers a safe and secure platform for food ordering and is rapidly transforming the operations of the food service industry.

This project proposes an Online Food Ordering System designed specifically for fast food restaurants, take-out services, or college cafeterias. However, it can be adapted for any segment of the food delivery industry. The system automates the entire order-taking process, reducing manual effort and improving efficiency for both customers and restaurant staff.

One of the key advantages of this project is its ability to significantly streamline the ordering process. When a customer visits the ordering webpage, they are presented with an interactive, real-time menu that displays all available food options, including dynamic pricing based on selected items. Once selections are made, items are added to the order, and the customer can view and modify their order at any time before final checkout. This provides instant visual confirmation and ensures a smooth, user-friendly experience.

Motivation

The motivation for designing this application came from my family's involvement in the fast food business.

Personally, I do not like waiting in long queues at the store or having to call to place an order, especially during peak lunch or dinner hours.

Additionally, I have recently gained interest and experience in Express.js, and I truly appreciate how powerful and efficient it is for building modern web applications.

For the back-end, we chose MongoDB instead of traditional relational databases, as I found it to be extremely flexible and efficient when working with modern web technologies.

This system is specifically designed to address these challenges and provide a seamless food ordering experience.

1. It reduce manual work.

2. The online food delivery system is overcome the problems in manual system.

3. This system is fully computerized.

4. It is user friendly.

5. Provide quickly reports.

6. Highly efficient and accurate.

7. Prevention of unauthorised access of data.

8. Automated.

Problem statements

1. Some of the major problems reported by the customers surveyed revolved around late
deliveries because of network problem

2. incorrect orders being delivered due to communication problem.

3. orders not being delivered at all, rude customer service.

4. , cold food being delivered, and the driver requiring a lot of guidance to find the delivery
location.

5. Sometime payment issue is occurred.

6. Online food ordering system service now days increase your budget.

7. lack of a visual confirmation that the order was placed correctly.

Purpose or objectives and goals

The proposed system is developed to manage ordering activities in fast food restaurant. It helps
to record customer submitted orders. The system should cover the following functions in order to
support the restaurant's business process for achieving the objectives:

1. To allow the customer to make order, view order and make changes before submitting their order and allow them make payment through prepayment card or credit card or debit card.
2. To provide interface that allows promotion and menu.
3. To prevent interface that shows customers' orders detail to front-end and kitchen staffs for delivering customers' orders
4. Tools that generate reports that can be used for decision making
5. A tool that allows the management to modify the food information such as price, add a new menu and many others as well as tools for managing user, system menu and promotion records.

This will minimize the number of employees at the back of the counter.
The system will help to reduce labor cost involved.
The system will be less probable to make mistake, since it's a machine.
This will avoid long queues at the counter due to the speed of execution and number of optimum
screens to accommodate the maximum throughput.

The main objective of the Online Food Ordering System is to manage the details of Item Category,Food,Delivery Address,Order,Shopping Cart. ... The purpose of the project is to build an application program to reduce the manual work for managing the Item Category, Food, Customer, Delivery Address

Objectives and Goals

1. To increase efficiency and improve services provided to the customers through better application of technology in daily operations.
2. To be able to stand out from competitors in the food service industry
3. To enable customers to order custom meals that aren't in the menu
4. To enable customers to have a visual confirmation that the order was placed correctly
5. To enable customers to know food ingredients before ordering
6. To reduce restaurant's food wastage
7. To ensure correct placement of orders through visual confirmation
8. Improve efficiency of restaurant's staff
9. Eliminate paper work and increase level of accuracy
10. Increase speed of service, sales volume and customer satisfaction
11. To increase efficiency by shortening the purchasing time and eliminating paper work like receipts through online transaction
12. To be able to stand out from competitors by automating daily operations whichwill give food service providers the opportunity to increase sales
13. To reduce restaurants food wastage and increasing efficiency of the restaurants staff by enabling the restaurants staff to know what food items the customers wantin advance.
14. To increase customer satisfaction by speeding up food delivery
15. To reduce time wasting by eliminating long queues
16. More accuracy and easy order processing.
17. 24

Literature Survey

Various case studies have highlighted the problems faced While setting up a restaurant.. Some of the problems Found during the survey in the existing system are listed Below :

1. To place the orders customer visits the restaurant, Checks the menu items available in the restaurant, and chooses the items required, then places the order And then do the payment.
2. This method demands Manual work and time on the part of the customer.
3. When the customer wants to order over the phone, Customer is unable to see the physical copy of the Menu available in the restaurant, this also lacks the Verification that the order was placed for the appropriate menu items.
4. Every restaurant needs someone or the other to take order personally or over phone, to offer the Customer a rich experience and even to process the payment.

Project scope and limitation

Note-Refer old document format that are already sent you.

1. This system will help to customer and restaurants and administrator for the ordering process.
2. Easy to make ordering and hopefully can smoothen up the job of administrator and waiter.
3. This system produce a computerized system in defining the best solution in food delivary system.
4. Easy acces to any stage.
5. Lot of time is save.
6. Easy back up of data.

Limitations

1. Cost associated with backp storage to the system than the cost associate with maintaining on-site alone.
2. A potential for customer to fail to adapt to online ordering or tablesite checkout.

Project Perspective

The Online Food Ordering System (SB Foods) is a comprehensive, web-based application that allows users to conveniently browse menus, place food orders, and track deliveries. This system is designed for access via internet browsers on various devices like PCs, laptops, and smartphones. It ensures a smooth, secure, and efficient ordering experience for customers, while also offering powerful tools for restaurant staff and administrators.

System Model

The Online Food Ordering System (SB Foods) is a comprehensive, web-based application that allows users to conveniently browse menus, place food orders, and track deliveries. This system is designed for access via internet browsers on various devices like PCs, laptops, and smartphones. It ensures a smooth, secure, and efficient ordering experience for customers, while also offering powerful tools for restaurant staff and administrators.

Smart Internz

The system is structured into three major logical components:

1. **Web Ordering System**:
   Facilitates customers in placing orders and entering their delivery and payment details.
2. **Menu Management**:
   Allows the restaurant admin to manage menu items, including additions, updates, and deletions.
3. **Order Retrieval System**:
   Empowers the restaurant to process, monitor, and manage incoming orders in real-time.

PRODUCT FUNCTIONS

1. **Web Ordering System Module :**
This module enables users to place orders and enter key details such as location, contact, and payment preferences. Key submodules include:

- Home Page
- Meal Plan Page
- My Cart Page
- Login/Signup Page

2. Menu Management Module :
Used by admins to manage food item listings, which are visible to users:

- Add/Edit/Delete Food Items
- Set Food Size Options
- Configure Pricing
- Upload Food Images
- Add Descriptions

3. Order Retrieval Module :
Allows restaurant staff to handle customer orders:

- View Order Plans
- Track Order Quantity
- Manage Delivery Status

Smart Internz

User Scenario: Late-Night Craving Resolution

Meet Lisa, a college student working late on an assignment. It's midnight, and she realizes she skipped dinner. Cooking is not an option, and going out isn't safe or convenient.

Using SSP Foods, Lisa:

1. Opens the app and goes to the "Late-Night Delivery" section.
2. Browses menus and finds her favorite diner still accepting orders.
3. Chooses chicken noodle soup and garlic bread.
4. Adds items to her cart, selects delivery address, and payment method.
5. Reviews order details and confirms with a tap.
6. Gets instant confirmation and estimated delivery time.
7. Receives her hot, delicious meal quickly and gets back to work.

This showcases how SSP Foods addresses user needs seamlessly even at odd hours—providing convenience, speed, and satisfaction without compromise.

Technical Architecture Overview

```
+---------------------------+     +-----------------------------+      +---------------------------+
|    Frontend               |     |     Backend                 |      |     Database              |
|---------------------------|     |-----------------------------|      |---------------------------|
| - User Interface          |<--->| - RESTful APIs              |<-->  | - Users Collection        |
| - User Auth               |     | - Admin Auth                |      | - Orders Collection       |
| - Product Pages           |     | - Admin Dashboard           |      | - Products Menu           |
| - Cart Management         |     | - Order Processing          |      | - Cart Items              |
| - User Profile            |     |                             |      | - Admin Records           |
+---------------------------+     +-----------------------------+      +---------------------------+
```

Key Components:

- **Frontend**: Built with responsive UI for user interaction, login, cart, and admin control.
- **Backend**: Node.js/Express or similar stack serving APIs for users, products, orders, and admin.
- **Database**: MongoDB or SQL storing persistent data for orders, users, and menu items.

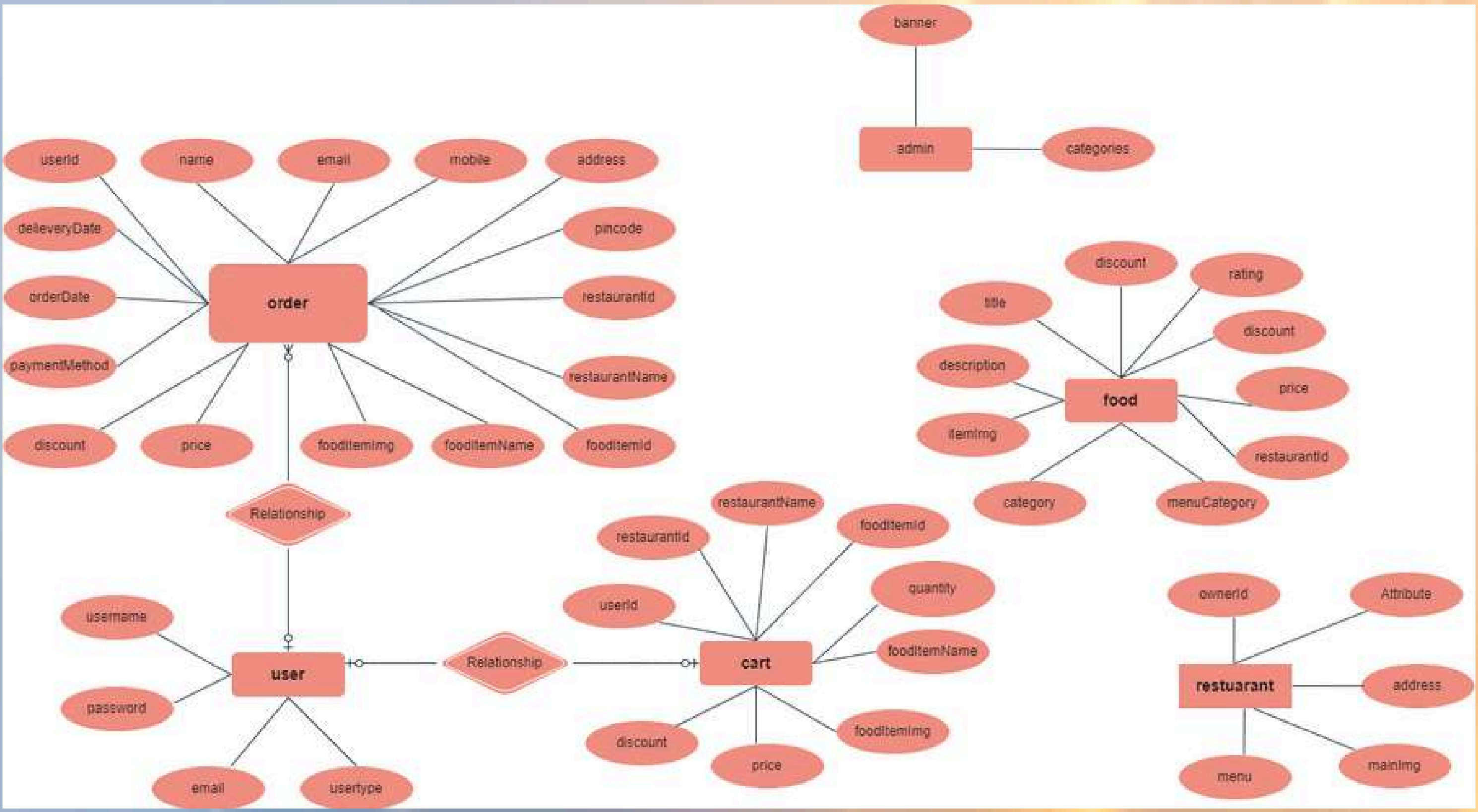
Smart Internz

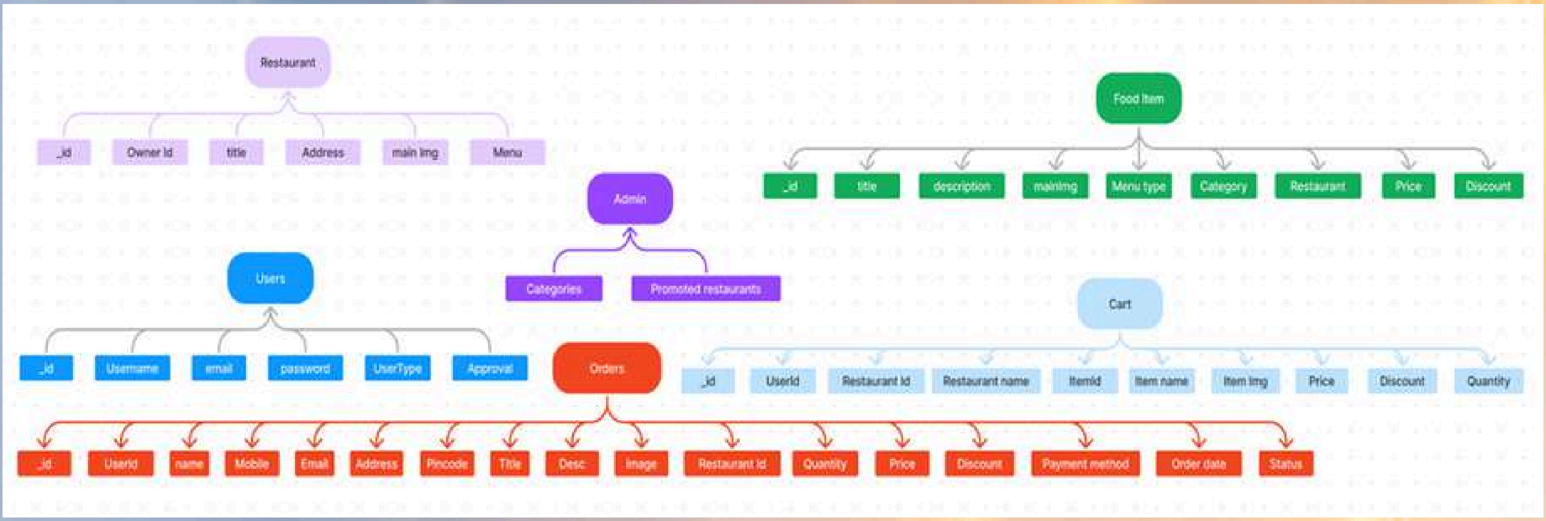
ER-Diagram Overview

System Design :

Design Constraints-Entity Relationship Diagram

The SSP Foods ER-Diagram visualizes the key entities and their relationships within the food ordering e-commerce platform. It outlines how users, restaurants, products, carts, and orders are structured and interact with each other in the system.





Entity Descriptions

- **User:**

Represents individuals registered on the platform. Each user has a unique ID and personal information like name, email, and address.

- **Restaurant**:

Stores details of restaurants listed on the platform. Includes restaurant name, contact, cuisine type, and availability.

- **Admin**:

Represents platform administrators who manage promoted restaurants, food categories, and have access to system-level controls.

- **Products:**

Contains all food items available for order. Each product is associated with a restaurant and includes name, description, price, size, image, and category.

- **Cart:**

Stores items added by users before placing an order. Each cart entry links products with the corresponding user ID and quantity.

- **Orders:**

Maintains records of all user purchases. Each order is linked to a user and contains order details like items, quantity, status, delivery address, and payment method.

Key Features of SSP Foods

1. Comprehensive Product Catalog

SSP Foods offers a wide variety of dishes from multiple restaurants. Users can explore food items through an organized catalog with:

- Detailed food descriptions
- Customer reviews
- Prices and discounts
- Availability by category or restaurant

Smart Internz

2. Order Details Page

After choosing their meals, users are guided to a dedicated Order Details Page where they can:

- Enter shipping/delivery address
- Select payment method (e.g., UPI, card, cash)
- Add notes or instructions (e.g., spice level, no onions)

3. Secure & Efficient Checkout

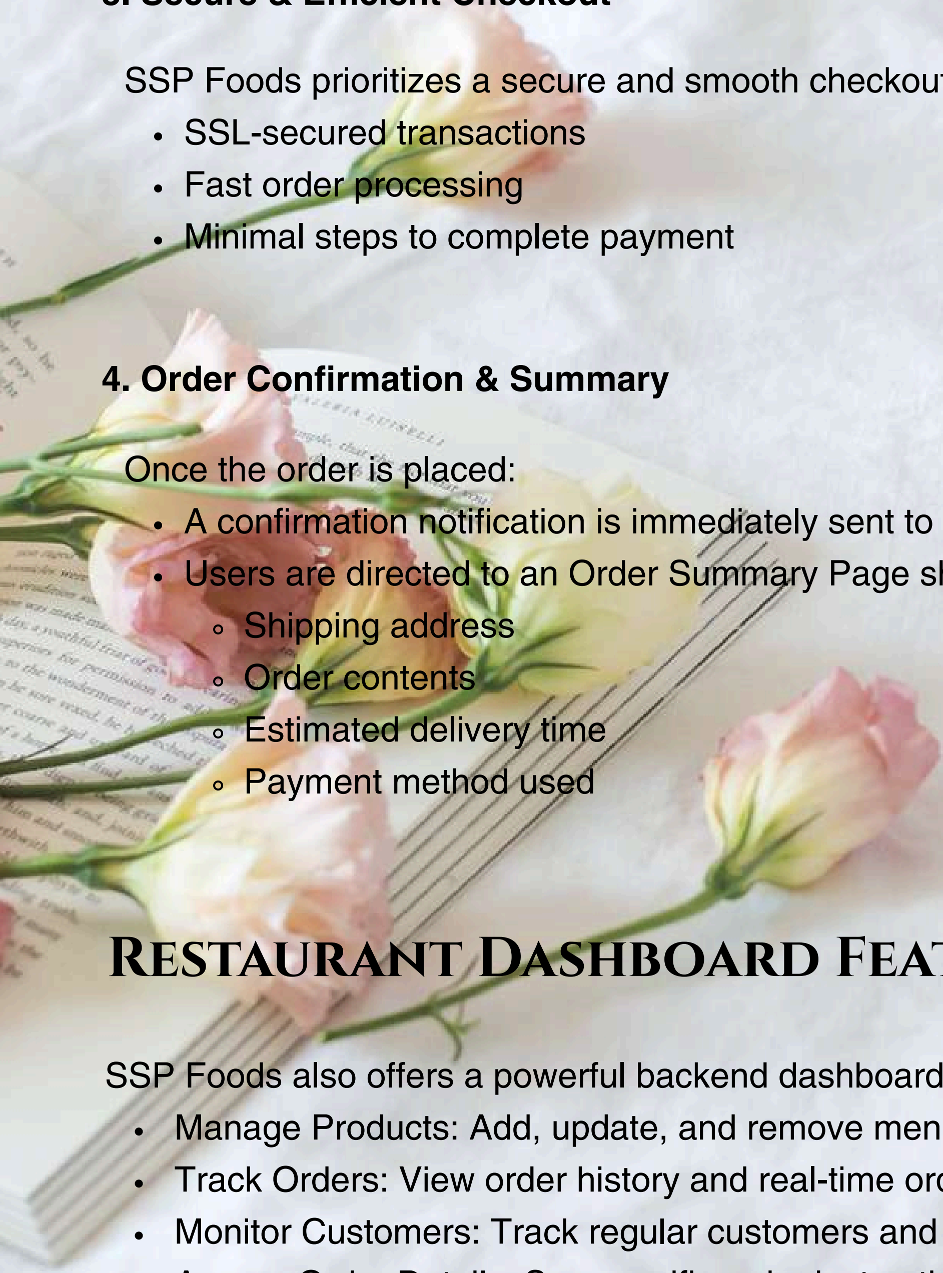
SSP Foods prioritizes a secure and smooth checkout experience:

- SSL-secured transactions
- Fast order processing
- Minimal steps to complete payment

4. Order Confirmation & Summary

Once the order is placed:

- A confirmation notification is immediately sent to the user
- Users are directed to an Order Summary Page showing:
  - Shipping address
  - Order contents
  - Estimated delivery time
  - Payment method used



Restaurant Dashboard Features

SSP Foods also offers a powerful backend dashboard for restaurants, enabling them to:

- Manage Products: Add, update, and remove menu items
- Track Orders: View order history and real-time order data
- Monitor Customers: Track regular customers and popular dishes
- Access Order Details: See specific order instructions and delivery data

Summary

SSP Foods is designed to enhance the food ordering experience for both users and restaurant partners. It combines a rich product catalog, an intuitive user interface, secure transactions, and a feature-rich admin dashboard. Whether you're placing a quick midnight snack order or managing a full restaurant menu, SSP Foods delivers efficiency and convenience at every step.

Pre-Requisites & Setup Guide

REQUIRED TOOLS & TECHNOLOGIES

1. Node.js and npm
Node.js enables JavaScript to run on the server side. npm manages project dependencies.
Download: https://nodejs.org/en/download

2. MongoDB
A NoSQL database to store user, order, and product data.
Download: https://www.mongodb.com/try/download/community

3. Express.js
A Node.js web framework for building APIs.
Install: npm install express

4. React.js
Frontend library for building dynamic user interfaces.
Guide: https://reactjs.org/docs/create-a-new-react-app.html

5. HTML, CSS, and JavaScript
Core technologies for building the web interface.

6. Mongoose
MongoDB ODM for Node.js.
Guide: https://www.section.io/engineering-education/nodejs-mongoosejs-mongodb/

7. Git
Version control system.
Download: https://git-scm.com/downloads

8. Code Editor (IDE)
Examples: -
Visual Studio Code: https://code.visualstudio.com/download
Sublime Text: https://www.sublimetext.com/download
WebStorm: https://www.jetbrains.com/webstorm/download

Project Setup Instructions

Step 1: Clone the Repository

```
git clone https://github.com/harsha-vardhan-reddy-07/Food-Ordering-App-MERN
```

Step 2: Navigate to Project Directory

```
cd Food-Ordering-App-MERN
```

Step 3: Install Dependencies

```
npm install
```

Step 4: Start the Development Server

```
npm run dev OR npm run start
```

Access the App:
Visit http://localhost:3000 in your browser.
You should see the SB Foods homepage.

Application 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

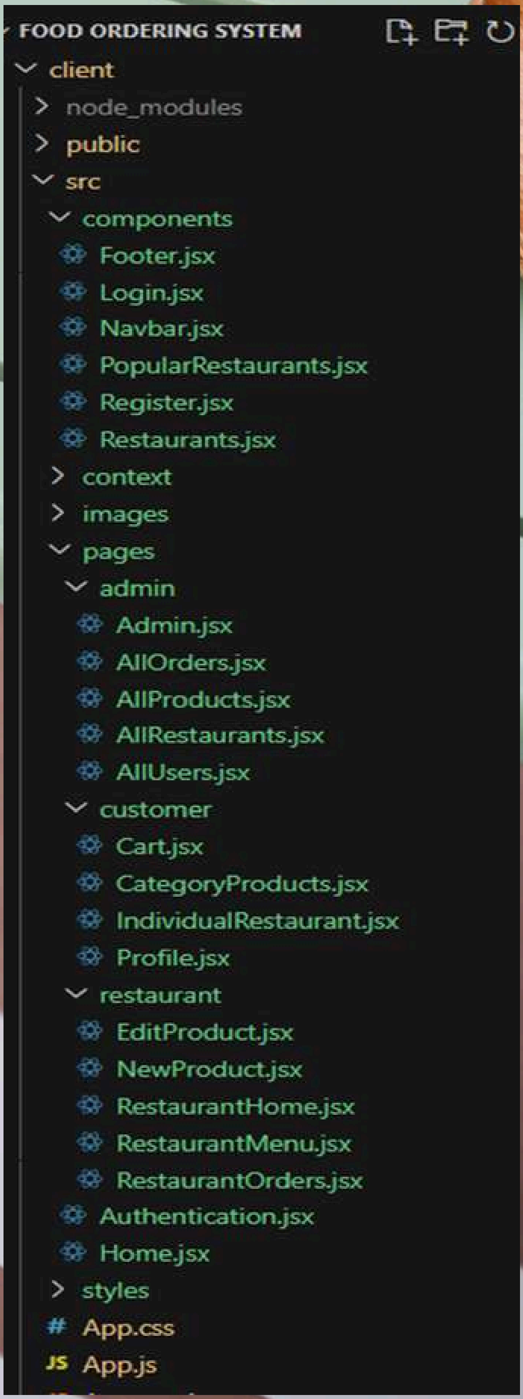
- Restaurants start by authenticating with their credentials.
- They need to get approval from the admin to start listing the products.
- They can add/edit the food items.

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.

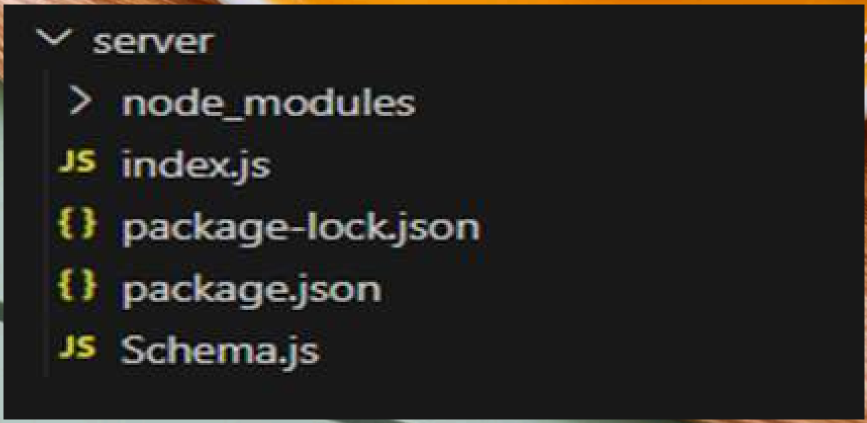
Project Structure

client structure



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

server structure



```
v server
  > node_modules
  JS index.js
  {} package-lock.json
  {} package.json
  JS Schema.js
```

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:

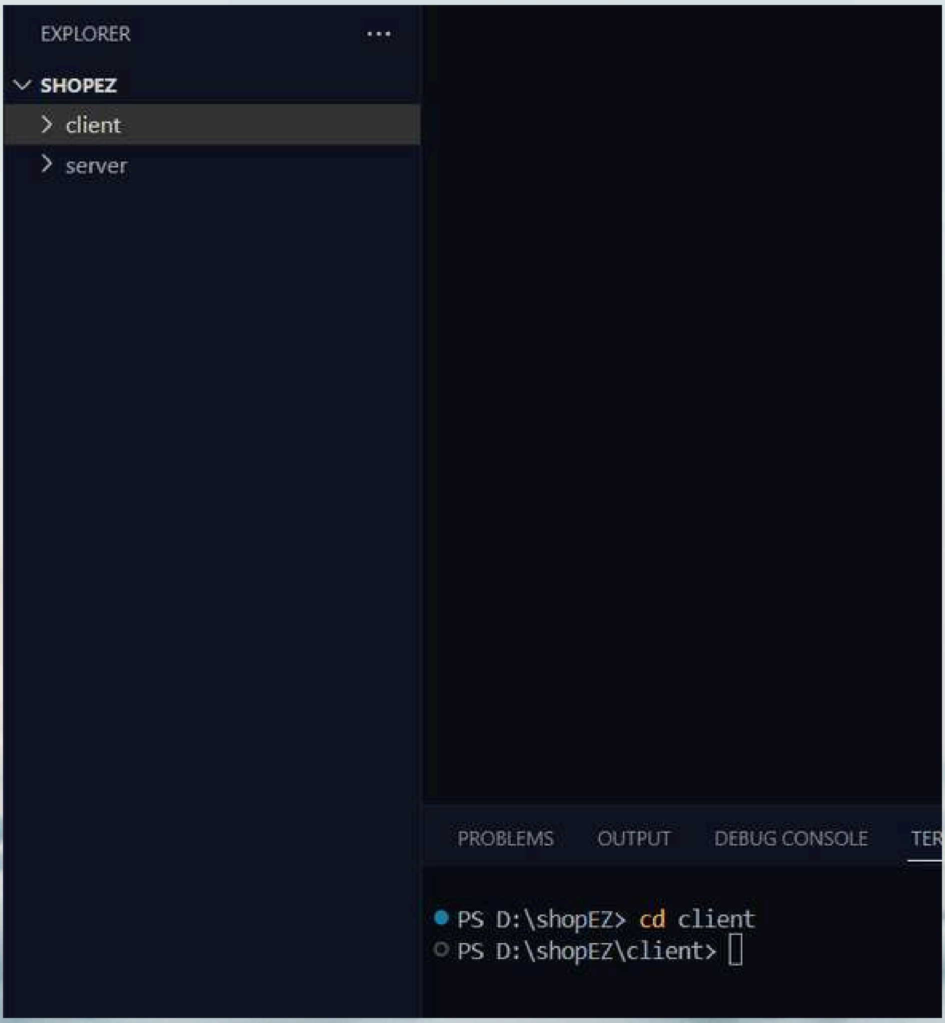
• Node.js.
Reference Article: https://www.geeksforgeeks.org/installation-of-node-js-on-windows/

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

Create project folders and files:

• Client folders.
• Server folders

Referral Image:



```
PS D:\shopEZ> cd client
PS D:\shopEZ\client>
```

This section outlines the key steps and tools required to set up the SSP Foods Full-Stack Application on your local development environment.

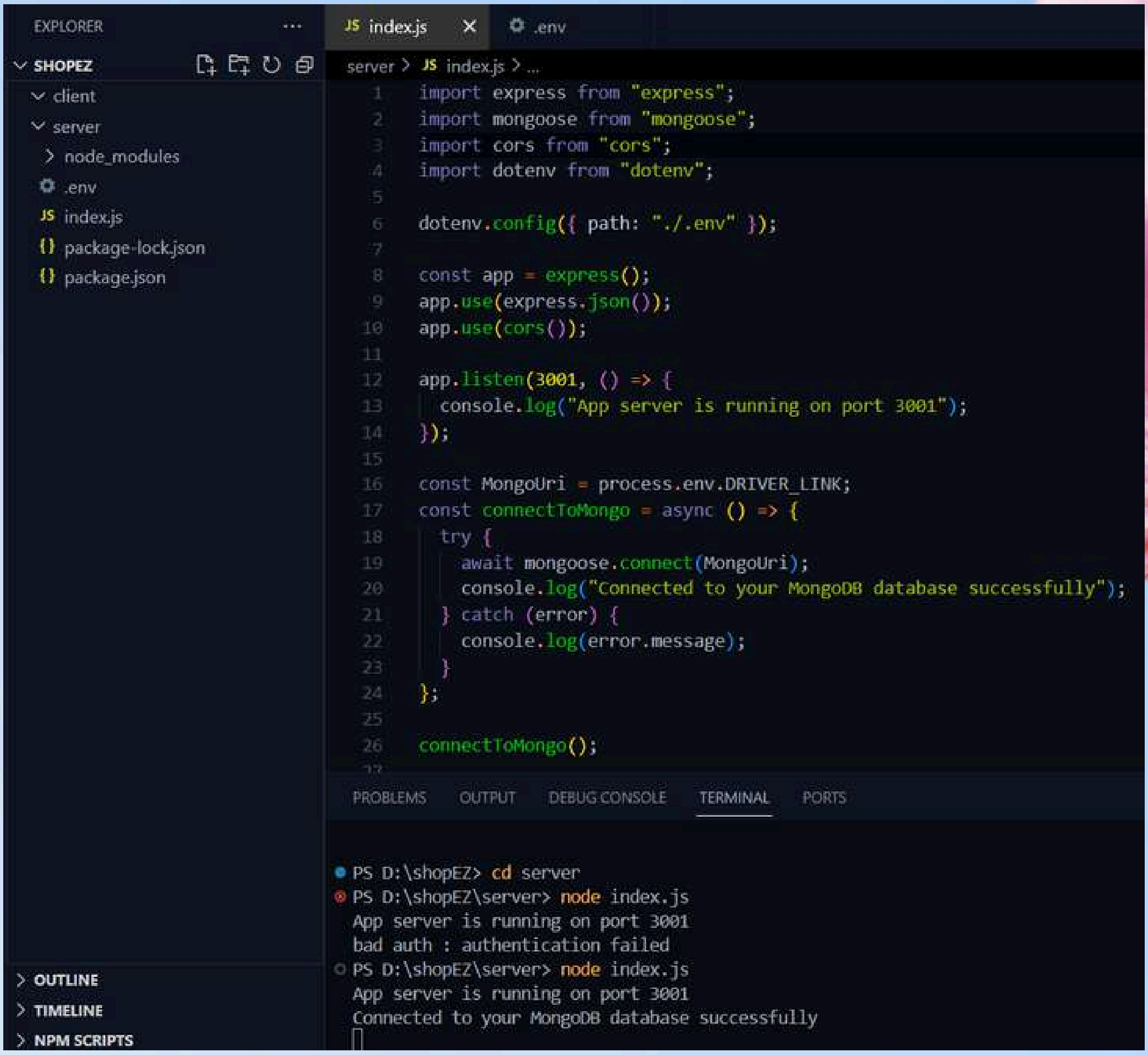
Database Development

Create database in cloud

- Install Mongoose.
- Create database connection.

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

Reference Image:



```javascript
import express from "express";
import mongoose from "mongoose";
import cors from "cors";
import dotenv from "dotenv";

dotenv.config({ path: "./.env" });

const app = express();
app.use(express.json());
app.use(cors());

app.listen(3001, () => {
  console.log("App server is running on port 3001");
});

const MongoUri = process.env.DRIVER_LINK;
const connectToMongo = async () => {
  try {
    await mongoose.connect(MongoUri);
    console.log("Connected to your MongoDB database successfully");
  } catch (error) {
    console.log(error.message);
  }
};

connectToMongo();
```

```
PS D:\shopEZ> cd server
PS D:\shopEZ\server> node index.js
App server is running on port 3001
bad auth : authentication failed
PS D:\shopEZ\server> node index.js
App server is running on port 3001
Connected to your MongoDB database successfully
```

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.

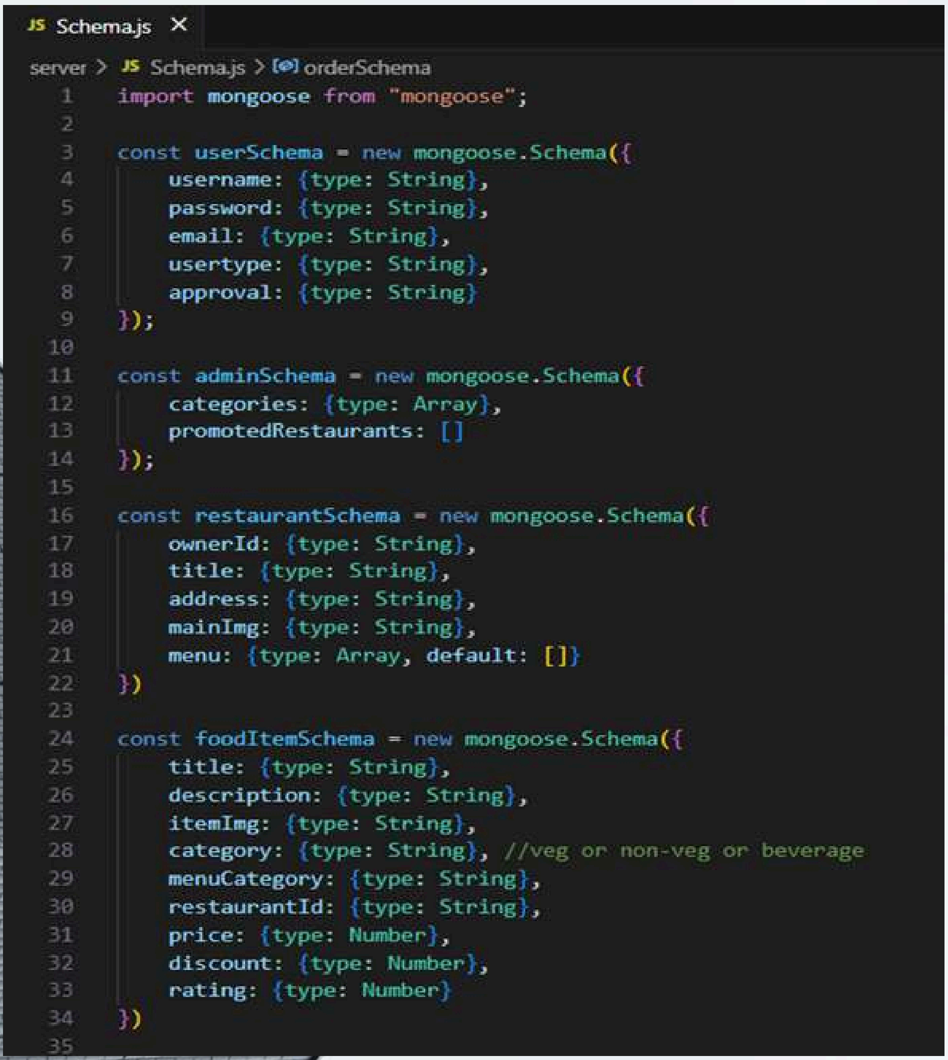
5. Admin Schema:

- 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 ×

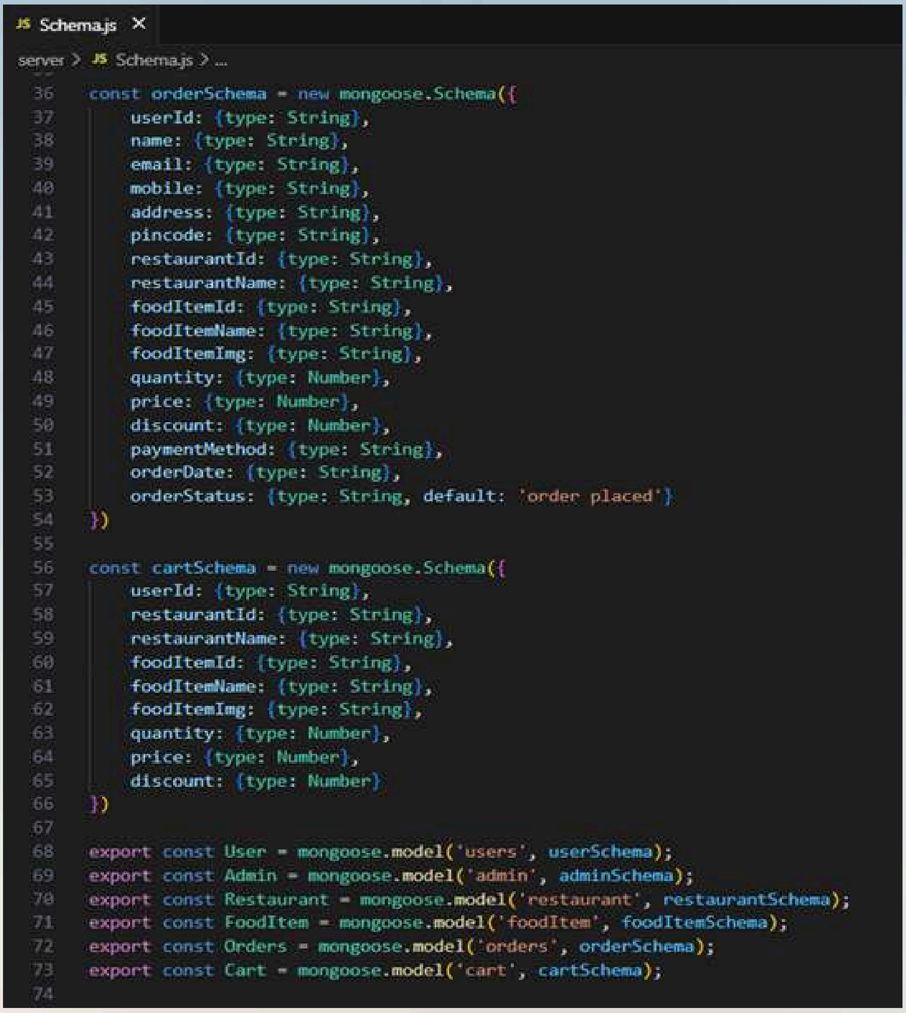
server > Schema.js > orderSchema

```
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}
})

```

This section outlines the setup, connection, and schema design for the MongoDB database used in the SSP Foods full-stack food ordering application.

Think of the SSP Foods App as a busy digital restaurant. Behind the beautiful menu and seamless ordering experience lies a smart kitchen — the MongoDB Database — quietly making everything work.

This code defines two important Mongoose schemas for managing Orders and Carts in your food ordering application using MongoDB.



```
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);
```

This block exports Mongoose models so that other files (like routes or controllers) can use them

```
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);
```

These models are essential for querying and manipulating your MongoDB collections.

Backend Development

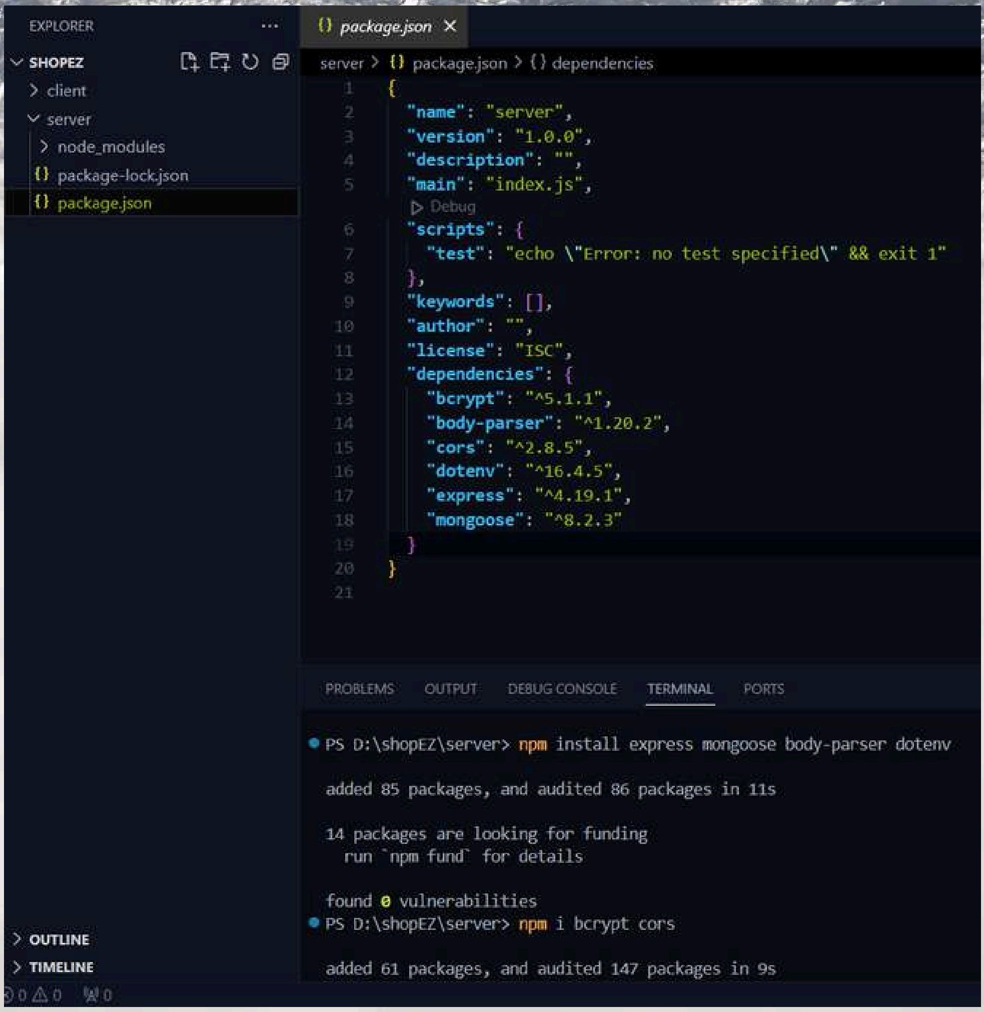
1. Initialize Project:

- Create a new directory:
  mkdir sb-foods-app && cd sb-foods-app

- Initialize with npm:
  npm init -y

- Install dependencies:
  npm install express mongoose dotenv cors body-parser

Reference Image:



server > package.json > dependencies

```json
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcrypt": "^5.1.1",
    "body-parser": "^1.20.2",
    "cors": "^2.8.5",
    "dotenv": "^16.4.5",
    "express": "^4.19.1",
    "mongoose": "^8.2.3"
  }
}
```

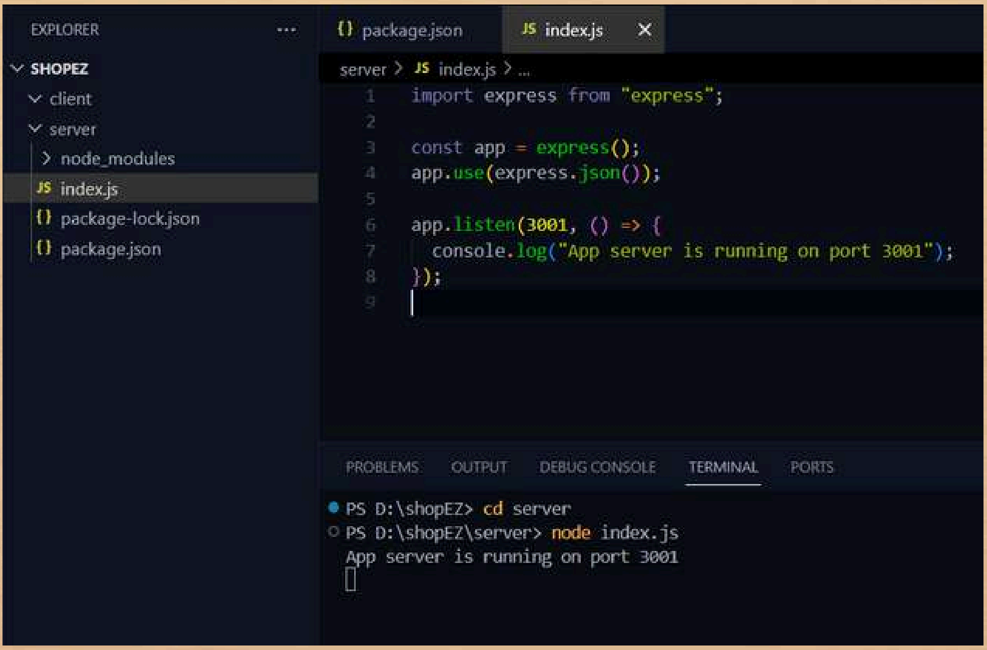
```
PS D:\shopEZ\server> npm install express mongoose body-parser dotenv

added 85 packages, and audited 86 packages in 11s

14 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS D:\shopEZ\server> npm i bcrypt cors

added 61 packages, and audited 147 packages in 9s
```

2. Setup Express Server:

Create index.js in root

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
const bodyParser = require("body-parser");
require("dotenv").config();

const app = express();
const PORT = process.env.PORT || 5000;

app.use(cors());
app.use(bodyParser.json());

app.get("/", (req, res) => {
  res.send("Server is running");
});

app.listen(PORT, () => {
  console.log(`Server started on port ${PORT}`);
});
```

Reference Image:



```
import express from "express";

const app = express();
app.use(express.json());

app.listen(3001, () => {
  console.log("App server is running on port 3001");
});
```

```
PS D:\shopEZ> cd server
PS D:\shopEZ\server> node index.js
App server is running on port 3001
```

3. Database Configuration:

- Use MongoDB Atlas or local MongoDB Compass.
- Set .env variables:

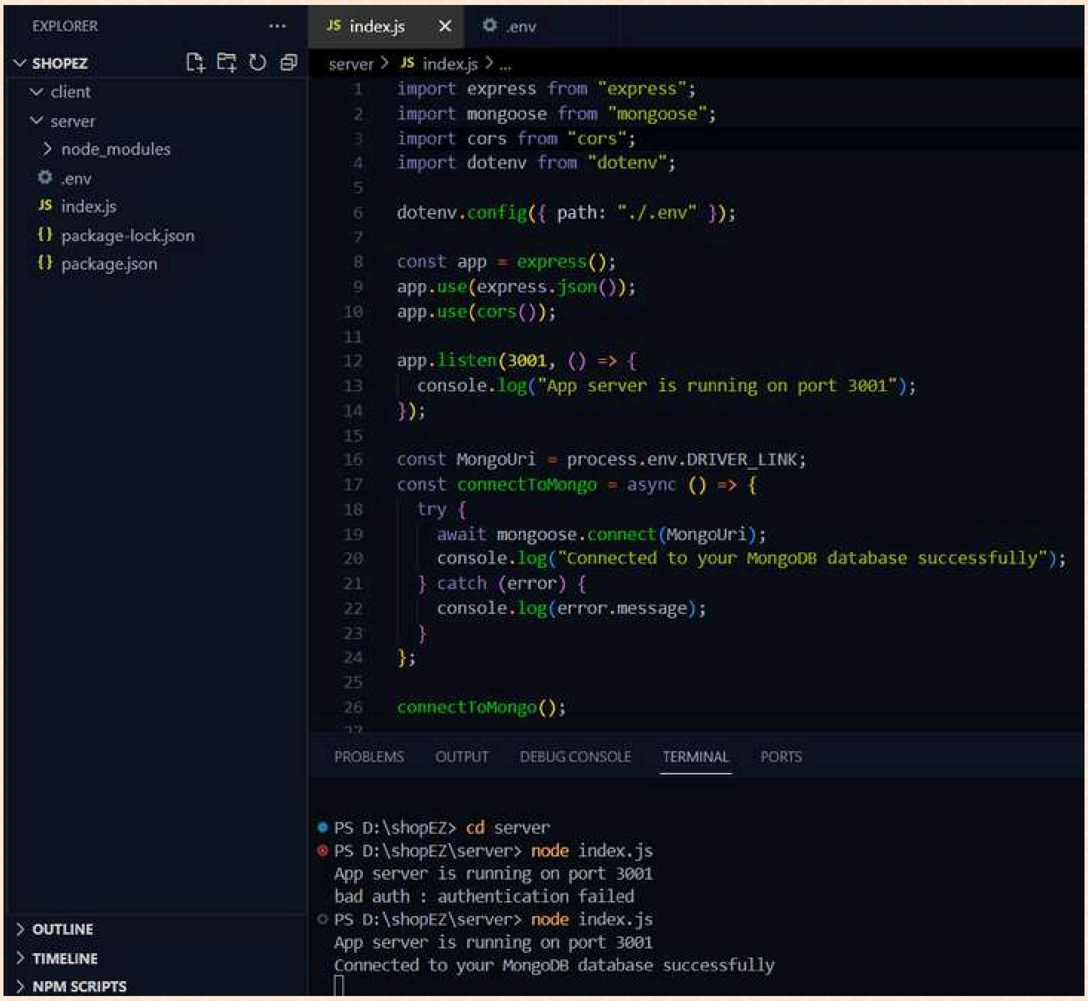
```
MONGO_URI=your_mongodb_connection_string
```

- Connect to DB:

```
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
})
.then(() => console.log("MongoDB Connected"))
.catch((err) => console.error(err));
```

Reference Video: Google Drive MongoDB Setup

Reference Article: MongoDB Atlas Guide



```
import express from "express";
import mongoose from "mongoose";
import cors from "cors";
import dotenv from "dotenv";

dotenv.config({ path: "./.env" });

const app = express();
app.use(express.json());
app.use(cors());

app.listen(3001, () => {
  console.log("App server is running on port 3001");
});

const MongoUri = process.env.DRIVER_LINK;
const connectToMongo = async () => {
  try {
    await mongoose.connect(MongoUri);
    console.log("Connected to your MongoDB database successfully");
  } catch (error) {
    console.log(error.message);
  }
};

connectToMongo();
```

```
PS D:\shopEZ> cd server
PS D:\shopEZ\server> node index.js
App server is running on port 3001
bad auth : authentication failed
PS D:\shopEZ\server> node index.js
App server is running on port 3001
Connected to your MongoDB database successfully
```

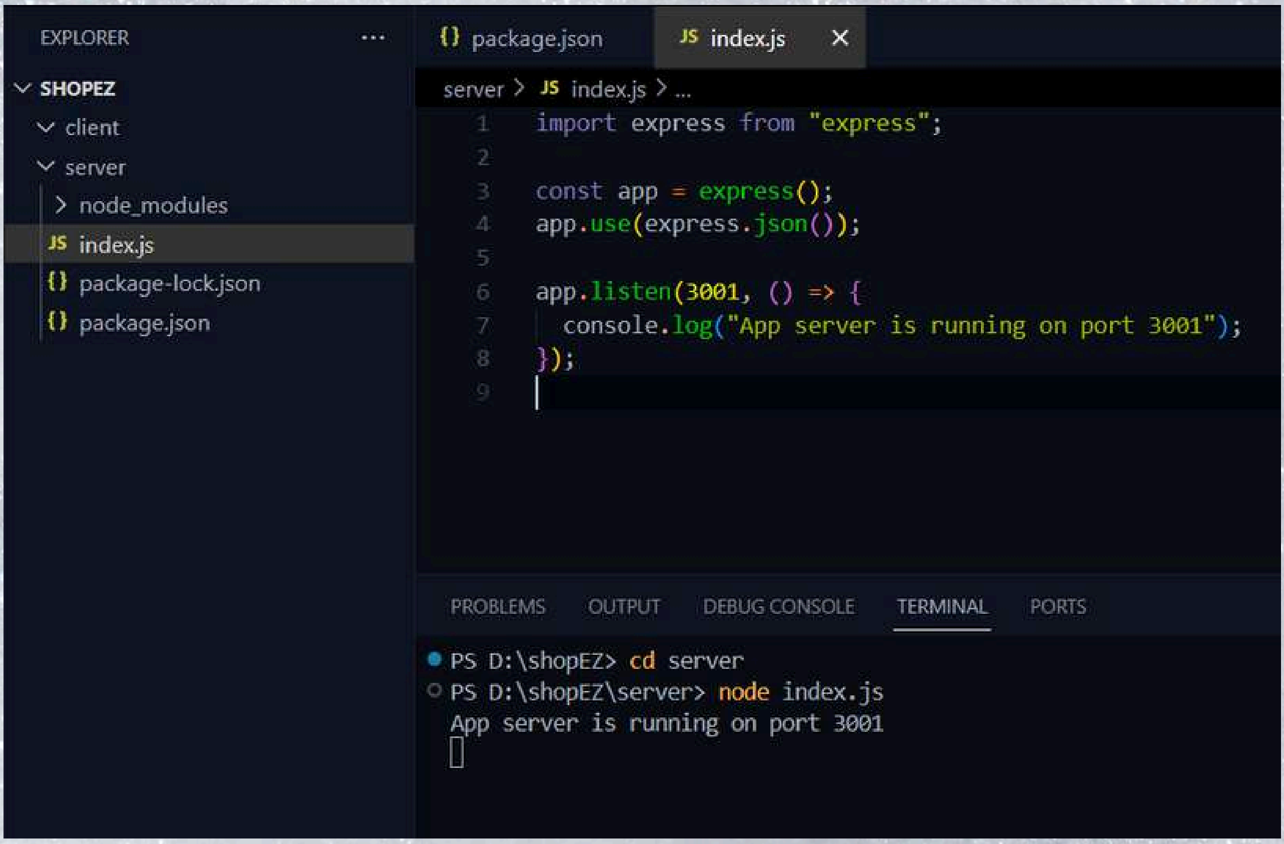
4. Create Express Routes:

- **Structure:**

```
/routes
  - users.js
  - products.js
  - orders.js
```

- **Sample route:**

```
const express = require("express");
const router = express.Router();

router.get("/", (req, res) => {
  res.send("All users");
});

module.exports = router;
```



```
import express from "express";

const app = express();
app.use(express.json());

app.listen(3001, () => {
  console.log("App server is running on port 3001");
});
```

```
PS D:\shopEZ> cd server
PS D:\shopEZ\server> node index.js
App server is running on port 3001
```

5. Implement Data Models (Schemas):

- Use Mongoose schemas for Users, Products, Orders, Carts, Admins, and Restaurants.
- Example schema:

```
const mongoose = require("mongoose");
const userSchema = new mongoose.Schema({
  username: String,
  email: String,
  password: String,
});
module.exports = mongoose.model("User", userSchema);
```

6. User Authentication:

- Implement registration, login, and protected routes.
- Use JWT for auth tokens.

7. Product and Order Handling:

- Endpoints for:
- Product listing (GET)
- Add product (POST)
- Place order (POST)

8. Admin Functionality:

- Admin dashboard endpoints:
- Manage users
- Add/edit/delete products
- View orders
- Middleware for role-based access

9. Error Handling:

- Global error middleware:

```
app.use((err, req, res, next) => {
  res.status(500).json({ error: err.message });
});
```

FRONTEND DEVELOPMENT

1. Setup React Application:

- Create a React app in the client folder:
  npx create-react-app client

- Install required libraries:
  npm install axios react-router-dom bootstrap

- Create required pages and components and set up routing using react-router-dom.

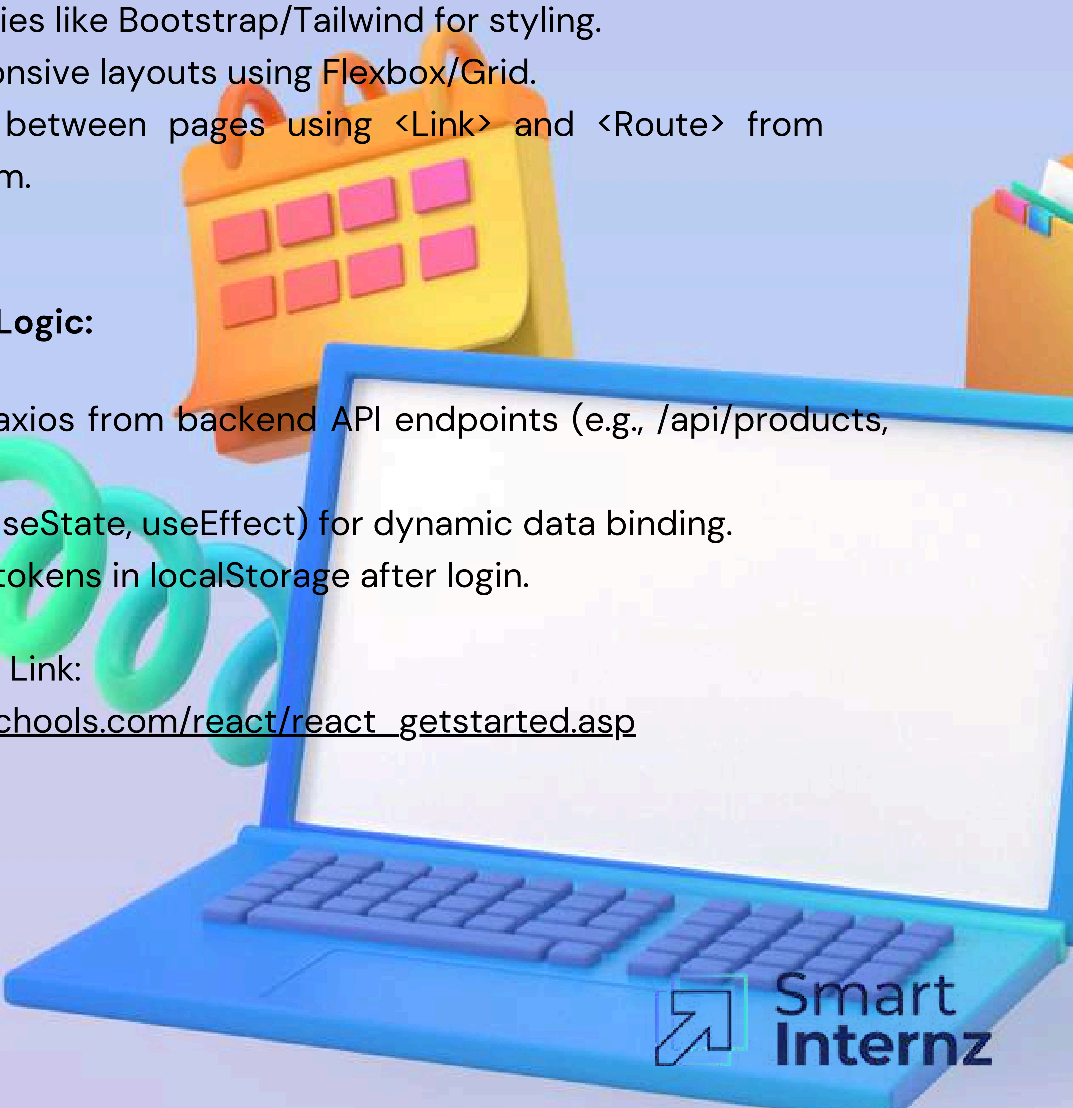
2. Design UI Components:

- Break down the UI into reusable components like Navbar, Footer, ProductCard, etc.
- Use CSS or libraries like Bootstrap/Tailwind for styling.
- Implement responsive layouts using Flexbox/Grid.
- Add navigation between pages using <Link> and <Route> from react-router-dom.

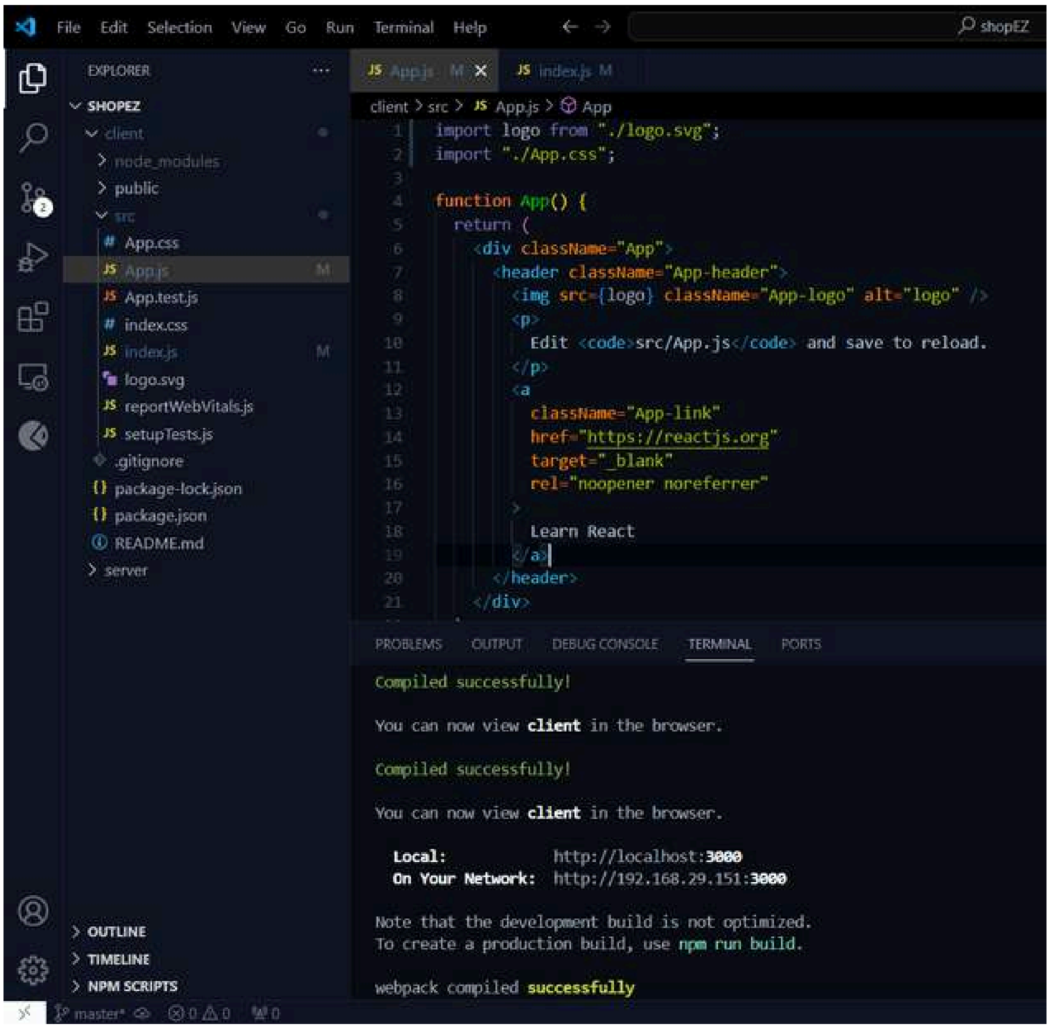
3. Implement Frontend Logic:

- Fetch data using axios from backend API endpoints (e.g., /api/products, /api/users/login).
- Use React state (useState, useEffect) for dynamic data binding.
- Store user info or tokens in localStorage after login.

Reference Article Link:
https://www.w3schools.com/react/react_getstarted.asp



Reference Image:



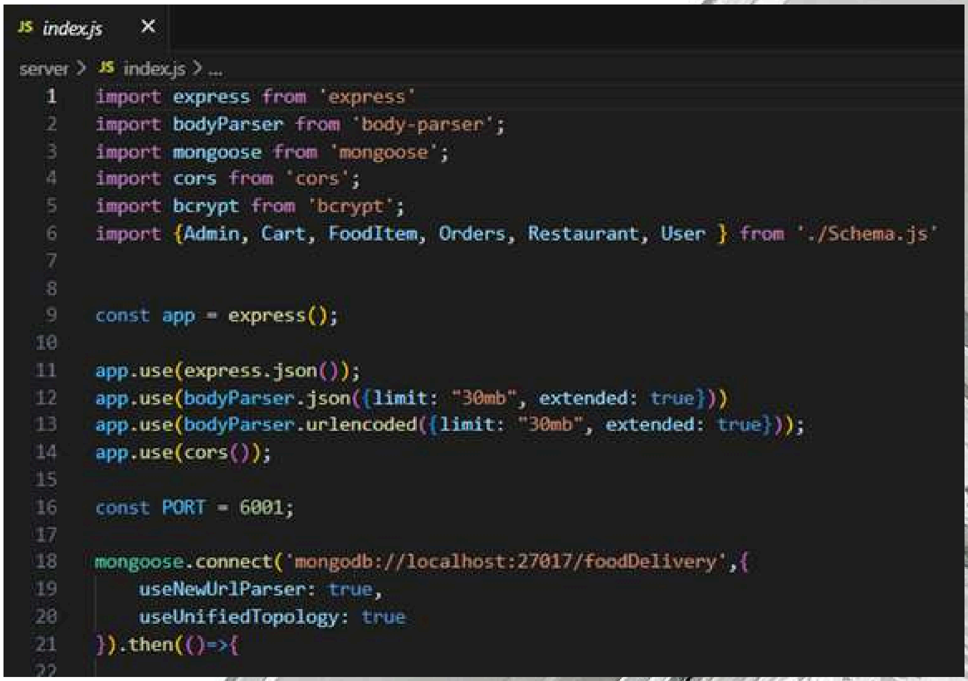
```
import logo from "./logo.svg";
import "./App.css";

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
```

```
Compiled successfully!

You can now view client in the browser.

Compiled successfully!

You can now view client in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.29.151:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

Code Explanation:

Server setup:

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

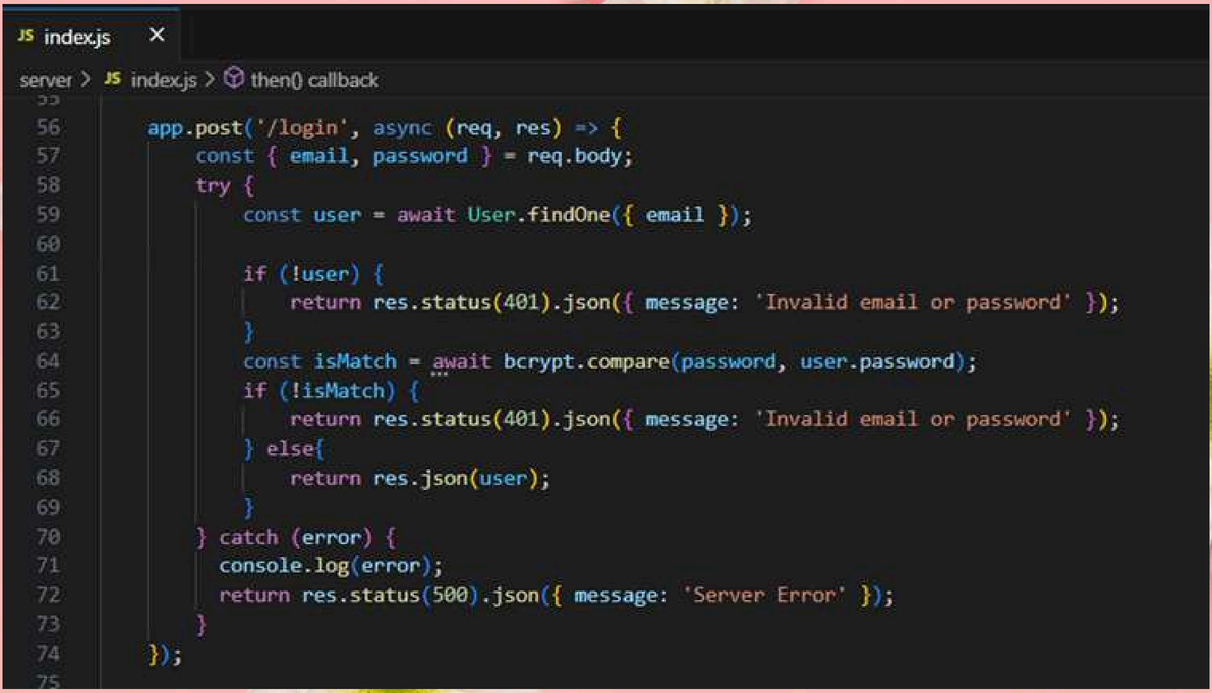


```
import express from 'express'
import bodyParser from 'body-parser';
import mongoose from 'mongoose';
import cors from 'cors';
import bcrypt from 'bcrypt';
import {Admin, Cart, FoodItem, Orders, Restaurant, User } from './Schema.js'


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(()=>{
```

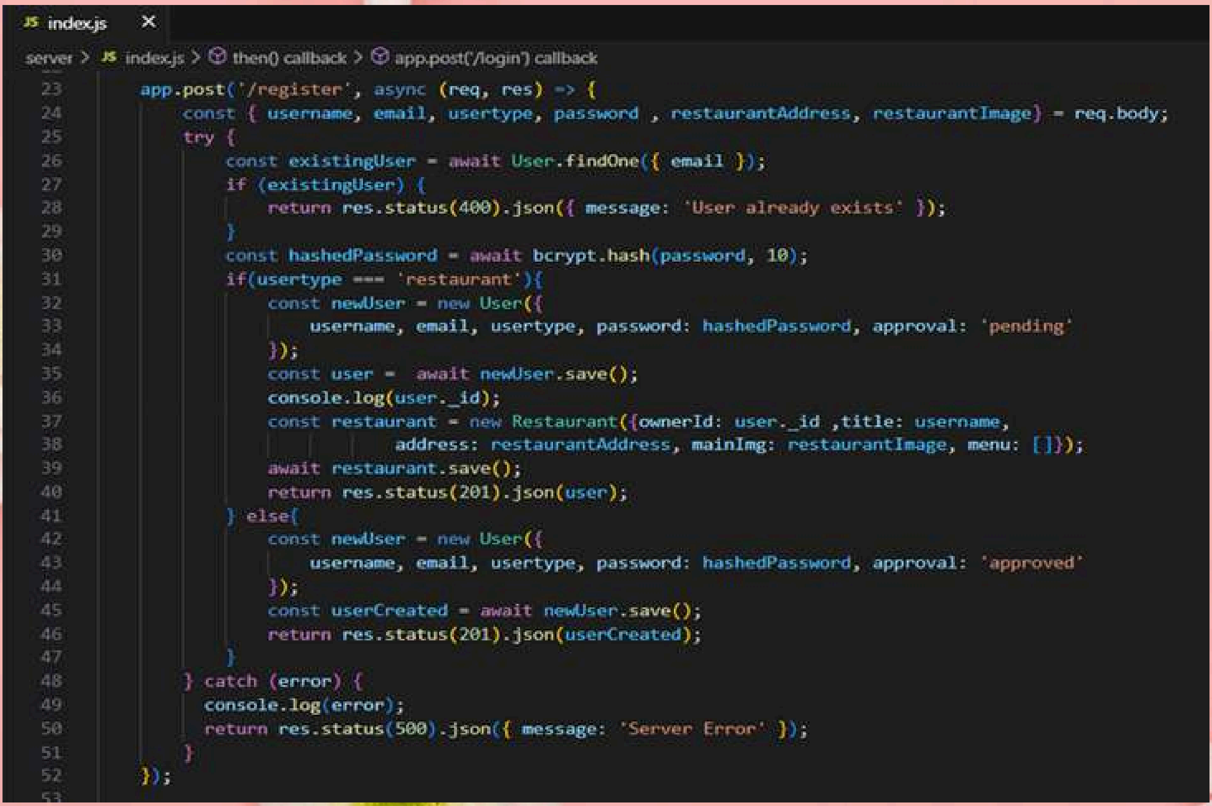
User Authentication:

Backend

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



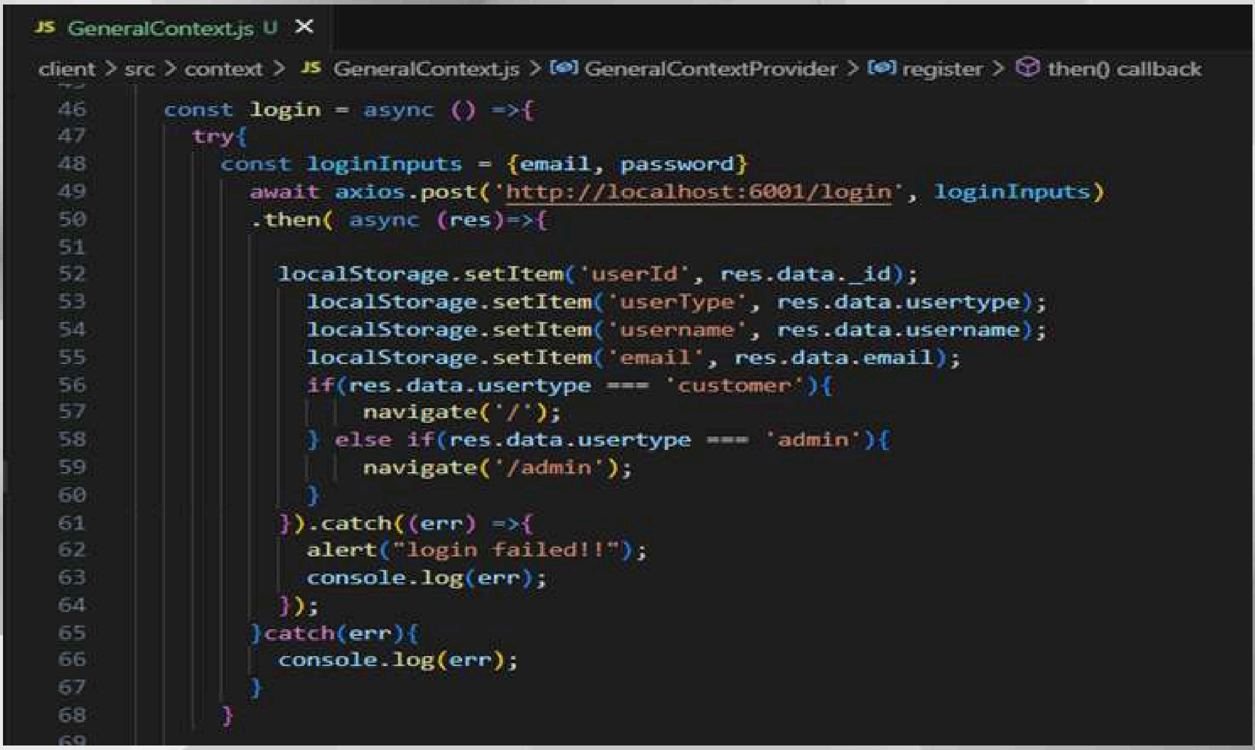
```javascript
app.post('/login', async (req, res) => {
    const { email, password } = req.body;
    try {
        const user = await User.findOne({ email });

        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

```javascript
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

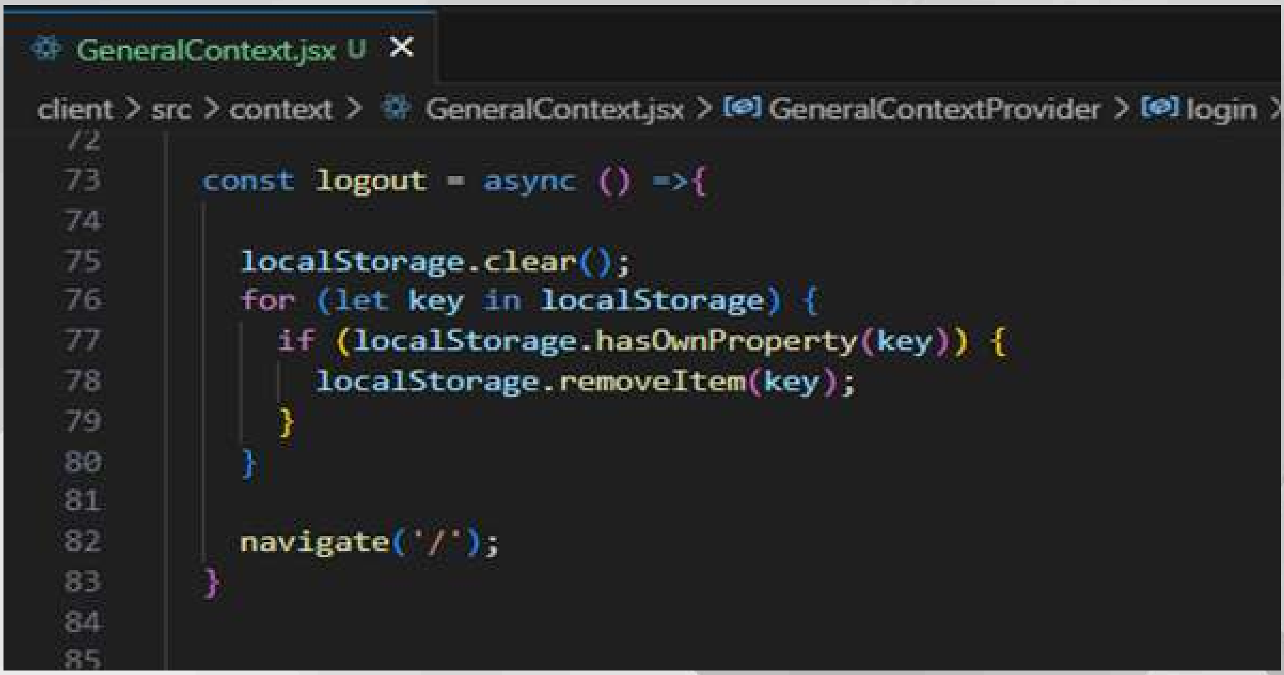
Login:



JS GeneralContext.js U ×
client > src > context > JS 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);
        }
    }
```

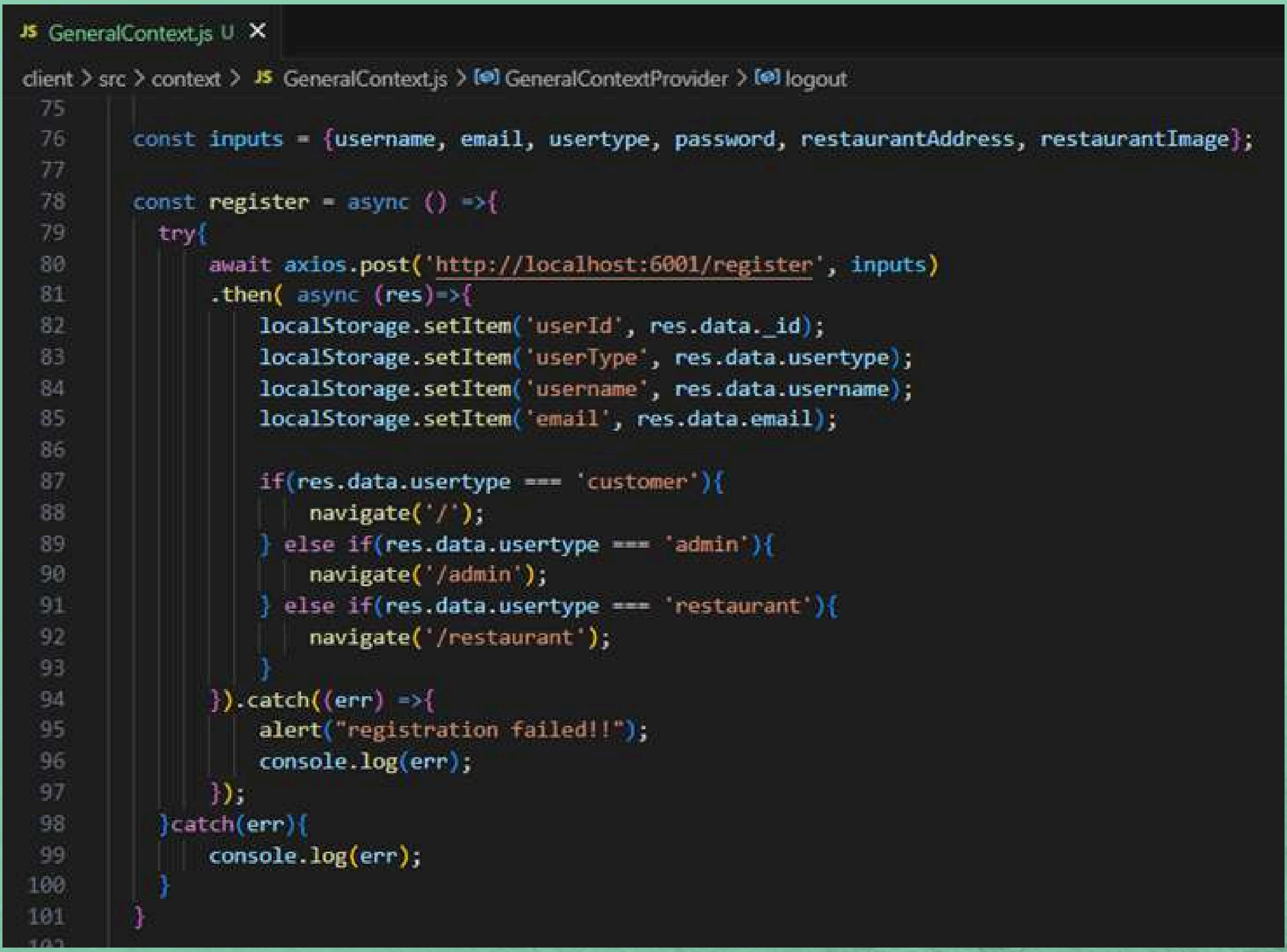
Logout:



GeneralContext.jsx U ×
client > src > context > GeneralContext.jsx > GeneralContextProvider > login >

```
  const logout = async () =>{

    localStorage.clear();
    for (let key in localStorage) {
      if (localStorage.hasOwnProperty(key)) {
        localStorage.removeItem(key);
      }
    }

    navigate('/');
  }
```

Register:


GeneralContext.js U
client > src > context > GeneralContext.js > GeneralContextProvider > logout

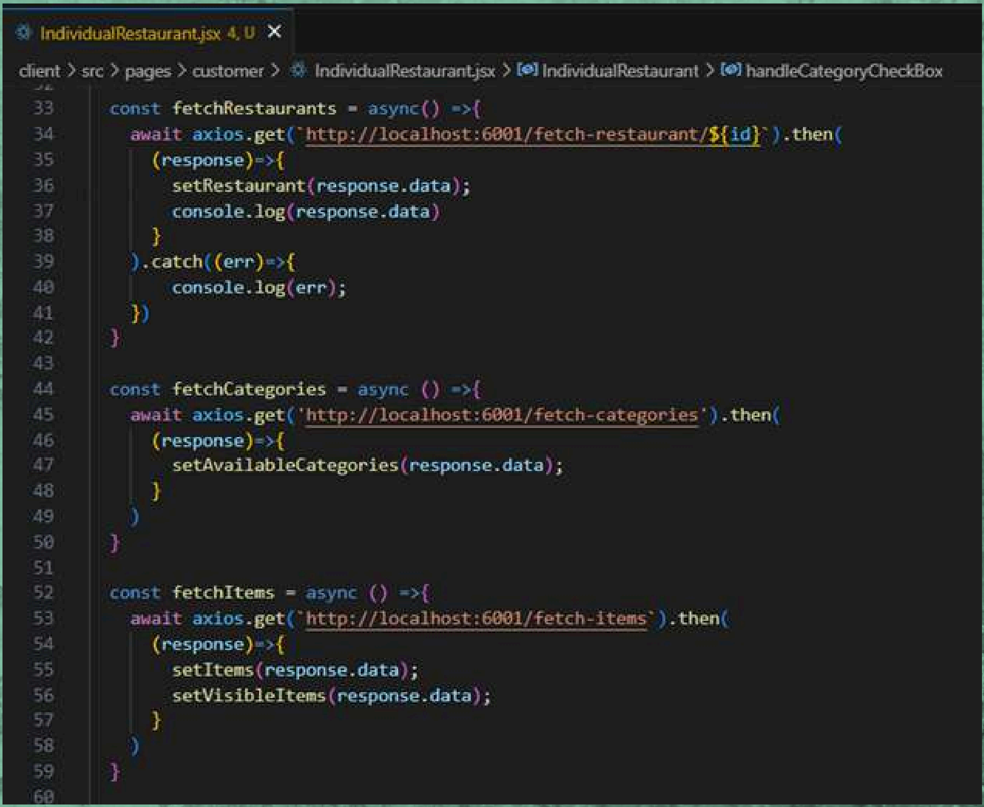
```
    const inputs = {username, email, usertype, password, restaurantAddress, restaurantImage};

    const register = async () =>{
      try{
          await axios.post('http://localhost:6001/register', inputs)
          .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');
              } else if(res.data.usertype === 'restaurant'){
                  navigate('/restaurant');
              }
          }).catch((err) =>{
              alert("registration failed!!");
              console.log(err);
          });
      }catch(err){
          console.log(err);
      }
    }
```

All Products (User):

Frontend

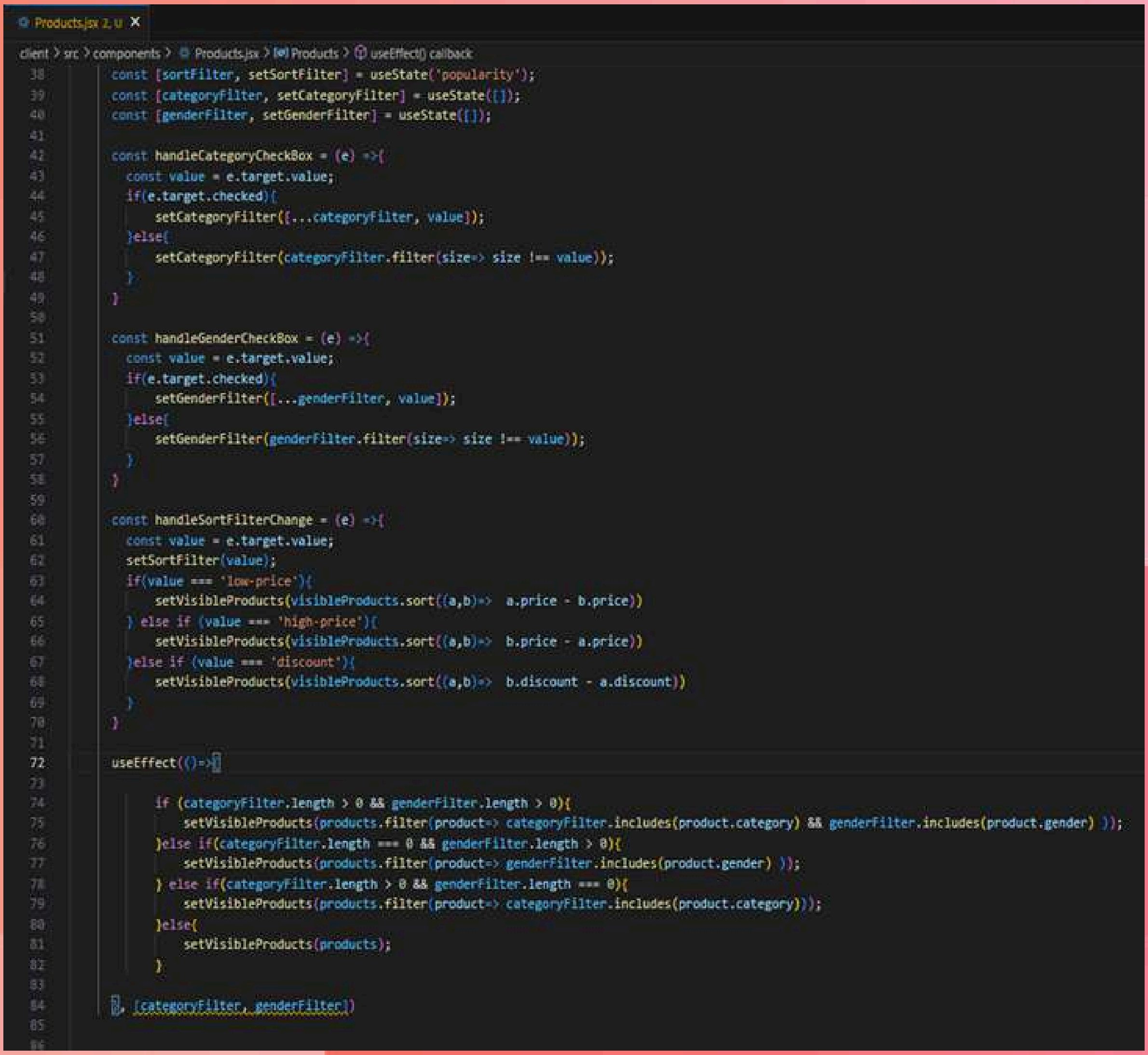
In the home page, we'll fetch all the products available in the platform along with the filters.

Fetching food items:


IndividualRestaurant.jsx 4, U
client > src > pages > customer > IndividualRestaurant.jsx > IndividualRestaurant > handleCategoryCheckBox

```
  const fetchRestaurants = async() =>{
    await axios.get(`http://localhost:6001/fetch-restaurant/${id}`).then(
      (response)=>{
        setRestaurant(response.data);
        console.log(response.data)
      }
    ).catch((err)=>{
        console.log(err);
    })
  }

  const fetchCategories = async () =>{
    await axios.get('http://localhost:6001/fetch-categories').then(
      (response)=>{
        setAvailableCategories(response.data);
      }
    )
  }

  const fetchItems = async () =>{
    await axios.get(`http://localhost:6001/fetch-items`).then(
      (response)=>{
        setItems(response.data);
        setVisibleItems(response.data);
      }
    )
  }
```

Filtering products:

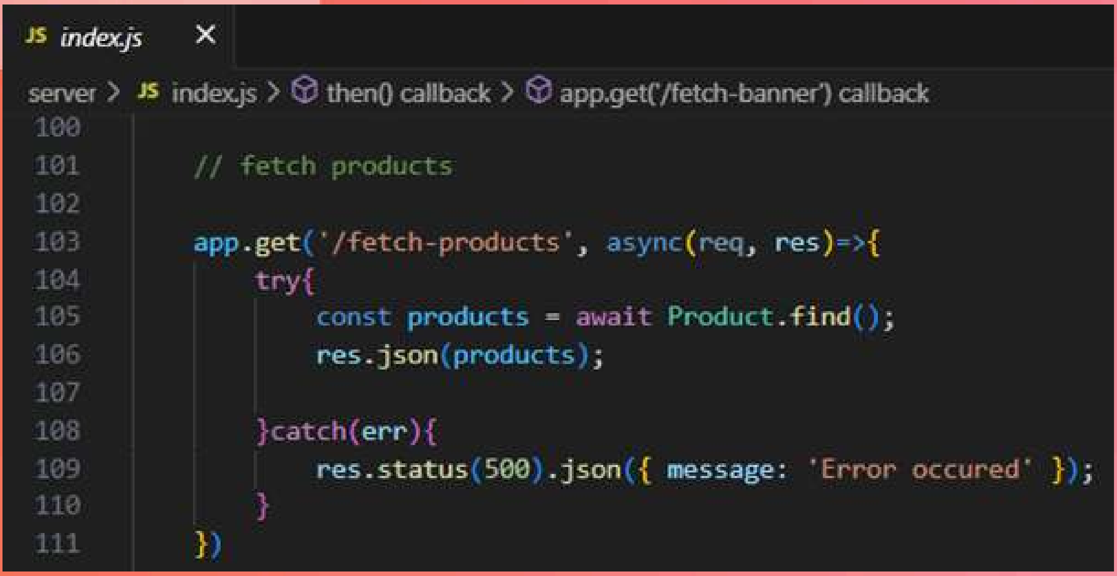


```
Products.jsx 2, U
client > src > components > Products.jsx > Products > useEffect() callback
```

```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.



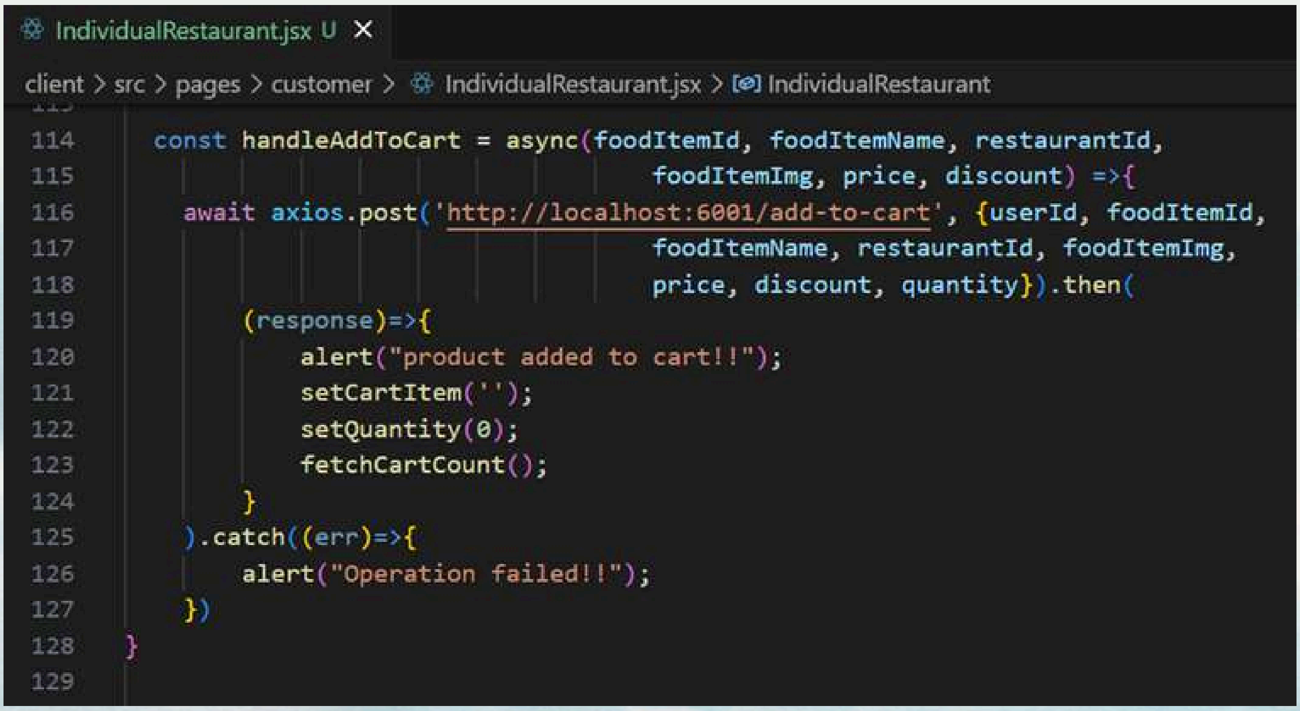
```
index.js
server > index.js > then() callback > app.get('/fetch-banner') callback
```

```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.



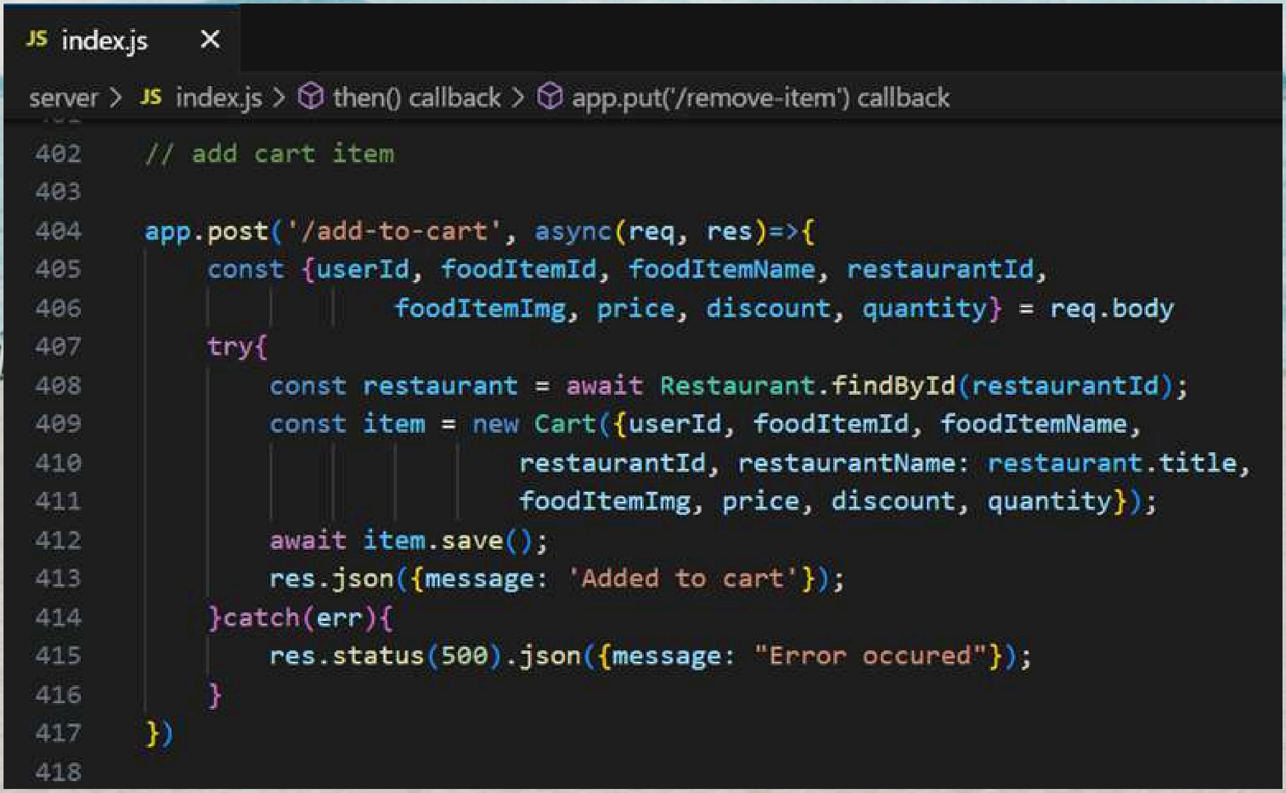
IndividualRestaurant.jsx U ×

client > src > pages > customer > IndividualRestaurant.jsx > IndividualRestaurant

```
  const handleAddToCart = async(foodItemId, foodItemName, restaurantId,
                                foodItemImg, price, discount) =>{
    await axios.post('http://localhost:6001/add-to-cart', {userId, foodItemId,
                                foodItemName, restaurantId, foodItemImg,
                                price, discount, quantity}).then(
        (response)=>{
            alert("product added to cart!!");
            setCartItem('');
            setQuantity(0);
            fetchCartCount();
        }
    ).catch((err)=>{
        alert("Operation failed!!");
    })
}
```

Backend

Add product to cart:



index.js ×

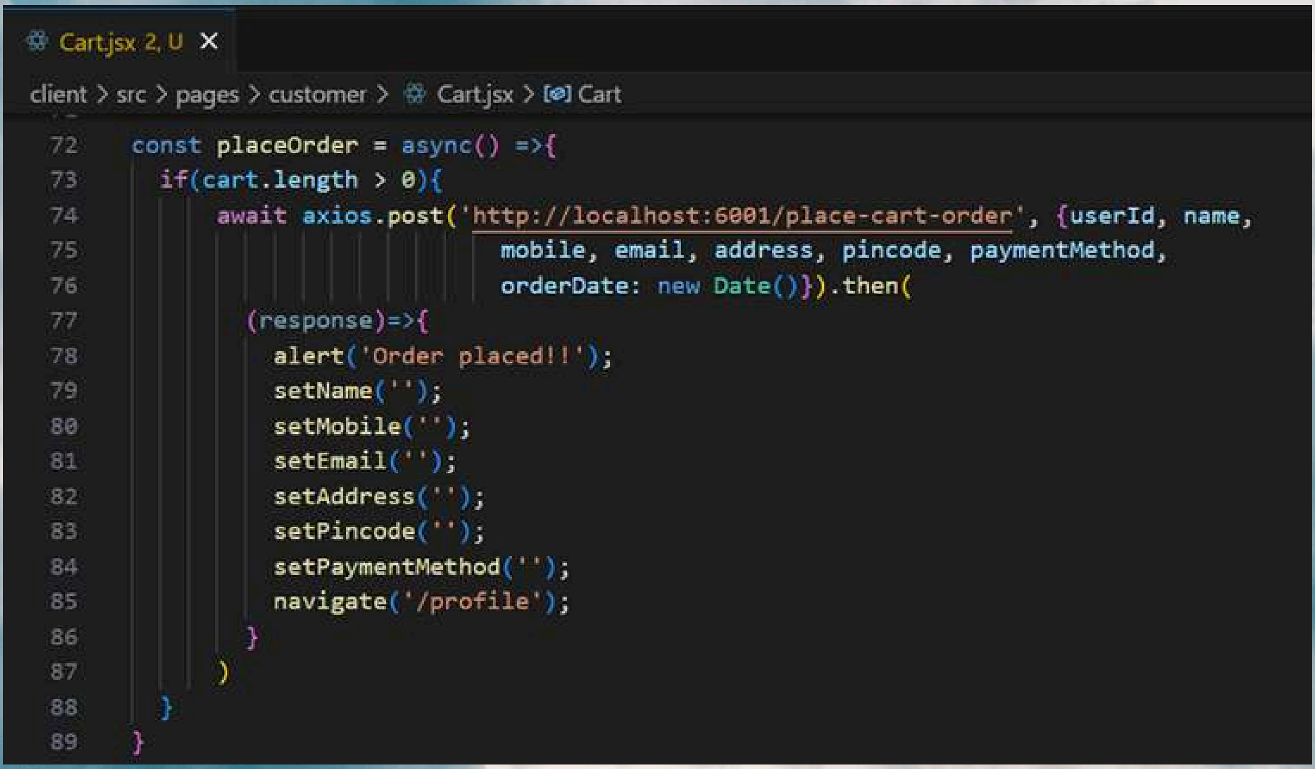
server > index.js > then() callback > app.put('/remove-item') callback

```
// add cart item

app.post('/add-to-cart', async(req, res)=>{
    const {userId, foodItemId, foodItemName, restaurantId,
                foodItemImg, price, discount, quantity} = req.body
    try{
        const restaurant = await Restaurant.findById(restaurantId);
        const item = new Cart({userId, foodItemId, foodItemName,
                            restaurantId, restaurantName: restaurant.title,
                            foodItemImg, price, discount, quantity});
        await item.save();
        res.json({message: 'Added to cart'});
    }catch(err){
        res.status(500).json({message: "Error occured"});
    }
})
```

Order products:

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

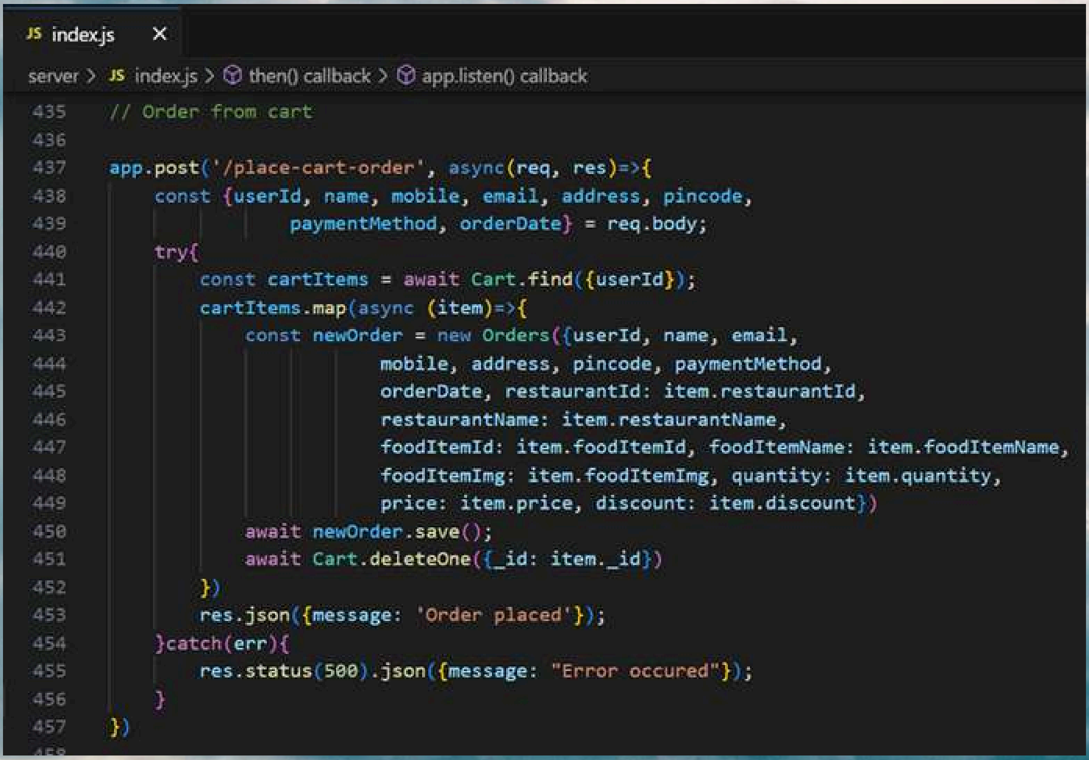
Frontend



```
const placeOrder = async() =>{
  if(cart.length > 0){
      await axios.post('http://localhost:6001/place-cart-order', {userId, name,
                                    mobile, email, address, pincode, paymentMethod,
                                    orderDate: new Date()}).then(
        (response)=>{
          alert('Order placed!!');
          setName('');
          setMobile('');
          setEmail('');
          setAddress('');
          setPincode('');
          setPaymentMethod('');
          navigate('/profile');
        }
      )
  }
}
```

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.

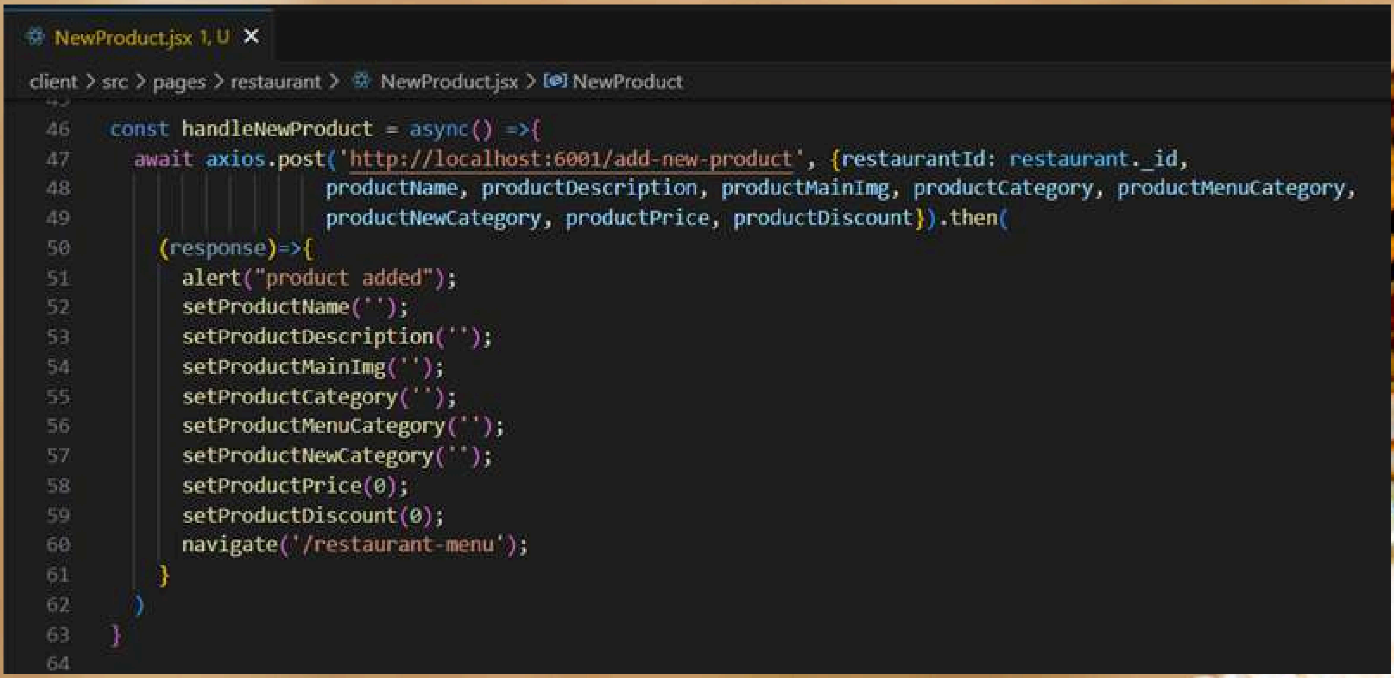


```
// Order from cart

app.post('/place-cart-order', async(req, res)=>{
    const {userId, name, mobile, email, address, pincode,
            paymentMethod, orderDate} = req.body;
    try{
        const cartItems = await Cart.find({userId});
        cartItems.map(async (item)=>{
            const newOrder = new Orders({userId, name, email,
                            mobile, address, pincode, paymentMethod,
                            orderDate, restaurantId: item.restaurantId,
                            restaurantName: item.restaurantName,
                            foodItemId: item.foodItemId, foodItemName: item.foodItemName,
                            foodItemImg: item.foodItemImg, quantity: item.quantity,
                            price: item.price, discount: item.discount})
            await newOrder.save();
            await Cart.deleteOne({_id: item._id})
        })
        res.json({message: 'Order placed'});
    }catch(err){
        res.status(500).json({message: "Error occured"});
    }
})
```

Add new product:

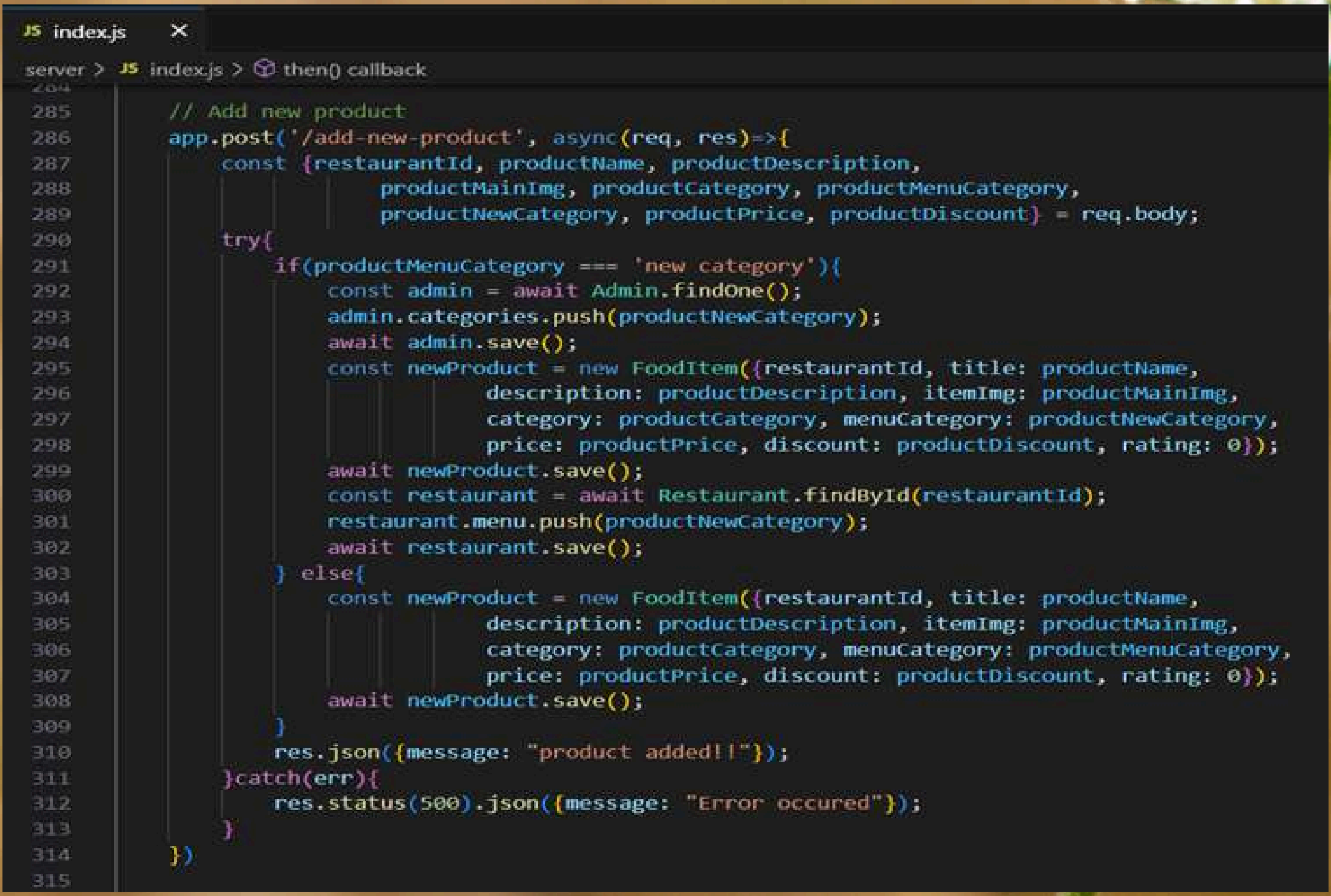
Here, in the admin dashboard, we will add a new product.

Frontend:


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

```jsx
const handleNewProduct = async() =>{
  await axios.post('http://localhost:6001/add-new-product', {restaurantId: restaurant._id,
                productName, productDescription, productMainImg, productCategory, productMenuCategory,
                productNewCategory, productPrice, productDiscount}).then(
    (response)=>{
      alert("product added");
      setProductName('');
      setProductDescription('');
      setProductMainImg('');
      setProductCategory('');
      setProductMenuCategory('');
      setProductNewCategory('');
      setProductPrice(0);
      setProductDiscount(0);
      navigate('/restaurant-menu');
    }
  )
}
```

Backend:



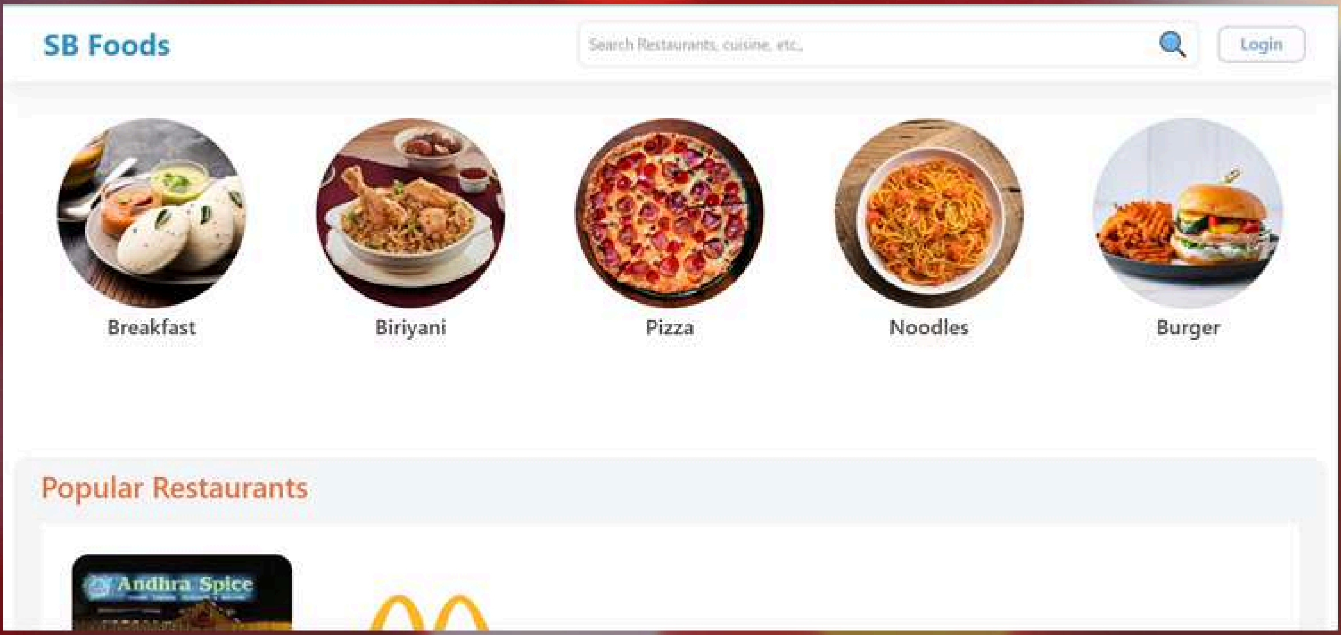
```js
// Add new product
app.post('/add-new-product', async(req, res)=>{
    const {restaurantId, productName, productDescription,
            productMainImg, productCategory, productMenuCategory,
            productNewCategory, productPrice, productDiscount} = req.body;
    try{
        if(productMenuCategory === '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.

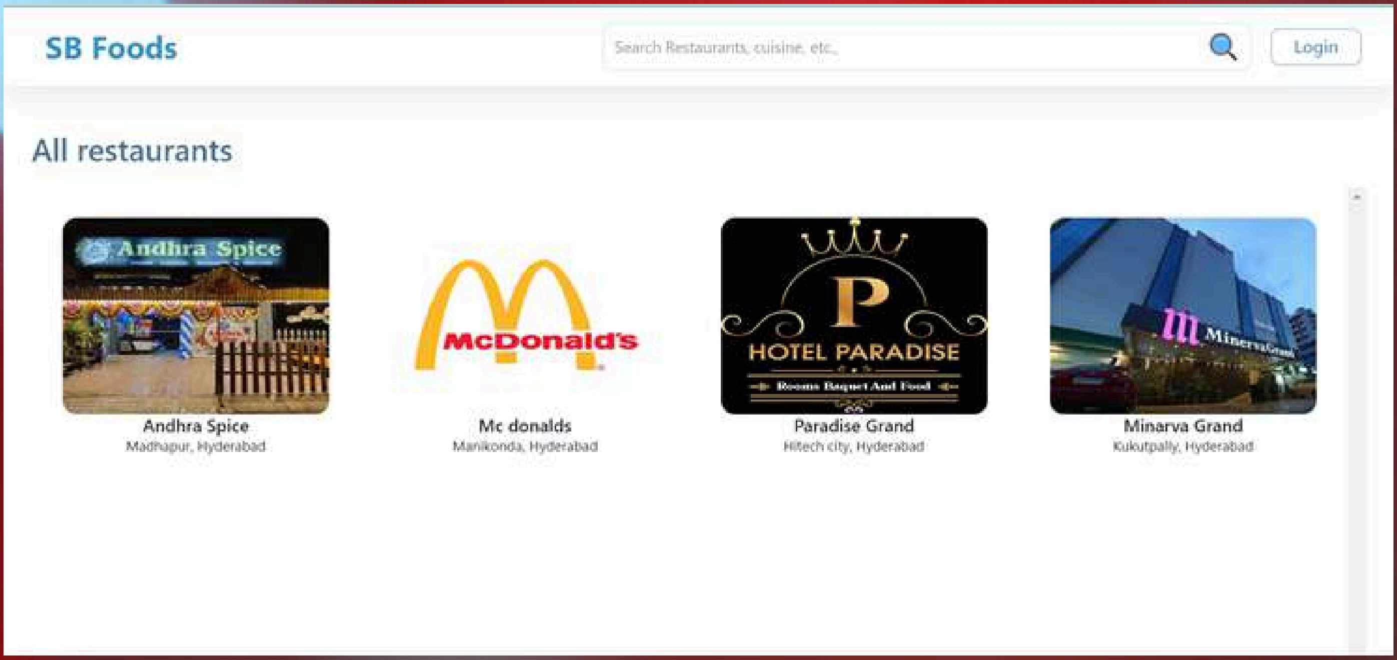
Smart Internz

Project Implementation & Execution

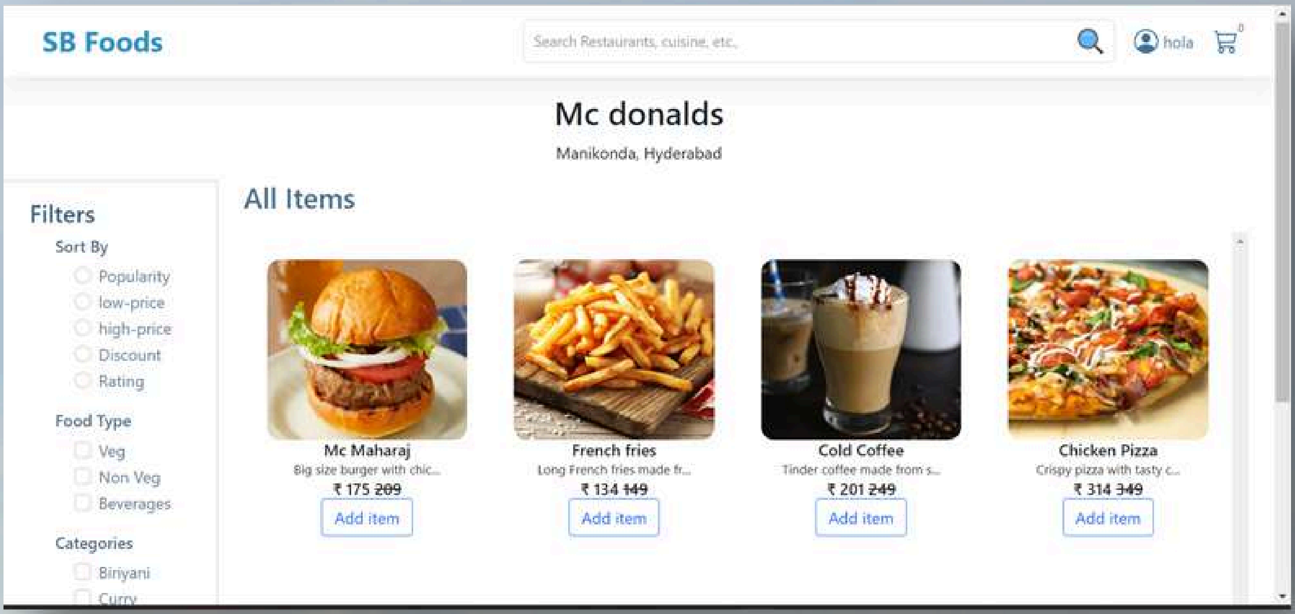
Landing page



Landing page



Restaurant Menu


SB Foods

Search Restaurants, cuisine, etc.,

hola

Mc donalds

Manikonda, Hyderabad

Filters

Sort By

- Popularity
- low-price
- high-price
- Discount
- Rating

Food Type

- Veg
- Non Veg
- Beverages

Categories

- Biriyani
- Curry

All Items

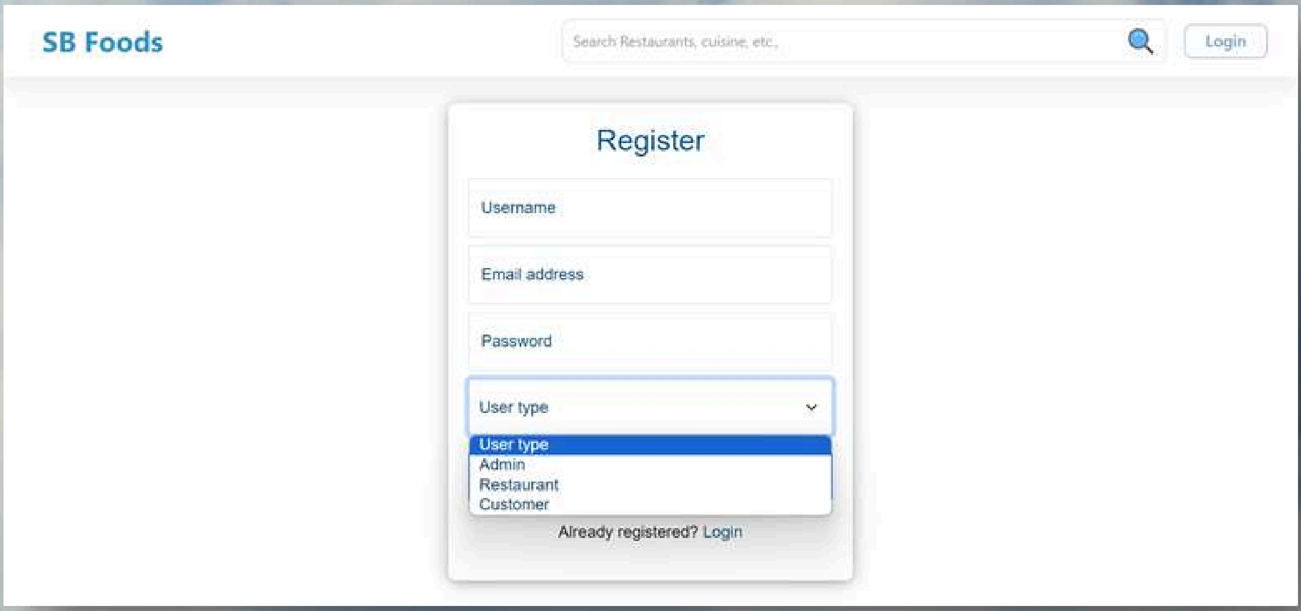
Mc Maharaj
Big size burger with chic...
₹ 175 ~~209~~
Add item

French fries
Long French fries made fr...
₹ 134 ~~149~~
Add item

Cold Coffee
Tinder coffee made from s...
₹ 201 ~~249~~
Add item

Chicken Pizza
Crispy pizza with tasty c...
₹ 314 ~~349~~
Add item

Authentication


SB Foods

Search Restaurants, cuisine, etc.,

Login

Register

Username

Email address

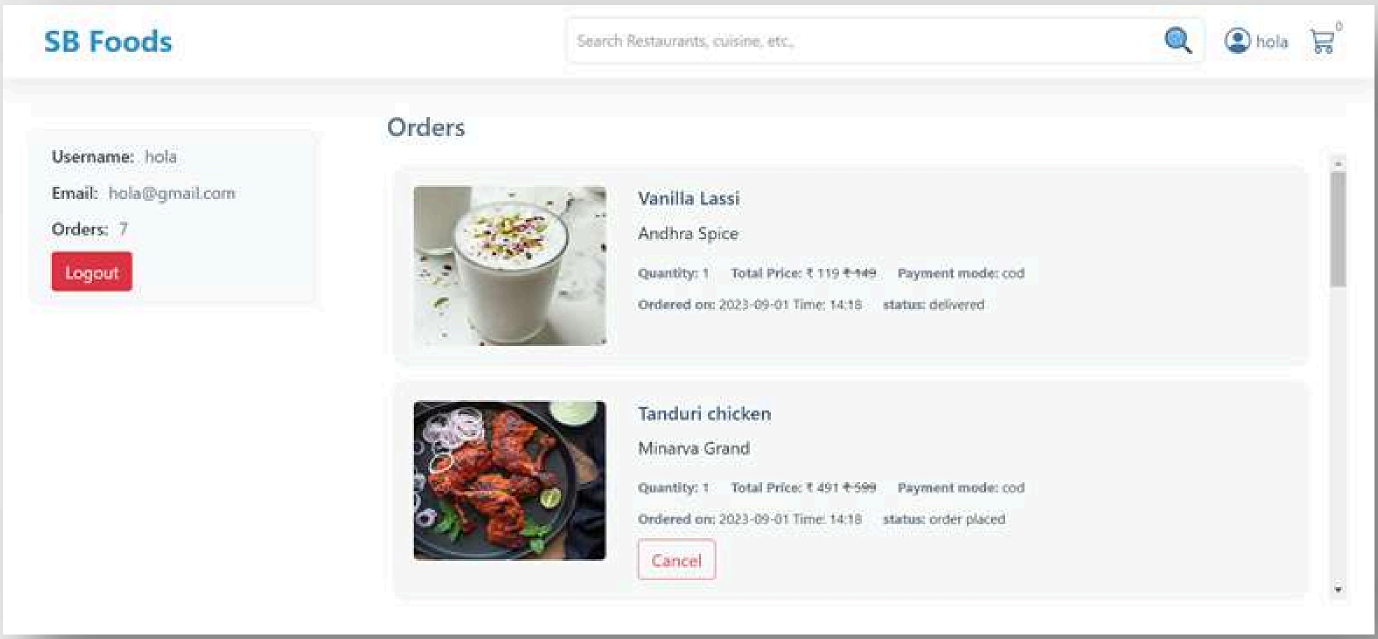
Password

User type

User type
Admin
Restaurant
Customer

Already registered? Login

User Profile



SB Foods

Search Restaurants, cuisine, etc.,

hola 0

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

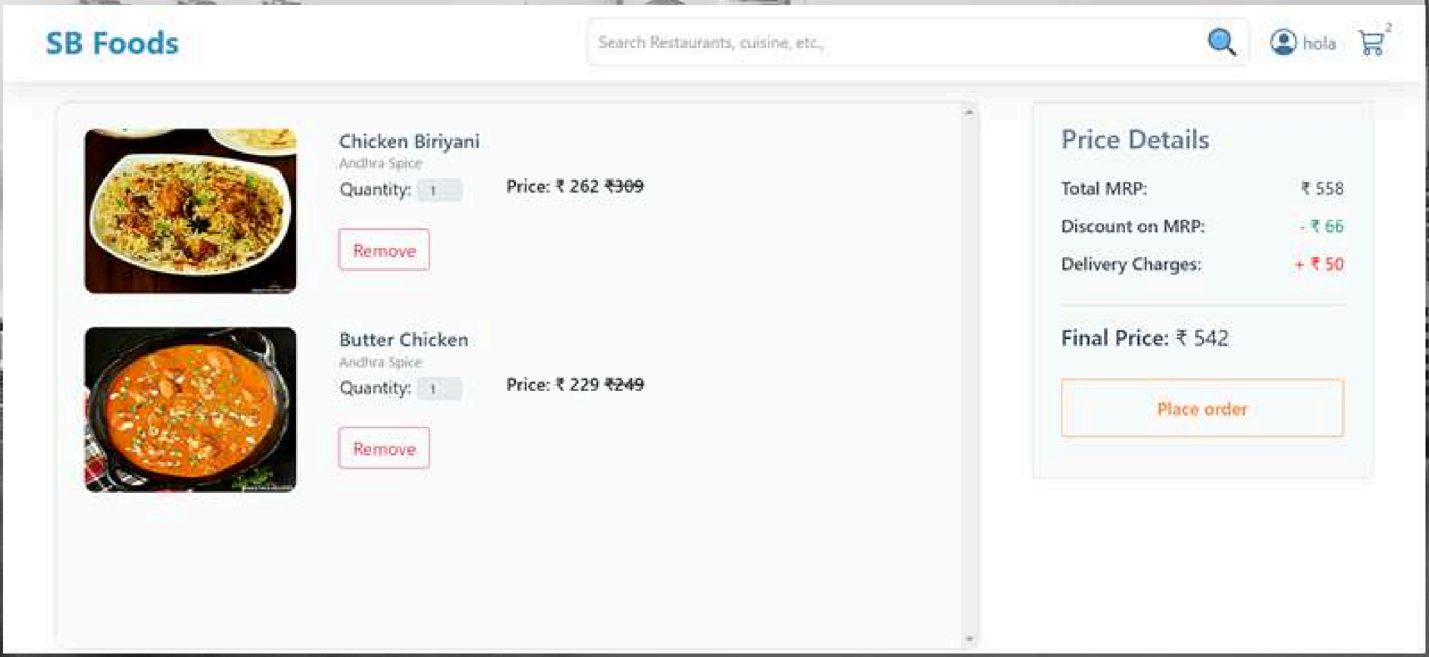
Minarva Grand

Quantity: 1 Total Price: ₹ 491 ~~₹ 599~~ Payment mode: cod

Ordered on: 2023-09-01 Time: 14:18 status: order placed

Cancel

Cart



SB Foods

Search Restaurants, cuisine, etc.,

hola 2

Chicken Biriyani

Andhra Spice

Quantity: 1 Price: ₹ 262 ~~₹309~~

Remove

Butter Chicken

Andhra Spice

Quantity: 1 Price: ₹ 229 ~~₹249~~

Remove

Price Details

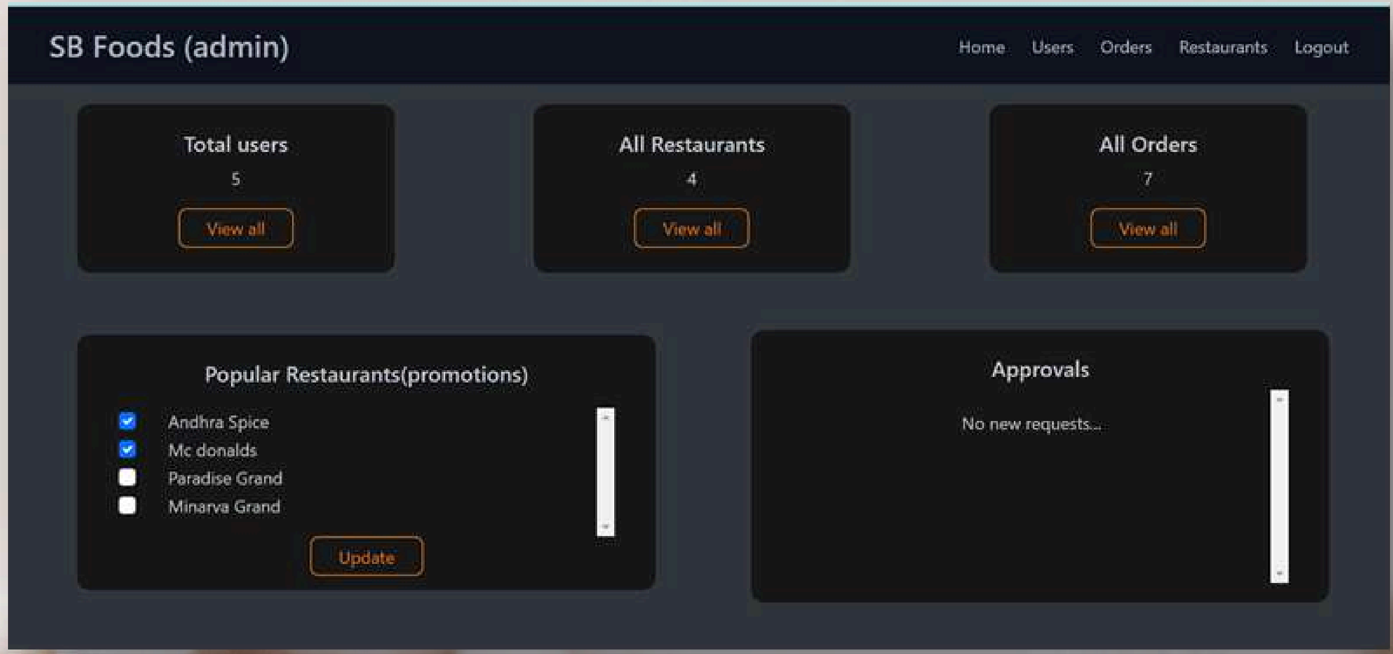
Total MRP:	₹ 558
Discount on MRP:	- ₹ 66
Delivery Charges:	+ ₹ 50

Final Price: ₹ 542

Place order

Smart Internz

Admin dashboard



Total users

5

All Restaurants

4

All Orders

7

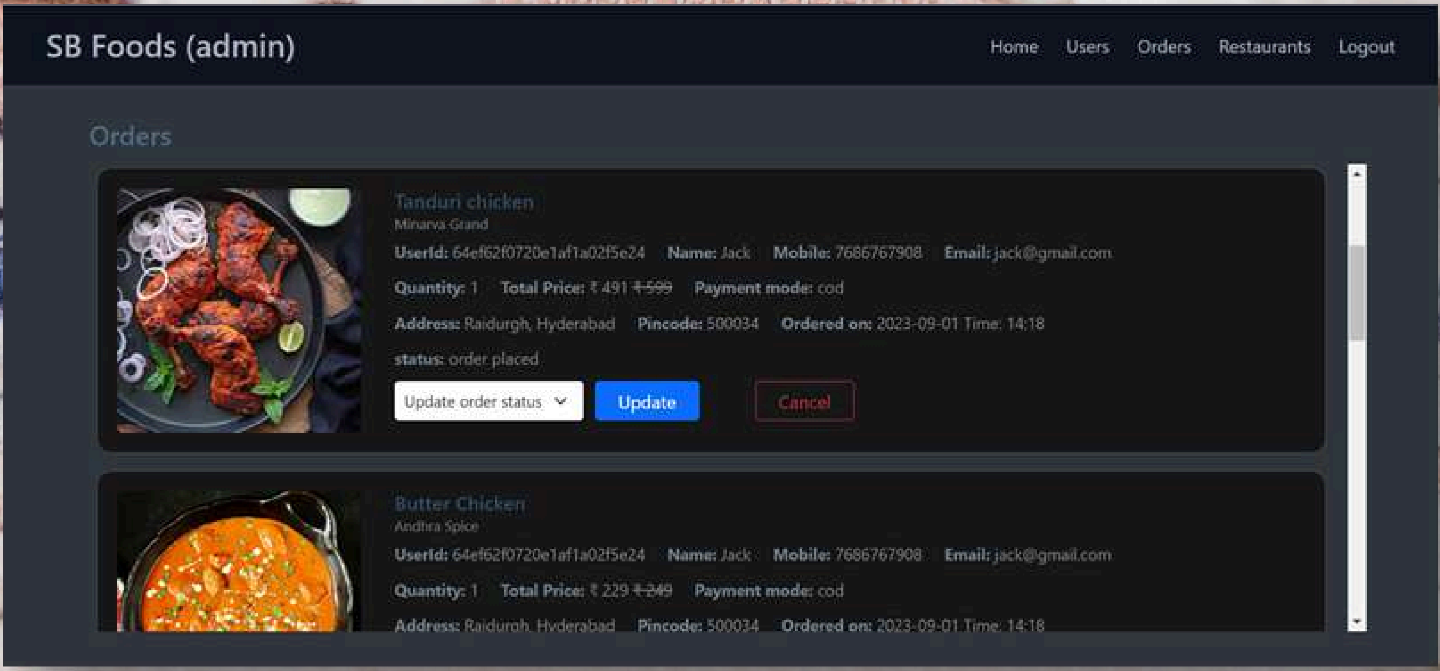
Popular Restaurants(promotions)

- [x] Andhra Spice
- [x] Mc donalds
- [ ] Paradise Grand
- [ ] Minarva Grand

Approvals

No new requests...

All Orders



Orders

Tanduri chicken
Minarva Grand

UserId: 64ef62f0720e1af1a02f5e24 **Name:** Jack **Mobile:** 7686767908 **Email:** jack@gmail.com

Quantity: 1 **Total Price:** ₹ 491 ~~₹ 599~~ **Payment mode:** cod

Address: Raidurgh, Hyderabad **Pincode:** 500034 **Ordered on:** 2023-09-01 Time: 14:18

status: order placed

Butter Chicken
Andhra Spice

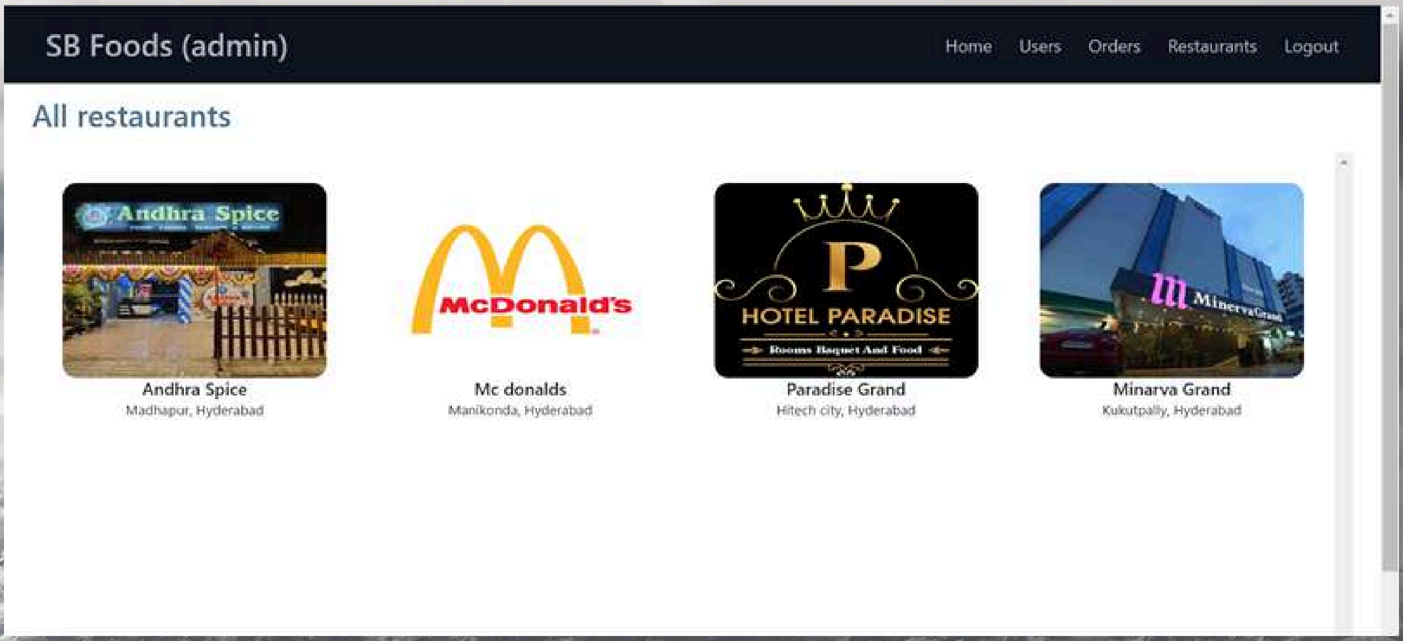
UserId: 64ef62f0720e1af1a02f5e24 **Name:** Jack **Mobile:** 7686767908 **Email:** jack@gmail.com

Quantity: 1 **Total Price:** ₹ 229 ~~₹ 249~~ **Payment mode:** cod

Address: Raidurgh, Hyderabad **Pincode:** 500034 **Ordered on:** 2023-09-01 Time: 14:18

Smart Internz

All restaurants


SB Foods (admin)

Home Users Orders Restaurants Logout

All restaurants

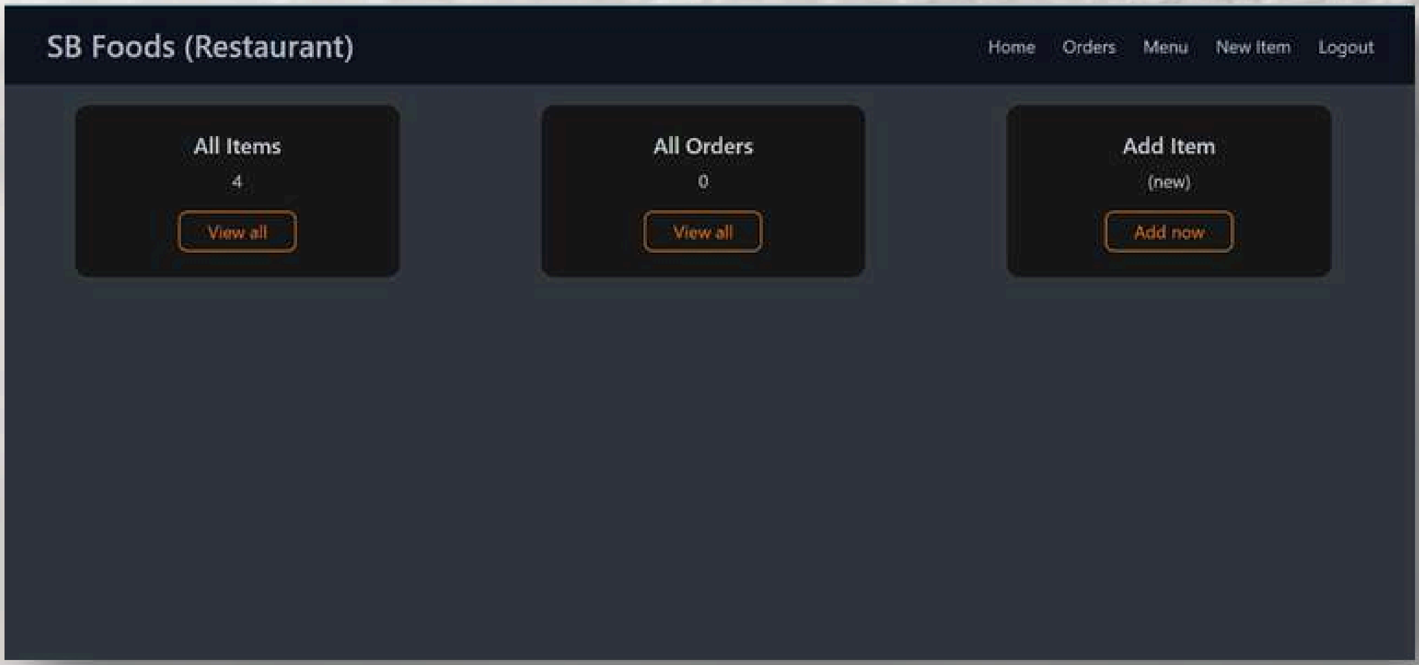
Andhra Spice
Madhapur, Hyderabad

Mc donalds
Manikonda, Hyderabad

Paradise Grand
Hitech city, Hyderabad

Minarva Grand
Kukutpally, Hyderabad

Restaurant Dashboard


SB Foods (Restaurant)

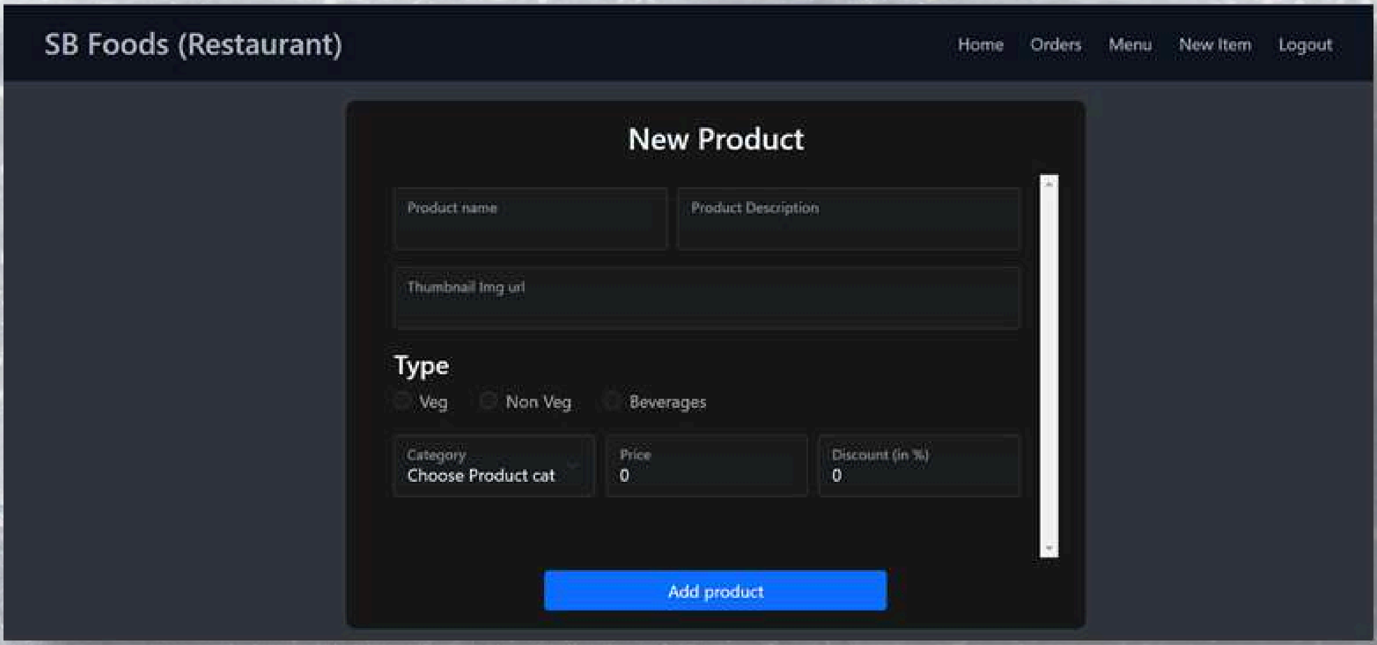
Home Orders Menu New Item Logout

All Items
4
View all

All Orders
0
View all

Add Item
(new)
Add now

New Item



New Product

Product name | Product Description

Thumbnail Img url

Type

Veg Non Veg Beverages

Category: Choose Product cat

Price: 0

Discount (in %): 0

Conclusion

The development of OrderOnTheGo: Your On-Demand Food Ordering Solution has been a highly enriching experience. This project enabled the application of theoretical knowledge in a practical environment, integrating technologies such as React.js, Node.js, Express, and MongoDB to build a dynamic and responsive full-stack application.

The journey involved real-world challenges such as UI/UX optimization, secure API integration, state management, and deployment strategies. Through this project, I have significantly enhanced my technical, problem-solving, and project management skills.

This solution not only simulates the functionality of modern food ordering platforms but also lays the groundwork for future innovations in the domain. It stands as a testament to continuous learning and the potential of web technologies in transforming business operations.

Project Links

GitHub Repository:

https://github.com/LeonardoXmariposa/food-ordering-systemopia0

Demo Video (YouTube):

Smart Internz

https://youtu.be/ntOTdudM9wE?si=uvfd2-zOv7gjryVX

About the Developers team

Team ID : LTVIP2025TMID55809
Team Leader : Sripriya Akula
Team Member : Poojitha Pasupuleti
Team Member : Parasa Sundar Singh (SSP)
Team Member : Sarva Sree Lakshmi Manaswini

Acknowledgment

We would like to express our heartfelt gratitude to our project guide, the faculty members, and the institution for their invaluable support, guidance, and encouragement throughout the duration of this project.

We are especially thankful to SmartInternz and the entire internship coordination team for offering this exceptional opportunity to gain real-world industry experience. The well-structured learning modules, expert mentorship, and continuous support provided by the platform played a pivotal role in shaping our project and enhancing our skills.

Our sincere appreciation also goes to our team members of Team ID: LTVIP2025TMID55809, whose dedication, cooperation, and collaborative spirit were essential to the successful completion of the OrderOnTheGo project. This experience has significantly improved our ability to work as a team, face challenges together, and deliver effective technical solutions.

Smart Internz