

# OrderOnTheGo: Your On-Demand Food Ordering Solution

**Team ID : LTVIP2025TMID55113**

**Team Size : 4**

**Team Leader : Kolla Yasaswini**

**Team member : Kemburu Surendra Kumar**

**Team member : Kodeti Likhitha**

**Team member : Kasula Lakshmi Gnana Deepika**

## INTRODUCTION

Introducing QuickBite, the cutting-edge digital platform poised to revolutionize the way you order food online. With QuickBite, 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 – QuickBite 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 – QuickBite 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.

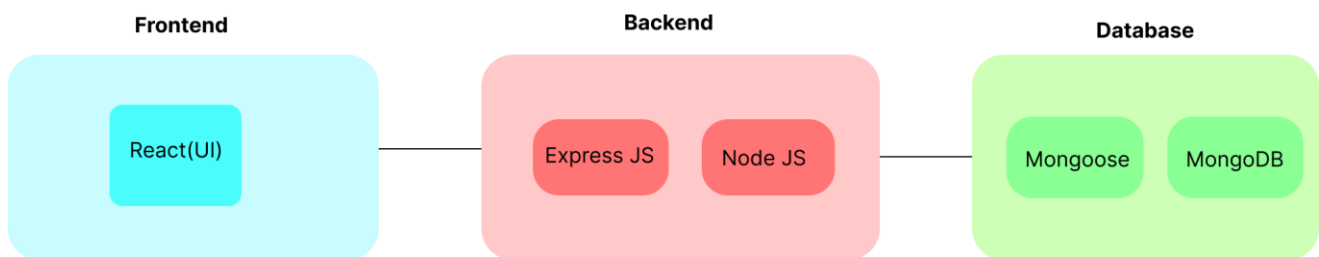
Solution with Food Ordering App:

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.

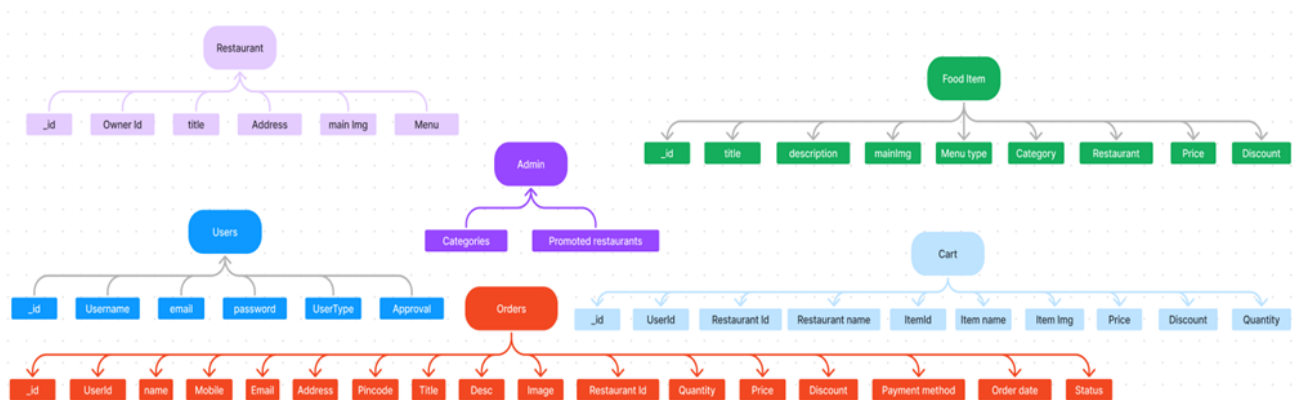
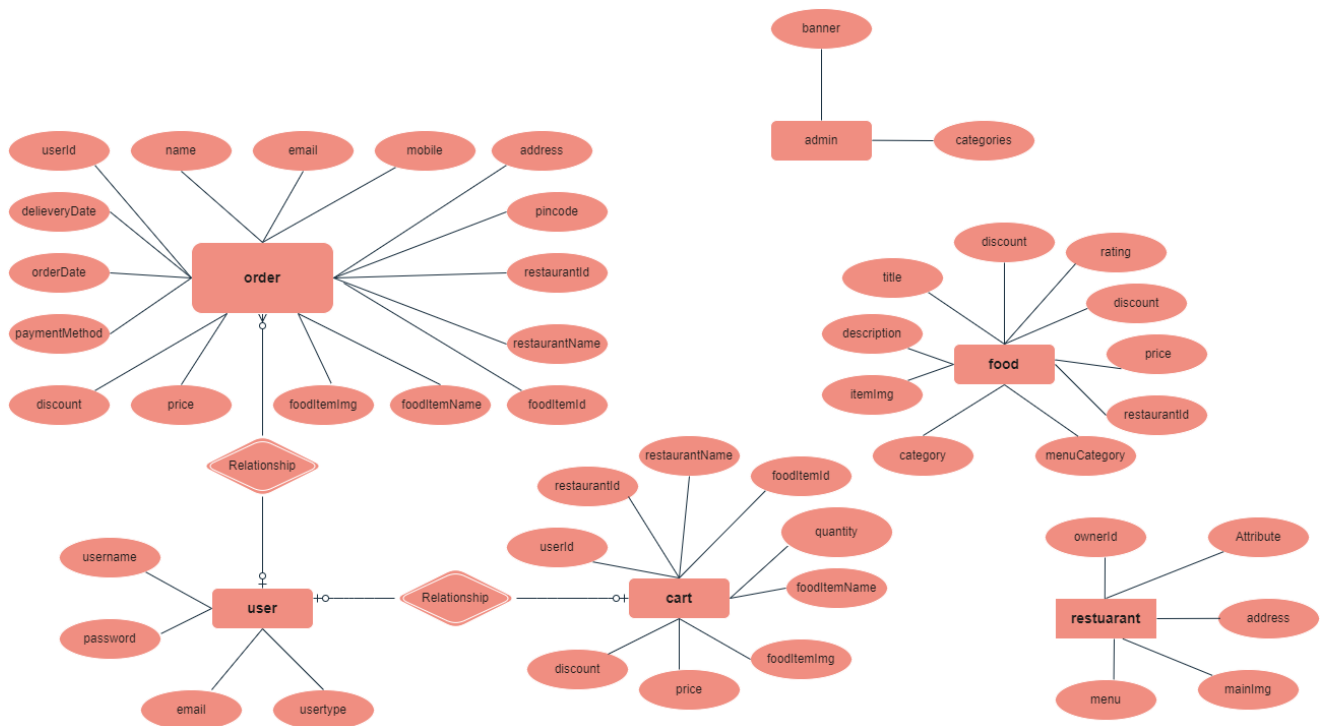
## TECHNICAL ARCHITECTURE:



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 QuickBite 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:** QuickBite 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:** QuickBite 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, QuickBite 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.

QuickBite 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 QuickBite App project downloaded from github:**

Follow below steps:

**Clone the repository:**

- Open your terminal or command prompt.
- Navigate to the directory where you want to store the e-commerce app.

- Execute the following command to clone the repository:

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

### **Dependencies:**

- Navigate into the cloned repository directory:  
**cd Food-Ordering-App-MERN**
- Install the required dependencies by running the following command:  
**npm install**

### **Start the Development Server:**

- To start the development server, execute the following command:  
**npm run dev or npm run start**
- The e-commerce app will be accessible at <http://localhost:3000> by default. You can change the port configuration in the .env file if needed.

### **Access the App:**

- Open your web browser and navigate to <http://localhost:3000>.
- You should see the flight booking app's homepage, indicating that the installation and setup were successful.

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

## **USER & ADMIN FLOW:**

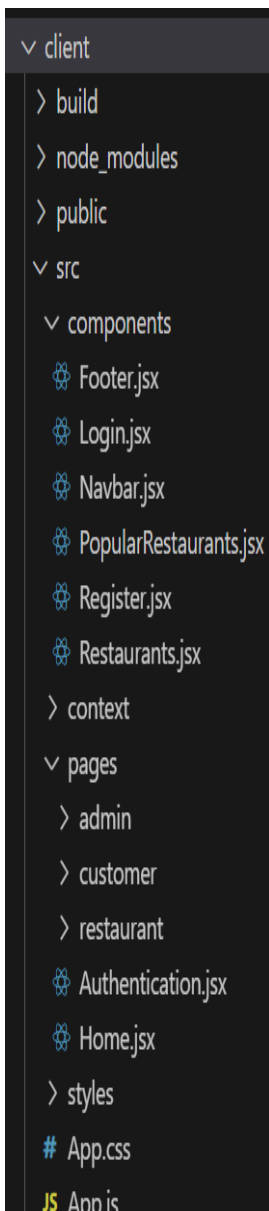
### **1. User Flow:**

- 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:**

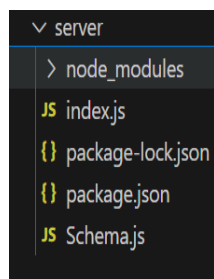
- Restaurants start by authenticating with their credentials.
  - They need to get approval from the admin to start listing the products. •
- They can add/edit the food items.

### **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.

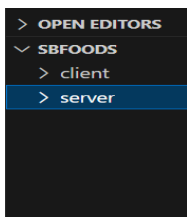
Reference Article: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

**Create project folders and files:**

- Client folders.
- Server folders

Referral Video Link:

[https://drive.google.com/file/d/1uSMbPIAR6rfAEMcb\\_nLZAZd5QIjTpnYQ/view?usp=sharing](https://drive.google.com/file/d/1uSMbPIAR6rfAEMcb_nLZAZd5QIjTpnYQ/view?usp=sharing)



# 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](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:



```

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

```

## 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

JS Schemajs X

server > JS Schemajs > [e] foodItemSchema > ⌘ rating

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

```

JS Schemas X
server > JS Schemas > | @ foodItemSchema > | rating
31 const orderSchema = new mongoose.Schema({
32   // ...
34   name: {type: String},
35   email: {type: String},
36   mobile: {type: String},
37   address: {type: String},
38   pincode: {type: String},
39   restaurantId: {type: String},
40   restaurantName: {type: String},
41   foodItemId: {type: String},
42   foodItemName: {type: String},
43   foodItemImg: {type: String},
44   quantity: {type: Number},
45   price: {type: Number},
46   discount: {type: Number},
47   paymentMethod: {type: String},
48   orderDate: {type: String},
49   orderStatus: {type: String, default: 'order placed'}
50 })
51 const cartSchema = new mongoose.Schema({
52   userId: {type: String},
53   restaurantId: {type: String},
54   restaurantName: {type: String},
55   foodItemId: {type: String},
56   foodItemName: {type: String},
57   foodItemImg: {type: String},
58   quantity: {type: Number},
59   price: {type: Number},
60   discount: {type: Number}
61 })

```

## BACKEND DEVELOPMENT:

### Set Up Project Structure:

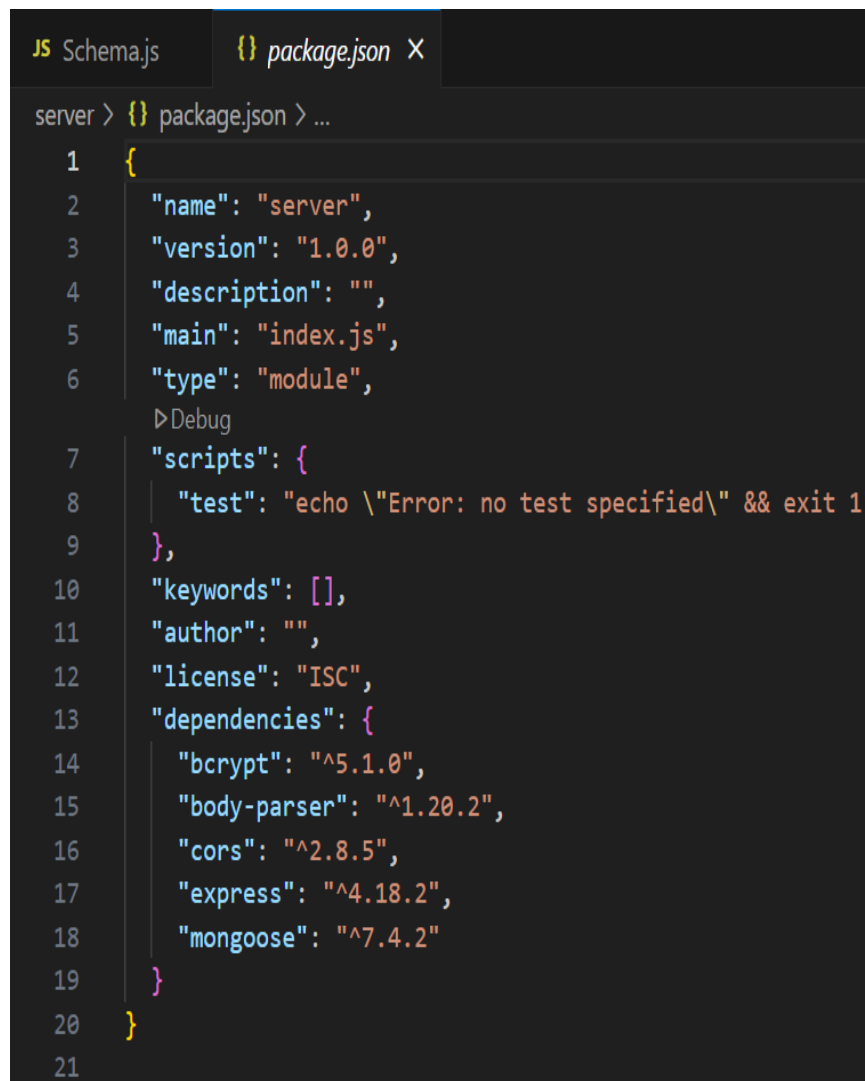
- Create a new directory for your project and set up a package.json file using the npm

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:



The image shows a code editor with two tabs: 'Schema.js' and 'package.json'. The 'package.json' tab is active, displaying the following JSON configuration:

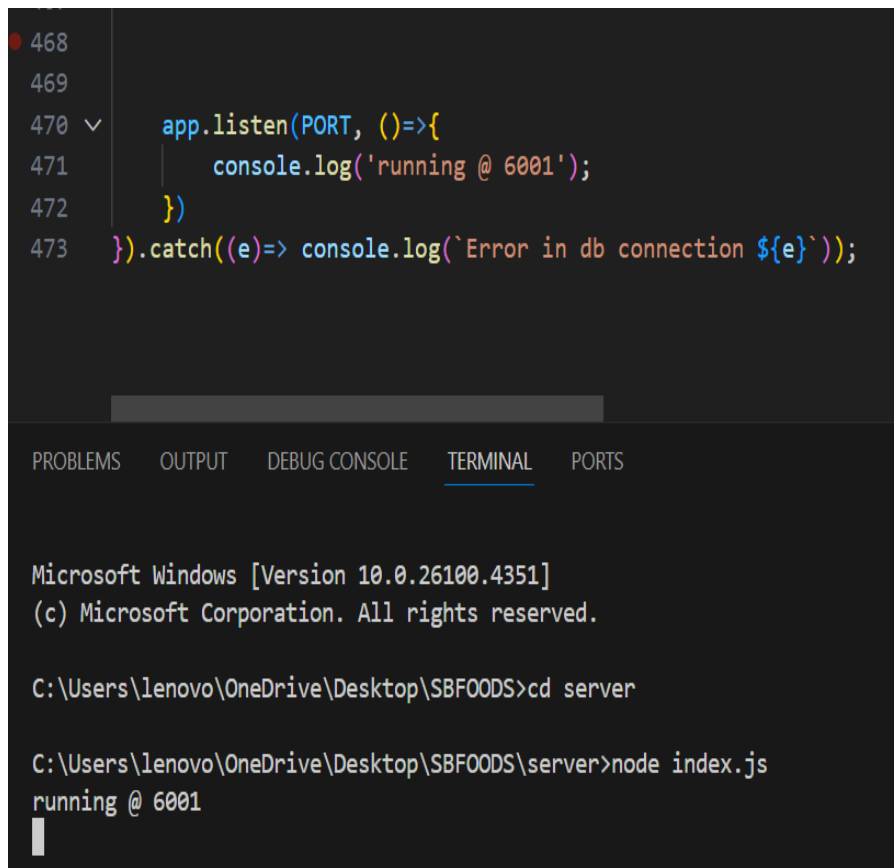
```
server > {} package.json > ...
1  {
2    "name": "server",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "type": "module",
7    "scripts": {
8      "test": "echo \"Error: no test specified\" && exit 1",
9    },
10   "keywords": [],
11   "author": "",
12   "license": "ISC",
13   "dependencies": {
14     "bcrypt": "^5.1.0",
15     "body-parser": "^1.20.2",
16     "cors": "^2.8.5",
17     "express": "^4.18.2",
18     "mongoose": "^7.4.2"
19   }
20 }
21
```

## 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](https://drive.google.com/file/d/1-uKMIcrok_ROHyZl2vRORggrYRio2qXS/view?usp=sharing)

Reference Image:



The image shows a code editor with a dark theme. The code is as follows:

```
468
469
470  ✓  app.listen(PORT, ()=>{
471      console.log('running @ 6001');
472  })
473  }).catch((e)=> console.log(`Error in db connection ${e}`));
```

Below the code editor is a terminal window with the following output:

```
Microsoft Windows [Version 10.0.26100.4351]
(c) Microsoft Corporation. All rights reserved.

C:\Users\lenovo\OneDrive\Desktop\SBFOODS>cd server

C:\Users\lenovo\OneDrive\Desktop\SBFOODS\server>node index.js
running @ 6001
```

## 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](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:

```
468
469
470 ✓ app.listen(PORT, ()=>{
471     console.log('running @ 6001');
472 })
473 }).catch((e)=> console.log(`Error in db connection ${e}`));
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Microsoft Windows [Version 10.0.26100.4351]  
(c) Microsoft Corporation. All rights reserved.

C:\Users\lenovo\OneDrive\Desktop\SBFOODS>cd server

C:\Users\lenovo\OneDrive\Desktop\SBFOODS\server>node index.js

running @ 6001

### 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](https://drive.google.com/file/d/1-uKMIcrok_ROHyZl2vRORggrYRio2qXS/view?usp=sharing)

Reference Image:

```
468
469
470  ✓   app.listen(PORT, ()=>{
471      console.log('running @ 6001');
472  })
473  }).catch((e)=> console.log(`Error in db connection ${e}`));
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Microsoft Windows [Version 10.0.26100.4351]  
(c) Microsoft Corporation. All rights reserved.

C:\Users\lenovo\OneDrive\Desktop\SBFOODS>cd server

C:\Users\lenovo\OneDrive\Desktop\SBFOODS\server>node index.js  
running @ 6001

#### 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.

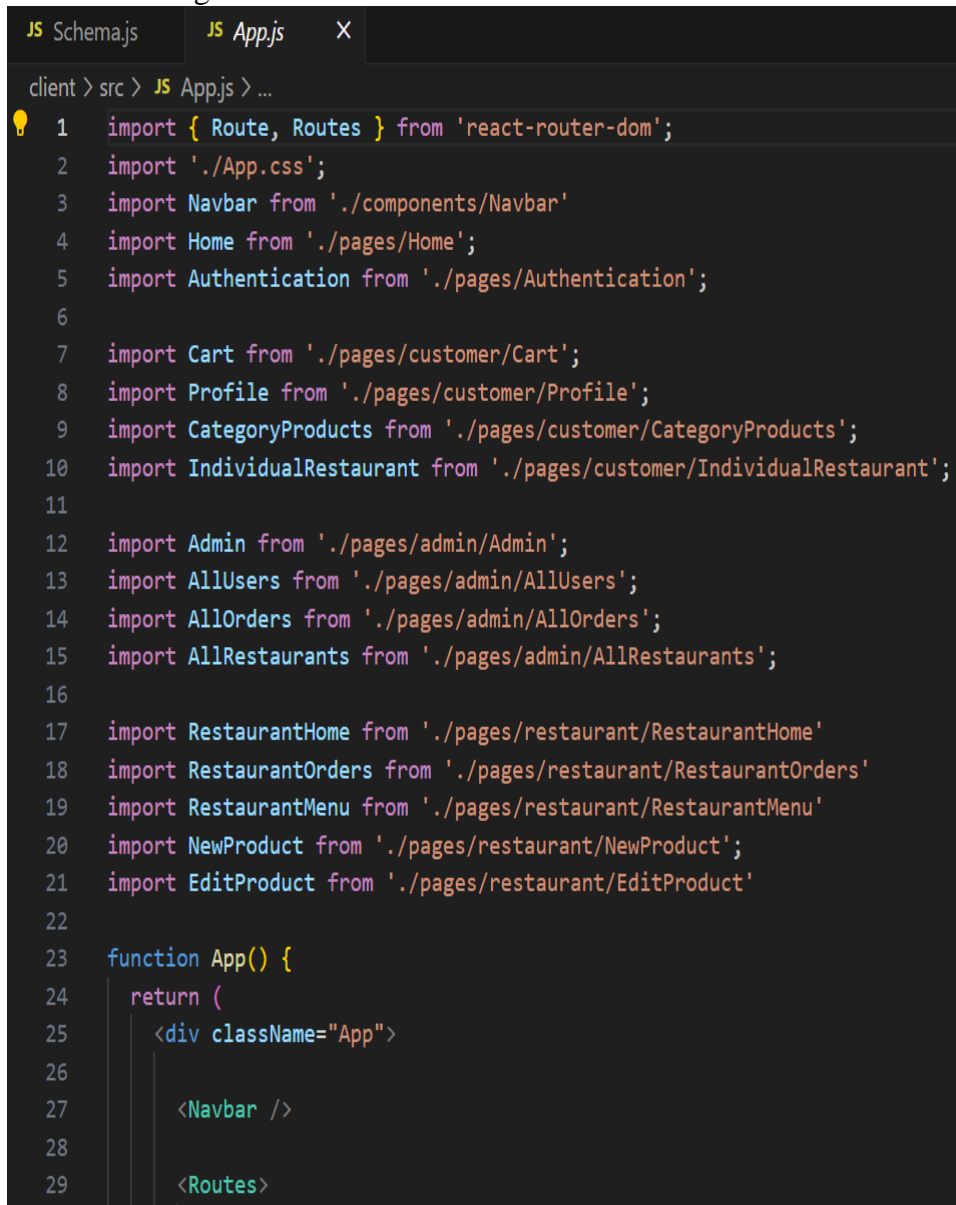
Reference Video Link:

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

Reference Article Link:

[https://www.w3schools.com/react/react\\_getstarted.asp](https://www.w3schools.com/react/react_getstarted.asp)

Reference Image:



```
JS Schema.js JS App.js X
client > src > JS App.js > ...
1 import { Route, Routes } from 'react-router-dom';
2 import './App.css';
3 import Navbar from './components/Navbar';
4 import Home from './pages/Home';
5 import Authentication from './pages/Authentication';
6
7 import Cart from './pages/customer/Cart';
8 import Profile from './pages/customer/Profile';
9 import CategoryProducts from './pages/customer/CategoryProducts';
10 import IndividualRestaurant from './pages/customer/IndividualRestaurant';
11
12 import Admin from './pages/admin/Admin';
13 import AllUsers from './pages/admin/AllUsers';
14 import AllOrders from './pages/admin/AllOrders';
15 import AllRestaurants from './pages/admin/AllRestaurants';
16
17 import RestaurantHome from './pages/restaurant/RestaurantHome';
18 import RestaurantOrders from './pages/restaurant/RestaurantOrders';
19 import RestaurantMenu from './pages/restaurant/RestaurantMenu';
20 import NewProduct from './pages/restaurant/NewProduct';
21 import EditProduct from './pages/restaurant/EditProduct';
22
23 function App() {
24   return (
25     <div className="App">
26
27       <Navbar />
28
29       <Routes>
```

## CODE EXPLANATION

### Server setup:

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

```

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

```

## User Authentication:

- **Backend**

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

### Login

```

61
62
63  app.post('/login', async (req, res) => {
64    const { email, password } = req.body;
65    try {
66
67      const user = await User.findOne({ email });
68
69      if (!user) {
70        return res.status(401).json({ message: 'Invalid email or password' });
71      }
72      const isMatch = await bcrypt.compare(password, user.password);
73      if (!isMatch) {
74        return res.status(401).json({ message: 'Invalid email or password' });
75      } else{
76        return res.json(user);
77      }
78
79    } catch (error) {
80      console.log(error);
81      return res.status(500).json({ message: 'Server Error' });
82    }
83  });

```

```

23 app.post('/register', async (req, res) => {
24   const { username, email, usertype, password, restaurantAddress, restaurantImage } = req.body;
25   try {
26     const existingUser = await User.findOne({ email });
27     if (existingUser) {
28       return res.status(400).json({ message: 'User already exists' });
29     }
30   }
31
32   const hashedPassword = await bcrypt.hash(password, 10);
33
34   if(usertype === 'restaurant'){
35     const newUser = new User({
36       username, email, usertype, password: hashedPassword, approval: 'pending'
37     });
38     const user = await newUser.save();
39     console.log(user._id);
40     const restaurant = new Restaurant({ownerId: user._id ,title: username, address: resta
41     await restaurant.save();
42
43     return res.status(201).json(user);
44   } else{
45
46     const newUser = new User({
47       username, email, usertype, password: hashedPassword, approval: 'approved'
48     });
49

```

## Frontend

Login:

```

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');
        } else if(res.data.usertype === 'restaurant'){
          navigate('/restaurant');
        }
      })
      .catch((err) =>{
        alert("login failed!!");
        console.log(err);
      });
  }catch(err){
    console.log(err);
  }
}

```

Logout:

```

const logout = async () =>{

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

  navigate('/');
}

```

Register:

```

18
19     const fetchRestaurants = useCallback(async () => {
20         try {
21             const response = await axios.get(`http://localhost:6001/fetch-restaurant/${id}`);
22             setRestaurant(response.data);
23         } catch (err) {
24             console.error("Error fetching restaurant:", err);
25         }
26     }, [id]);
27
28     const fetchCategories = useCallback(async () => {
29         try {
30             const response = await axios.get('http://localhost:6001/fetch-categories');
31             setAvailableCategories(response.data);
32         } catch (err) {
33             console.error("Error fetching categories:", err);
34         }
35     }, []);
36
37     const fetchItems = useCallback(async () => {
38         try {
39             const response = await axios.get('http://localhost:6001/fetch-items');
40             setItems(response.data);
41         } catch (err) {
42             console.error("Error fetching items:", err);
43         }

```

## 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:

```

18
19     const fetchRestaurants = useCallback(async () => {
20         try {
21             const response = await axios.get(`http://localhost:6001/fetch-restaurant/${id}`);
22             setRestaurant(response.data);
23         } catch (err) {
24             console.error("Error fetching restaurant:", err);
25         }
26     }, [id]);
27
28     const fetchCategories = useCallback(async () => {
29         try {
30             const response = await axios.get('http://localhost:6001/fetch-categories');
31             setAvailableCategories(response.data);
32         } catch (err) {
33             console.error("Error fetching categories:", err);
34         }
35     }, []);
36
37     const fetchItems = useCallback(async () => {
38         try {
39             const response = await axios.get('http://localhost:6001/fetch-items');
40             setItems(response.data);
41         } catch (err) {
42             console.error("Error fetching items:", err);
43         }

```

Filtering products:

```

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

```

## • Backend

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

```

app.get('/fetch-products', async(req, res)=>{
  try{
    const products = await Product.find();
    res.json(products);
  }catch(err){
    res.status(500).json({ message: 'Error occurred' });
  }
})

```

## Add product to cart:

### • Frontend

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

```

90
91 const handleAddToCart = async (foodItemId, foodItemName, restaurantId, foodItemImg, price, discount) => {
92   if (quantity < 1) {
93     alert('Quantity must be at least 1');
94     return;
95   }
96
97   try {
98     await axios.post('http://localhost:6001/add-to-cart', {
99       userId,
100       foodItemId,
101       foodItemName,
102       restaurantId,
103       foodItemImg,
104       price,
105       discount,
106       quantity,
107     });
108     alert('Item added to cart!');
109     setCartItem(null);
110     setQuantity(1);
111     fetchCartCount();
112   } catch (error) {
113     console.error("Error adding to cart:", error);
114     alert('Failed to add item to cart');
115   }
116 }

```

- **Backend**

Add product to cart:

```

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

```

## Order products:

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

- **Frontend**

```

47
48 // Order from cart
49
50 app.post('/place-cart-order', async(req, res)=>{
51   const {userId, name, mobile, email, address, pincode, paymentMethod, orderDate} = req
52   try{
53
54     const cartItems = await Cart.find({userId});
55     cartItems.map(async (item)=>{
56
57       const newOrder = new Orders({userId, name, email, mobile, address, pincode, p
58       await newOrder.save();
59       await Cart.deleteOne({_id: item._id})
60     })
61     res.json({message: 'Order placed'});
62
63   }catch(err){
64     res.status(500).json({message: "Error occured"});
65   }
66 })
67

```

## · 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.

```

47
48 // Order from cart
49
50 app.post('/place-cart-order', async(req, res)=>{
51   const {userId, name, mobile, email, address, pincode, paymentMethod, orderDate} = req
52   try{
53
54     const cartItems = await Cart.find({userId});
55     cartItems.map(async (item)=>{
56
57       const newOrder = new Orders({userId, name, email, mobile, address, pincode, p
58       await newOrder.save();
59       await Cart.deleteOne({_id: item._id})
60     })
61     res.json({message: 'Order placed'});
62
63   }catch(err){
64     res.status(500).json({message: "Error occured"});
65   }
66 })
67

```

## Add new product:

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

- Frontend:



```

const NewProduct = () => {
  }, [fetchCategories, fetchRestaurant]); // Now, these functions are stable dependencies

const handleNewProduct = async () => {
  if (!restaurant || !restaurant._id) { // Ensure restaurant and its _id are available
    alert('Restaurant data is not loaded. Please try again.');
```

return;

```
  }

  if (!productName || !productDescription || !productMainImg || !productCategory || !productMenuCategory) {
    alert('Please fill in all required product fields.');
```

return;

```
  }

  if (productMenuCategory === 'new category' && !productNewCategory) {
    alert('Please enter a name for the new category.');
```

return;

```
  }

  try {
    await axios.post('http://localhost:6001/add-new-product', {
      restaurantId: restaurant._id,
      productName,
      productDescription,
      (property) productCategory: string
      productCategory,
      // Use productNewCategory if 'new category' is selected, otherwise use productMenuCategory
      productMenuCategory: productMenuCategory === 'new category' ? productNewCategory : productMenuCategory,
      productPrice: Number(productPrice),
    });
  } catch (err) {
    console.log(err);
  }
};

```

## Backend:

```

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

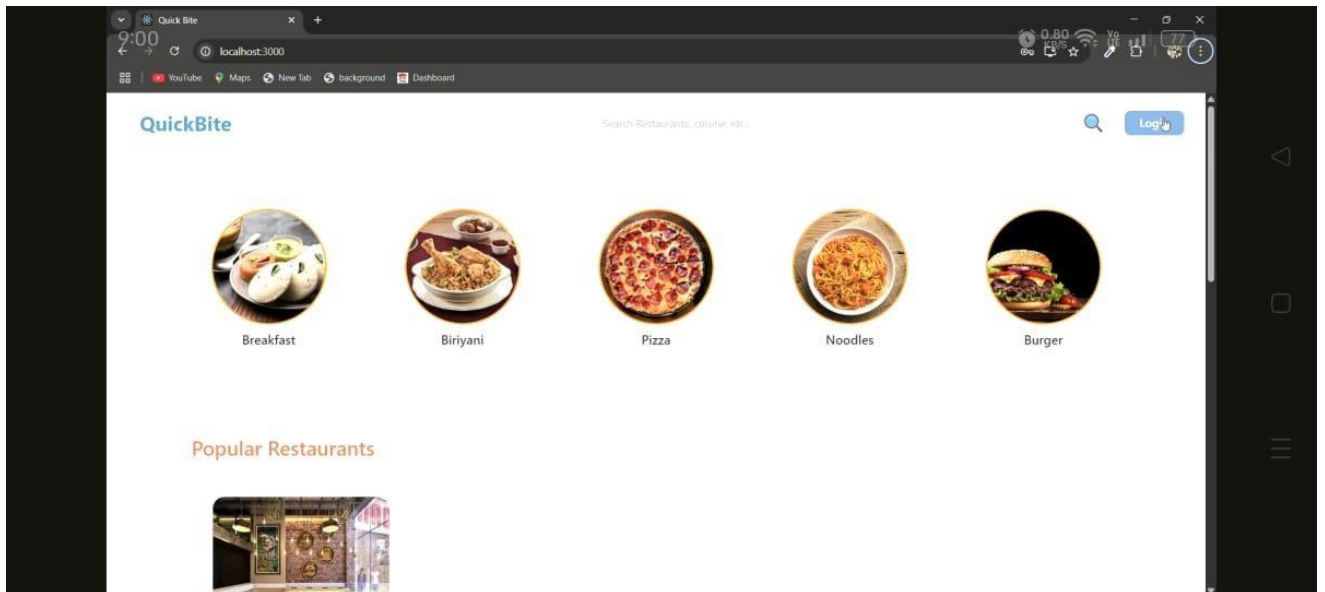
  try{
    if(productMenuCategory === 'new category'){
      const admin = await Admin.findOne();
      admin.categories.push(productNewCategory);
      await admin.save();
      const newProduct = new FoodItem({restaurantId, title: productName, description: productDescription, mainImage: productMainImg, category: productCategory, price: productPrice});
      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, mainImage: productMainImg, category: productCategory, price: productPrice});
      await newProduct.save();
    }
    res.json({message: "product added!!"});
  }catch(err){
    res.status(500).json({message: "Error occured"});
  }
})

```

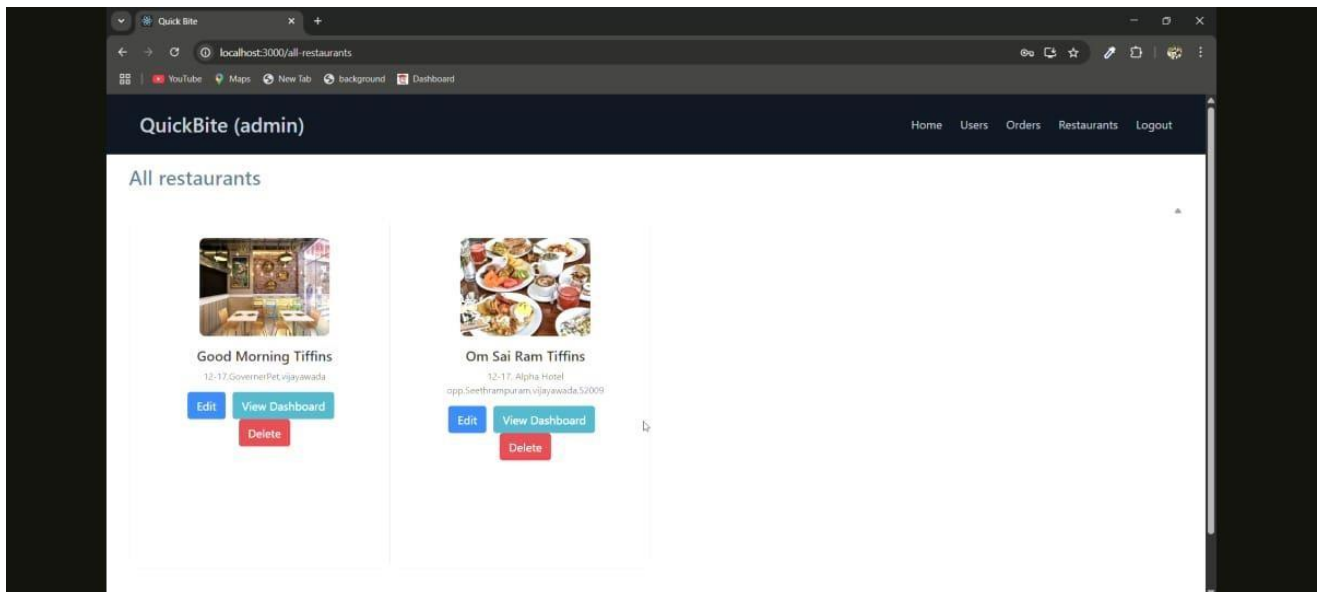
Along with this, implement additional features to view all orders, products, etc., in the admin dashboard.

## Demo UI images:

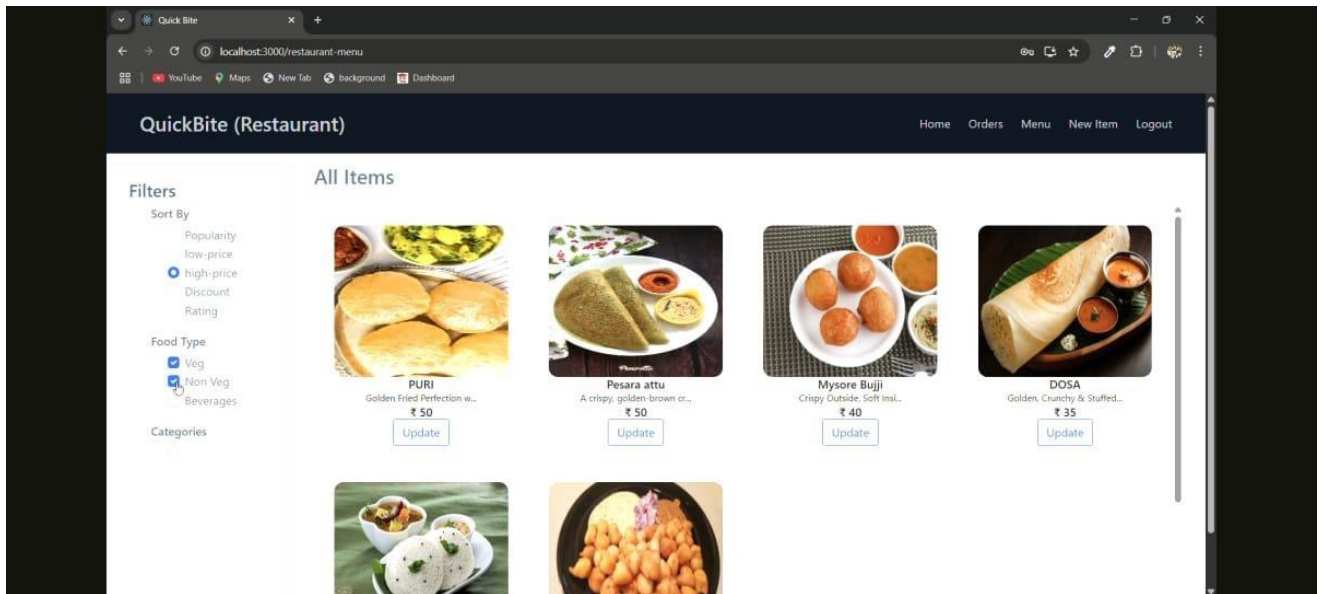
- Landing page



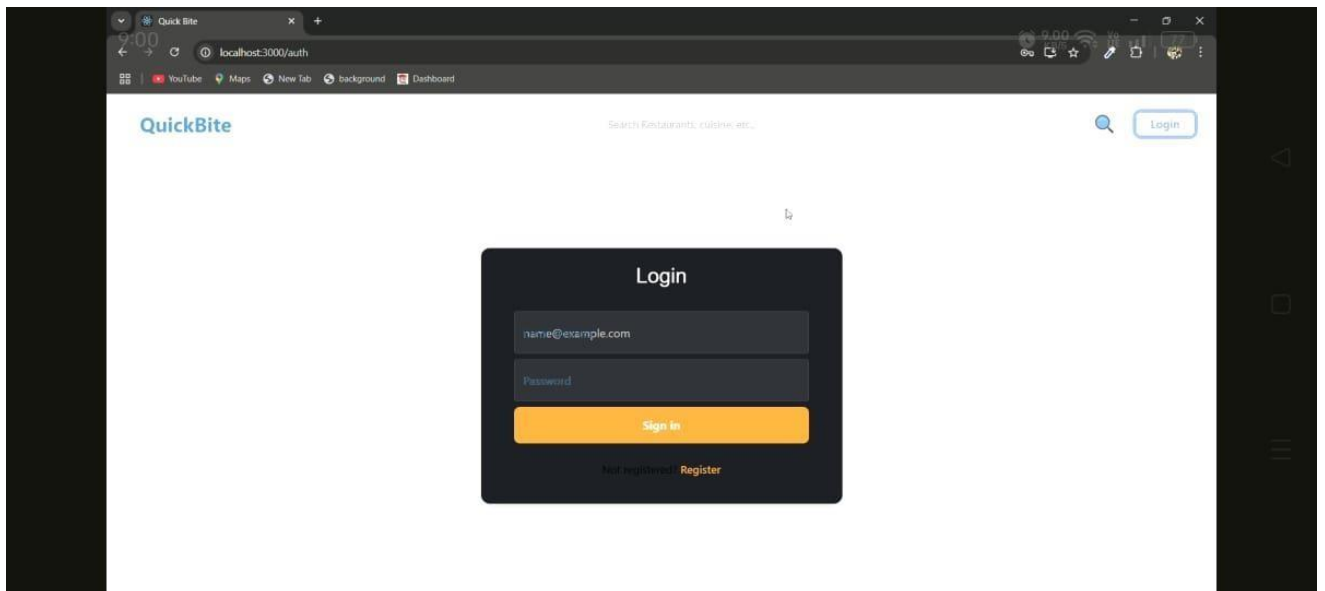
## Restaurants



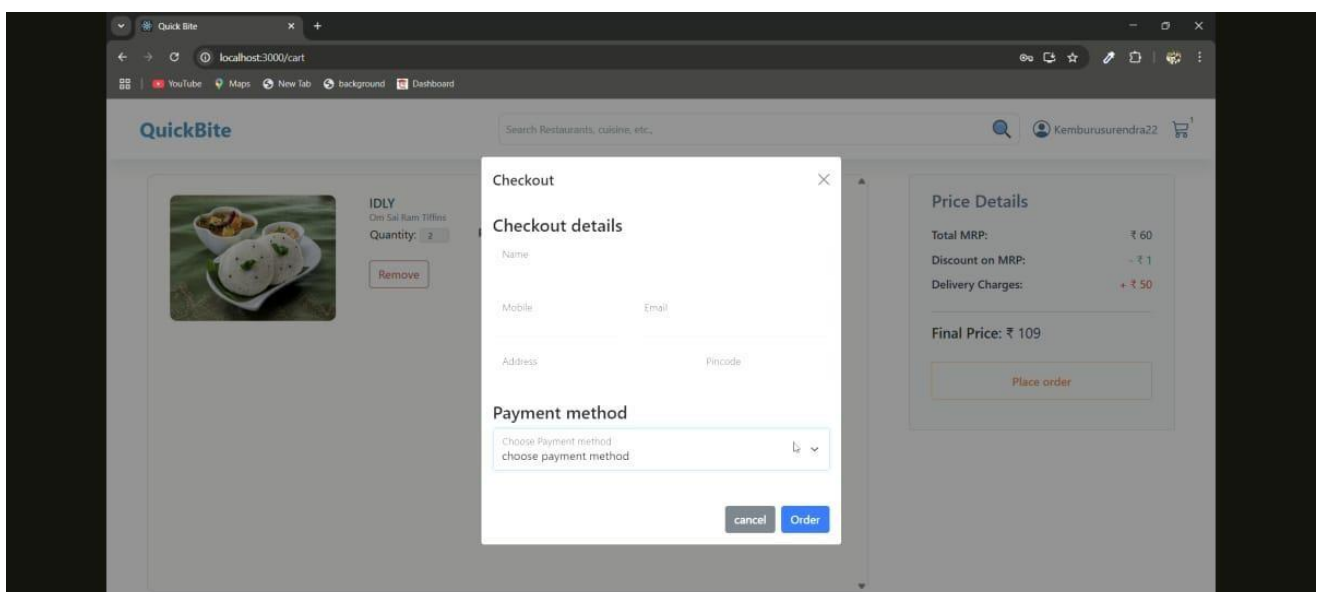
- Restaurant Menu



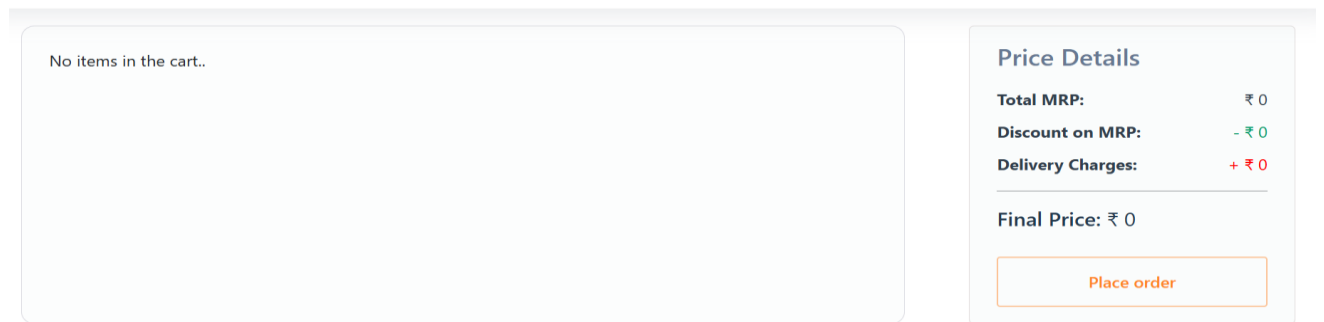
## • Authentication



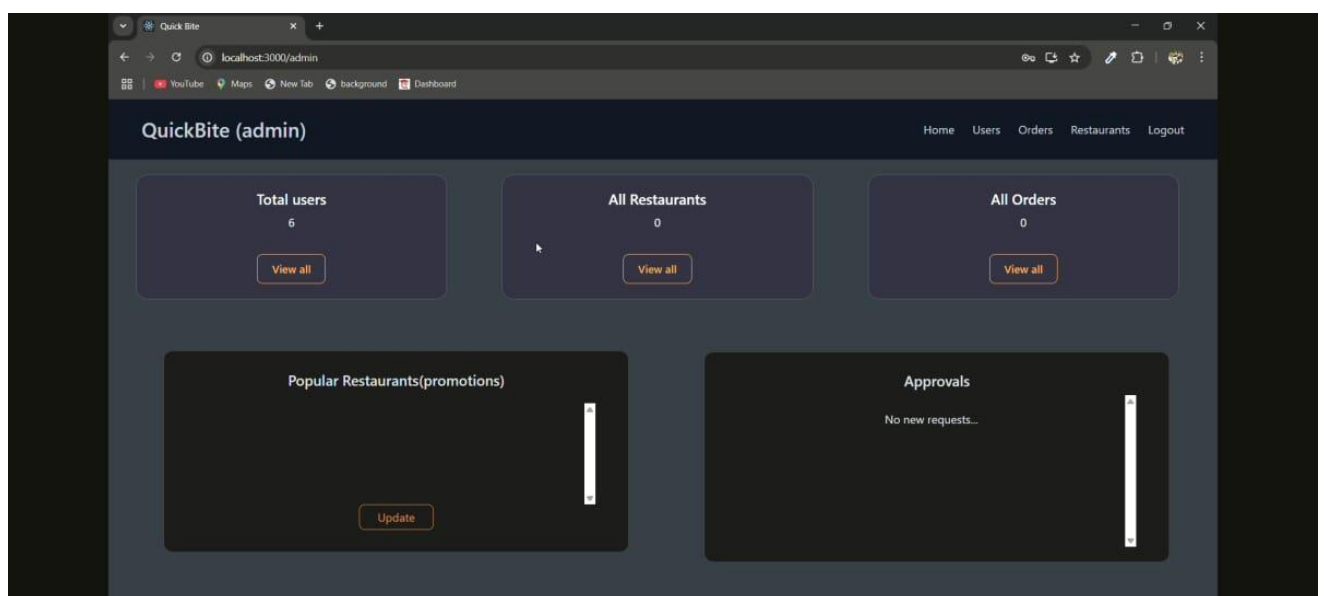
## • User Profile



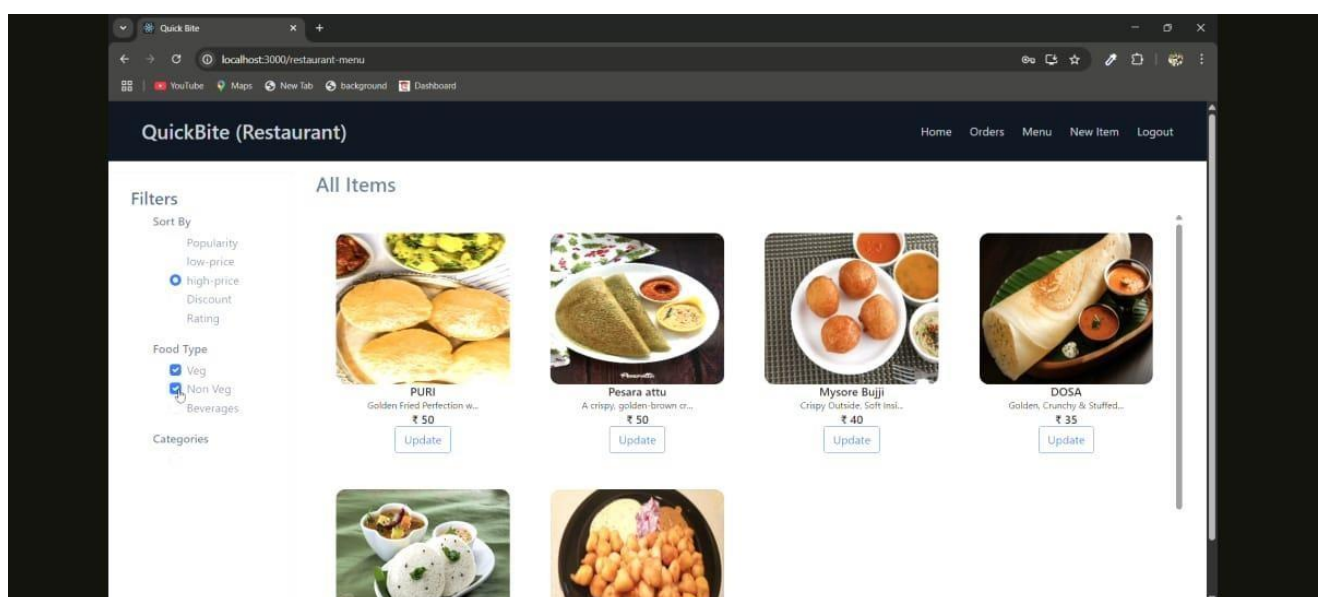
## • Cart



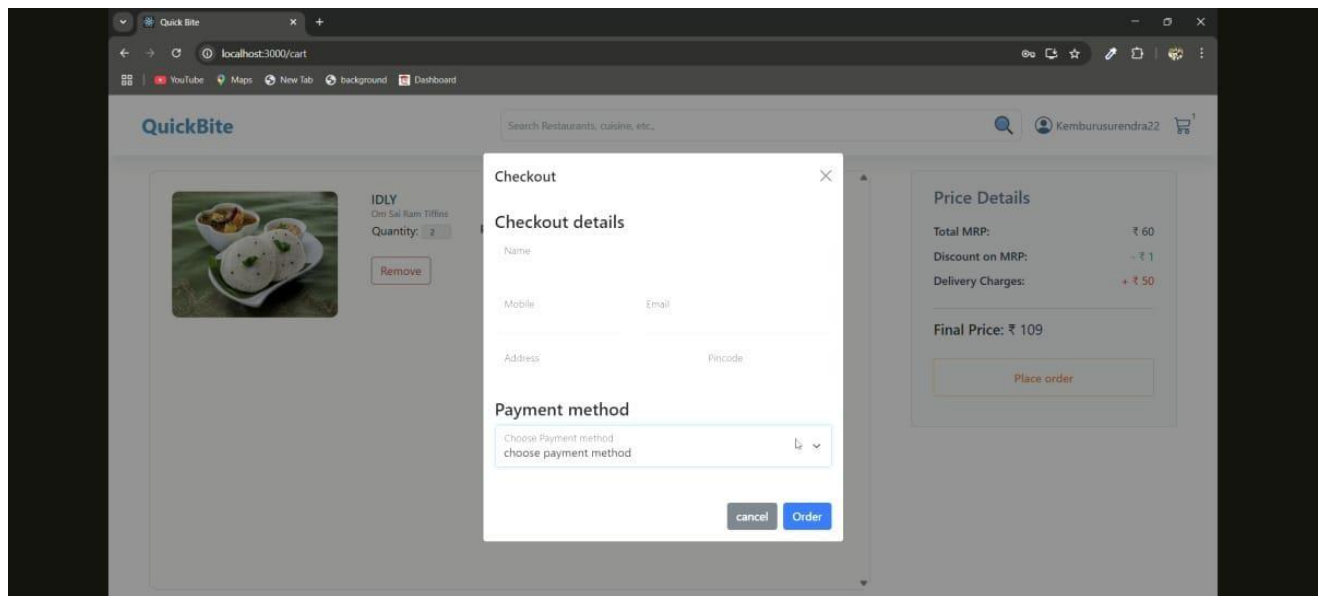
## •Admin dashboard



## • All Orders



## • New Item



The demo of the app is available at:

<https://drive.google.com/file/d/1FWtPXRcN1OmsjP9K5IWtcvb7ofhLoM0n/view?usp=drivesdk>

**\*\* Happy Coding \*\***