

OrderOnTheGo: Your On-Demand Food Ordering Solution

Introduction

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

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

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

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

SCENARIO:

Late-Night Craving Resolution

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

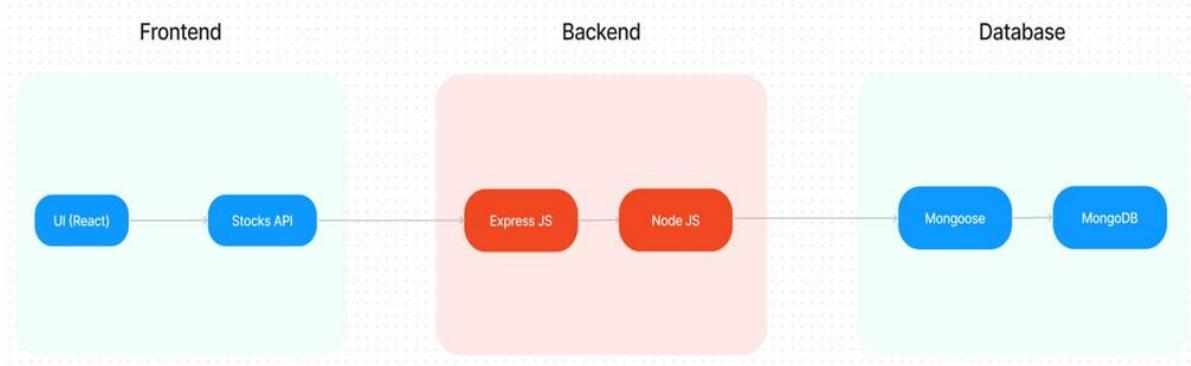
Solution with Food Ordering App:

1. Lisa opens the Food Ordering App on her smartphone and navigates to the late-night delivery section, where she finds a variety of eateries still open for orders.
2. She scrolls through the options, browsing menus and checking reviews until she spots her favorite local diner offering comfort food classics.
3. Lisa selects a hearty bowl of chicken noodle soup and a side of garlic bread, craving warmth and satisfaction in each bite.
4. With a few taps, she adds the items to her cart, specifies her delivery address, and chooses her preferred payment method.
5. Lisa double-checks her order details on the confirmation page, ensuring everything looks correct, before tapping the "Place Order" button.
6. Within minutes, she receives a notification confirming her order and estimated delivery time, allowing her to continue working with peace of mind.

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

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

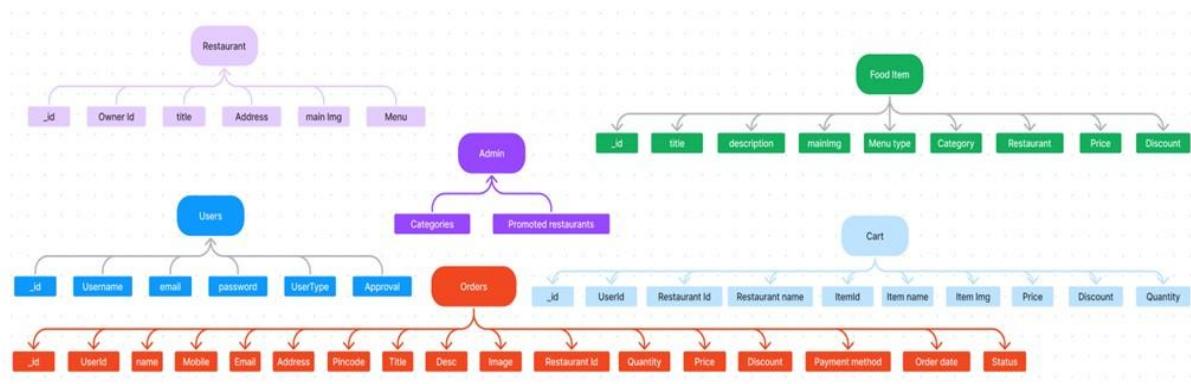
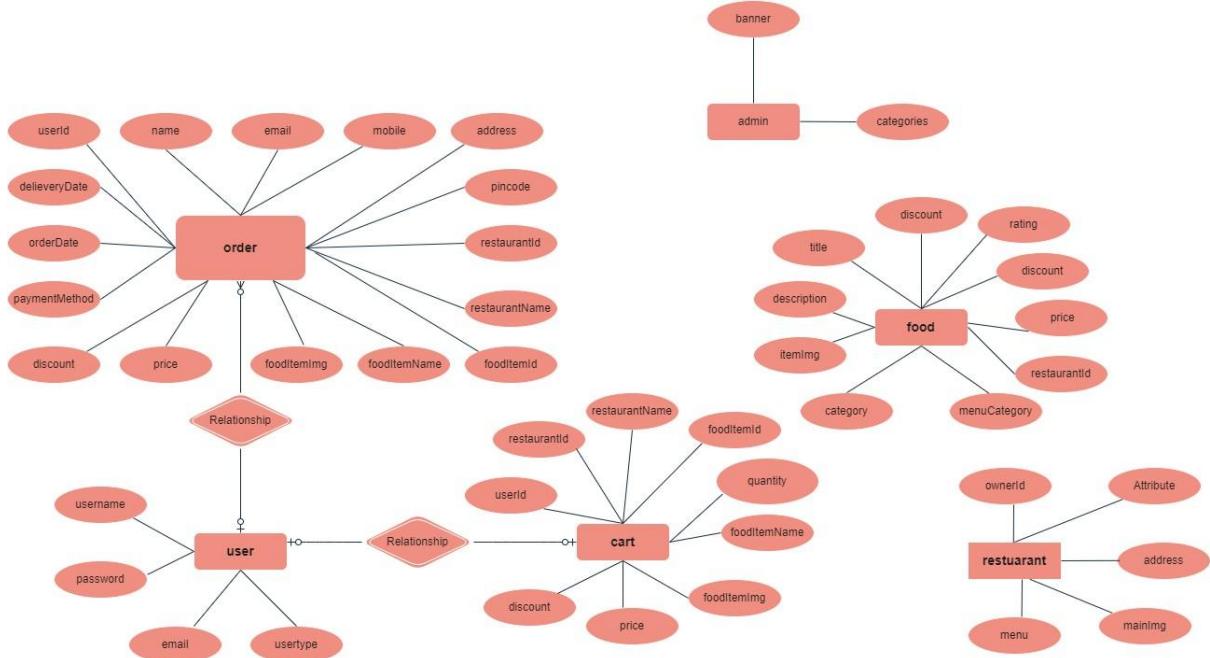
TECHNICAL ARCHITECTURE:



In this architecture diagram:

- The frontend is represented by the "Frontend" section, including user interface components such as User Authentication, Cart, Products, Profile, Admin dashboard, etc.,
- The backend is represented by the "Backend" section, consisting of API endpoints for Users, Orders, Products, etc., It also includes Admin Authentication and an Admin Dashboard.
- The Database section represents the database that stores collections for Users, Admin, Cart, Orders, and products.

ER-Diagram



The SB Foods ER-diagram represents the entities and relationships involved in an food ordering ecommerce 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:

- Comprehensive Product Catalog:** SB Foods boasts an extensive catalog of food items from various restaurants, offering a diverse range of items and options for shoppers. You can effortlessly

explore and discover various products, complete with detailed descriptions, customer reviews, pricing, and available discounts, to find the perfect food for your hunger.

2. **Order Details Page:** Upon clicking the "Shop Now" button, you will be directed to an order details page. Here, you can provide relevant information such as your shipping address, preferred payment method, and any specific product requirements.
3. **Secure and Efficient Checkout Process:** SB Foods guarantees a secure and efficient checkout process. Your personal information will be handled with the utmost security, and we strive to make the purchasing process as swift and trouble-free as possible.
4. **Order Confirmation and Details:** After successfully placing an order, you will receive a confirmation notification. Subsequently, you will be directed to an order details page, where you can review all pertinent information about your order, including shipping details, payment method, and any specific product requests you specified.

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

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

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

React.js: React.js is a popular JavaScript library for building user interfaces. It enables developers to create interactive and reusable UI components, making it easier to build dynamic and responsive web

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

HTML, CSS, and JavaScript: Basic knowledge of HTML for creating the structure of your app, CSS for styling, and JavaScript for client-side interactivity is essential.

Database Connectivity: Use a MongoDB driver or an Object-Document Mapping (ODM) library like Mongoose to connect your Node.js server with the MongoDB database and perform CRUD (Create, Read, Update, Delete) operations.

Front-end Framework: Utilize Angular to build the user-facing part of the application, including product listings, booking forms, and user interfaces for the admin dashboard.

Version Control: Use Git for version control, enabling collaboration and tracking changes throughout the development process. Platforms like GitHub or Bitbucket can host your repository.

- Git: Download and installation instructions can be found at: <https://git-scm.com/downloads>

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

- Visual Studio Code: Download from <https://code.visualstudio.com/download>
- Sublime Text: Download from <https://www.sublimetext.com/download>
- WebStorm: Download from <https://www.jetbrains.com/webstorm/download>

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

<https://www.section.io/engineering-education/nodejs-mongoosejs-mongodb/>

To run the existing SB Foods App project downloaded from github:

Follow below steps:

Clone the repository:

- Open your terminal or command prompt.
- Navigate to the directory where you want to store the e-commerce app.
- Execute the following command to clone the repository:

Git clone: <https://github.com/ramakrishna-50/OrderOnTheGo-Your-On-Demand-Food-Ordering-Solution/upload/main/Document>

- Navigate into the cloned repository directory:

cd Food-Ordering-App-MERN

- Install the required dependencies by running the following command:

npm install

Start the Development Server:

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

npm run dev or npm run start

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

Access the App:

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

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

Application Flow

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	server
> node_modules	
> public	
< src	
> components	
> context	
< pages	
> admin	
> restaurant	
Cart.jsx	
Home.jsx	
Login.jsx	
Orders.jsx	
ProductDetail.jsx	
ProductList.jsx	
Profile.jsx	
Register.jsx	
> styles	
# App.css	
App.jsx	
# index.css	
main.jsx	
# .gitignore	
# eslint.config.js	
# index.html	
{} package-lock.json	
{} package.json	
# README.md	
	< controllers
	JS adminController.js
	JS cartController.js
	JS orderController.js
	JS productController.js
	JS restaurantAuthController.js
	JS restaurantController.js
	JS userAuthController.js
	< models
	JS Admin.js
	JS Cart.js
	JS Order.js
	JS Product.js
	JS Restaurant.js
	JS User.js
	> node_modules
	< routes
	JS adminRoutes.js
	JS authRoutes.js
	JS cartRoutes.js
	JS orderRoutes.js
	JS productRoutes.js
	JS restaurantRoutes.js
	# .gitignore
	JS index.js
	{} package-lock.json
	{} package.json

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 Flow

Let's start with the project development with the help of the given activities.

1. 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

2.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:

The screenshot shows a code editor interface with two main sections: an Explorer sidebar on the left and a code editor on the right.

Explorer Sidebar:

- > SEARCH
- < ORDERONTHEGO-MERN-STACK-INTERNSHIP-P...
- < code
 - < server
 - < controllers
 - JS** restaurantController.js
 - JS** userAuthController.js
 - < models
 - JS** Admin.js
 - JS** Cart.js
 - JS** Order.js
 - JS** Product.js
 - JS** Restaurant.js
 - JS** User.js
 - > node_modules
 - < routes
 - JS** adminRoutes.js
 - JS** authRoutes.js
 - JS** cartRoutes.js
 - JS** orderRoutes.js
 - JS** productRoutes.js
 - JS** restaurantRoutes.js
 - ◆ .gitignore
 - JS** index.js
 - { package-lock.json
 - { package.json
 - > documentation
 - Word document: ~\$derOnTheGo.docx
 - Word document: FSD Final Documentation.docx
 - Word document: OrderOnTheGo.docx

```

JS index.js  X
code > server > JS index.js > ...
1  < import express from "express";
2  import cors from "cors";
3  import mongoose from "mongoose";
4  import authRoutes from "./routes/authRoutes.js";
5  import productRoutes from "./routes/productRoutes.js";
6  import restaurantRoutes from "./routes/restaurantRoutes.js";
7  import adminRoutes from "./routes/adminRoutes.js";
8  import orderRoutes from './routes/orderRoutes.js';
9  import cartRoutes from './routes/cartRoutes.js';

10
11
12 const app = new express();
13 app.use(express.json())
14 app.use(cors());
15
16 mongoose.connect("mongodb://localhost:27017/SBfoods").
17 then().
18 catch((err)=> console.log("error connecting mongoDb"));
19
20 app.use("/api/auth",authRoutes);
21 app.use('/api/products',productRoutes);
22 app.use('/api/restaurants',restaurantRoutes);
23 app.use('/api/admin',adminRoutes);
24 app.use('/api/orders/',orderRoutes);
25 app.use('/api/cart/',cartRoutes);

26
27 < app.get('/',(req,res)=>{
28   | res.send("Hello");
29 });
30 < app.listen(5000,()=>{
31   | console.log("Server is running")
32 });

```

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

3. 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 Image:

```
JS Admin.js X
code > server > models > JS Admin.js > ...
1 import mongoose from "mongoose";
2
3 const adminSchema = new mongoose.Schema({
4   uname: {type:String},
5   password : {type : String},
6   promotedRestaurants : [{type : string}]
7 })
8
9 export default mongoose.model("Admin",adminSchema);
```

```
JS Cart.js X
code > server > models > JS Cart.js > ...
1 import mongoose from "mongoose";
2
3 const cartSchema = new mongoose.Schema({
4   userId : {type : String},
5   restaurantId : {type : String},
6   items : [
7     {
8       productId : {type : String},
9       quantity : {type : Number}
10    }
11  ],
12   totalPrice : {type : Number}
13 })
14
15 export default mongoose.model("Cart",cartSchema);
```

```
JS Order.js X
code > server > models > JS Order.js > ...
D:\OrderOnTheGo-mern-stack-internship-project\code
1
2 const orderSchema = new mongoose.Schema({
3   userId : {type : String},
4   items: [
5     {
6       productId: { type: string },
7       quantity: { type: Number, default: 1 },
8     }
9   ],
10  totalPrice: { type: Number },
11  status: { type: String, default: 'pending' },
12  discount:{type: Number , default:0},
13  finalPrice: {type: Number},
14  createdAt: { type: string }
15 });
16
17
18 export default mongoose.model('Order', orderSchema);
19
```

```
JS Restaurant.js X
code > server > models > JS Restaurant.js > ...
1 import mongoose from "mongoose";
2
3 const restaurantSchema = new mongoose.Schema({
4   restaurantName : {type : String},
5   ownerId : {type : String},
6   address : {type : String},
7   imageUrl : {type : String},
8   rating : {type : Number},
9   totalRatings : {type : Number}
10 });
11
12 export default mongoose.model("Restaurant",restaurantSchema);
```

```
JS Product.js X
code > server > models > JS Product.js > ...
1 import mongoose from "mongoose";
2
3 const productSchema = new mongoose.Schema({
4   productName : {type : String},
5   description : {type: String},
6   restaurantid : {type : String},
7   price : {type: Number},
8   discount:{type:Number},
9   category : {type: String},
10  imageUrl : {type : String},
11  rating: {type : Number},
12  totalRatings : {type: Number}
13 });
14
15 export default mongoose.model("Product",productSchema);
```

OnTheGo-mern-stack-internship-project\code\server\node_modules

```
JS User.js X
code > server > models > JS User.js > ...
1 import mongoose from "mongoose";
2
3 const userSchema = new mongoose.Schema({
4   userName:{type : String},
5   password:{type : String},
6   email:{type : String},
7   userType:{type : String},
8 });
9
10 export default mongoose.model("Users",userSchema);
```

```
{} package.json 1 X  
code > server > {} package.json > ...  
1  {  
2    "name": "server",  
3    "version": "1.0.0",  
4    "main": "index.js",  
5    "type": "module",  
6    "scripts": {  
7      "test": "echo \\\"Error: no test specified\\\" && exit 1",  
8      "start": "nodemon index.js"  
9    },  
10   "keywords": [],  
11   "author": "Vakada Srinu",  
12   "license": "ISC",  
13   "description": "",  
14   "dependencies": {  
15     "bcryptjs": "^3.0.2",  
16     "cors": "^2.8.5",  
17     "express": "^5.1.0",  
18     "jsonwebtoken": "^9.0.2",  
19     "mongoose": "^8.16.0",  
20     "nodemon": "^3.1.10"  
21   }  
22 }  
23
```

1. Setup express server:

- Create index.js file.
- Create an express server on your desired port number.
- Define API's

Reference Image:

The screenshot shows the Visual Studio Code interface. The left side displays the file `index.js` with code for setting up an Express server and connecting to a MongoDB database. The right side shows a terminal window with the output of running the application.

```

JS index.js    X
code > server > JS index.js > ...
1 import express from "express";
2 import cors from "cors";
3 import mongoose from "mongoose";
4 import authRoutes from "./routes/authRoutes.js";
5 import productRoutes from "./routes/productRoutes.js";
6 import restaurantRoutes from "./routes/restaurantRoutes.js";
7 import adminRoutes from "./routes/adminRoutes.js";
8 import orderRoutes from './routes/orderRoutes.js';
9 import cartRoutes from './routes/cartRoutes.js';
10
11
12 const app = new express();
13 app.use(express.json())
14 app.use(cors());
15
16 mongoose.connect("mongodb://localhost:27017/SBfoods").
17 then().
18 catch((err)=> console (alias) const authRoutes: Router
19 import authRoutes
20 app.use("/api/auth",authRoutes);
21 app.use('/api/products',productRoutes);
22 app.use('/api/restaurants',restaurantRoutes);

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

D:\OrderOnTheGo-mern-stack-internship-project\code\server>npm start

```

> server@1.0.0 start
> nodemon index.js

[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Server is runnnning

```

2. Database Configuration:

- Set up a MongoDB database either locally or using a cloud-based MongoDB service like MongoDB Atlas or use locally with MongoDB compass.
- Create a database and define the necessary collections for admin, users, restaurants, food products, orders, and other relevant data.

Reference Video of connect node with mongoDB database:

https://drive.google.com/file/d/1cTS3_EOAAvDctkibG5zVikrTdmoY2Ag/view?usp=sharing

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

Reference Image:

The image shows a code editor interface with a terminal window below it. The code editor has a file named 'index.js' open, which contains the following code:

```
JS index.js  ×
code > server > JS index.js > ...
1 import express from "express";
2 import cors from "cors";
3 import mongoose from "mongoose";
4 import authRoutes from "./routes/authRoutes.js";
5 import productRoutes from "./routes/productRoutes.js";
6 import restaurantRoutes from "./routes/restaurantRoutes.js";
7 import adminRoutes from "./routes/adminRoutes.js";
8 import orderRoutes from './routes/orderRoutes.js';
9 import cartRoutes from './routes/cartRoutes.js';
10
11
12 const app = new express();
13 app.use(express.json())
14 app.use(cors());
15
16 mongoose.connect("mongodb://localhost:27017/SBfoods").
17 then(console.log("Database connected successfully"))
18 catch((err)=> console.log("error connecting mongodb"));
19
20 app.use("/api/auth",authRoutes);
21 app.use('/api/products',productRoutes);
22 app.use('/api/restaurants',restaurantRoutes);
```

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

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

> nodemon index.js

[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Server is running
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Database connected successfully
Server is running
[]
```

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 crossorigin requests.

Reference Video:

https://drive.google.com/file/d/1uKMIcrok_ROHyZl2vRORggrYRlo2qXS/view?usp=sharing

Reference Image:

The screenshot shows a Visual Studio Code interface. The left side displays the file `index.js` with line numbers and code. The right side shows a terminal window with command-line output.

File Content (`index.js`):

```
1 import express from "express";
2 import cors from "cors";
3 import mongoose from "mongoose";
4 import authRoutes from "./routes/authRoutes.js";
5 import productRoutes from "./routes/productRoutes.js";
6 import restaurantRoutes from "./routes/restaurantRoutes.js";
7 import adminRoutes from "./routes/adminRoutes.js";
8 import orderRoutes from './routes/orderRoutes.js';
9 import cartRoutes from './routes/cartRoutes.js';
10
11
12 const app = new express();
13 app.use(express.json())
14 app.use(cors());
15
16 mongoose.connect("mongodb://localhost:27017/SBfoods").
17 then().
18 catch((err)=> console [alias] const authRoutes: Router ;
19 import authRoutes
20 app.use("/api/auth",authRoutes);
21 app.use('/api/products',productRoutes);
22 app.use('/api/restaurants',restaurantRoutes);
```

Terminal Output:

```
D:\OrderOnTheGo-mern-stack-internship-project\code\server>npm start

> server@1.0.0 start
> nodemon index.js

[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Server is running
```

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.

4.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:

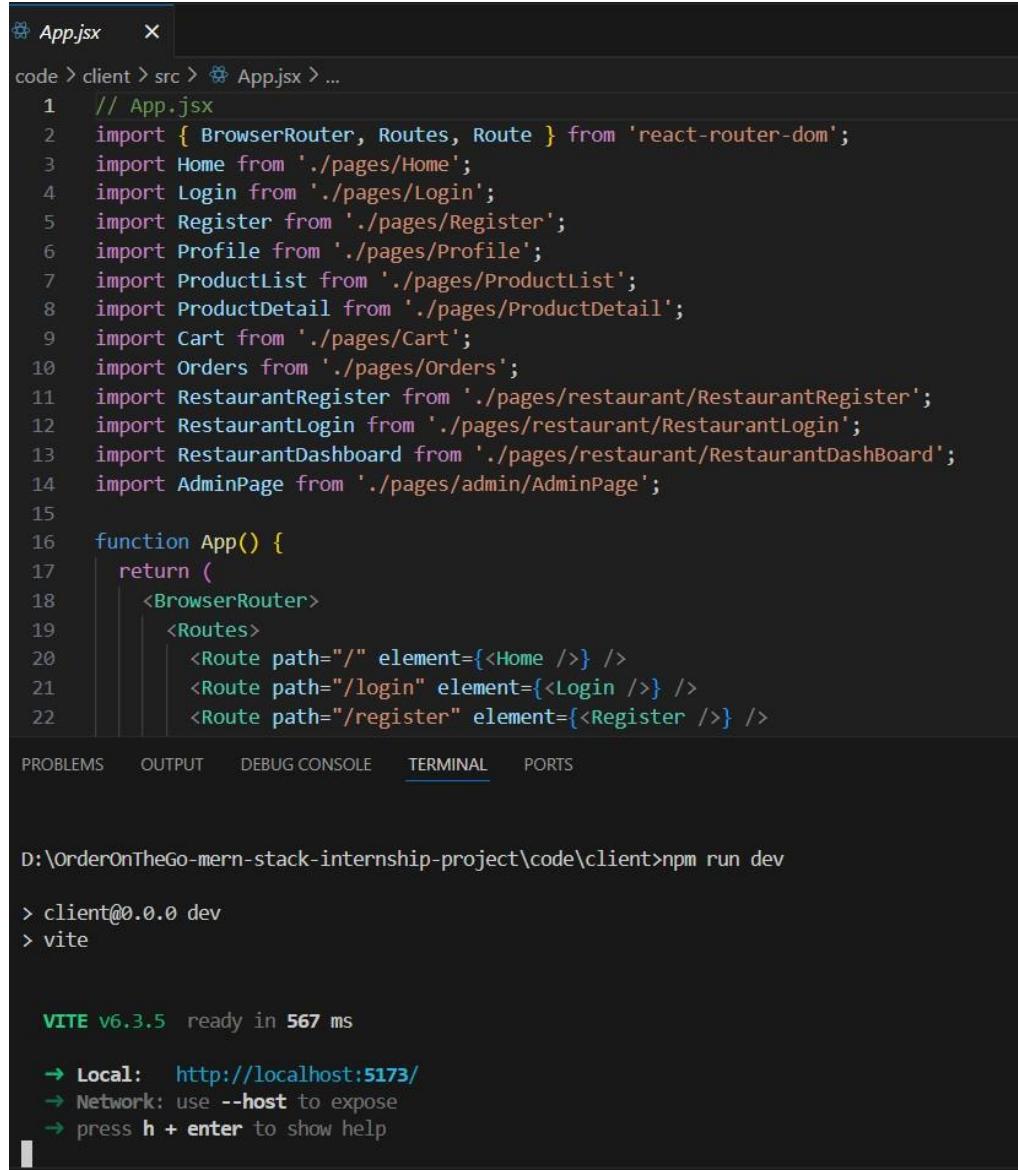
- Integration with API endpoints.

- Implement data binding.

Reference Article Link:

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

Reference Image:



```
// App.jsx
code > client > src > App.jsx > ...
1 // App.jsx
2 import { BrowserRouter, Routes, Route } from 'react-router-dom';
3 import Home from './pages/Home';
4 import Login from './pages/Login';
5 import Register from './pages/Register';
6 import Profile from './pages/Profile';
7 import ProductList from './pages/ProductList';
8 import ProductDetail from './pages/ProductDetail';
9 import Cart from './pages/Cart';
10 import Orders from './pages/Orders';
11 import RestaurantRegister from './pages/restaurant/RestaurantRegister';
12 import RestaurantLogin from './pages/restaurant/RestaurantLogin';
13 import RestaurantDashboard from './pages/restaurant/RestaurantDashBoard';
14 import AdminPage from './pages/admin/AdminPage';
15
16 function App() {
17   return (
18     <BrowserRouter>
19       <Routes>
20         <Route path="/" element={<Home />} />
21         <Route path="/login" element={<Login />} />
22         <Route path="/register" element={<Register />} />
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
```

D:\OrderOnTheGo-mern-stack-internship-project\code\client>npm run dev

> client@0.0.0 dev
> vite

VITE v6.3.5 ready in 567 ms

→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help

Code Explanation:

Server setup:

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

```
JS index.js X

code > server > JS index.js > ...

1 import express from "express";
2 import cors from "cors";
3 import mongoose from "mongoose";
4 import authRoutes from "./routes/authRoutes.js";
5 import productRoutes from "./routes/productRoutes.js";
6 import restaurantRoutes from "./routes/restaurantRoutes.js";
7 import adminRoutes from "./routes/adminRoutes.js";
8 import orderRoutes from './routes/orderRoutes.js';
9 import cartRoutes from './routes/cartRoutes.js';

10
11
12 const app = new express();
13 app.use(express.json())
14 app.use(cors());
15
16 mongoose.connect("mongodb://localhost:27017/SBfoods").
17 then(console.log("Database connected successfully"))
18 catch((err)=> console.log("error connecting mongoDb"));
19
20 app.use("/api/auth",authRoutes);
21 app.use('/api/products',productRoutes);
22 app.use('/api/restaurants',restaurantRoutes);

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

> nodemon index.js

[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Server is running
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Database connected successfully
Server is running
[]
```

User Authentication:

- Backend

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

```

JS userAuthController.js ×
code > server > controllers > JS userAuthController.js > ...
1 import User from './models/User.js';
2 import bcrypt from 'bcryptjs';
3
4 const registerUser = async (req, res) => {
5   try {
6     const { userName, password, email } = req.body;
7
8     const existingUser = await User.findOne({ email, userType: "user" });
9     if (existingUser) {
10       return res.status(400).json({ message: "Email already in use" });
11     }
12
13     const hashedPassword = await bcrypt.hash(password, 10);
14
15     const newUser = new User({
16       userName,
17       password: hashedPassword,
18       email,
19       userType: "user"
20     });
21
22     await newUser.save();
23
24     return res.status(201).json({
25       message: "User Registered Successfully",
26       user: {
27         _id: newUser._id,
28         userName: newUser.userName,
29         email: newUser.email
30       }
31     });
32
33   } catch (error) {
34     return res.status(500).json({ message: "Server error", error: error.message });
35   }
36 };
37

```

```

JS userAuthController.js ×
code > server > controllers > JS userAuthController.js > ...
38 const loginUser = async (req, res) => {
39   try {
40     const { email, password } = req.body;
41     const user = await User.findOne({ email, userType: "user" });
42
43     if (!user) {
44       return res.status(404).json({ message: "User not found" });
45     }
46
47     const isMatch = await bcrypt.compare(password, user.password);
48     if (!isMatch) {
49       return res.status(400).json({ message: "Incorrect Password" });
50     }
51
52     return res.status(200).json({
53       message: "Login Successful",
54       user: {
55         _id: user._id,
56         userName: user.userName,
57         email: user.email
58       }
59     });
60   } catch (error) {
61     return res.status(500).json({ message: "Server error", error: error.message });
62   }
63 };
64
65 export { registerUser, loginUser };
66

```

- **Frontend Login:**

```

AuthContext.jsx ×
code > client > src > context > AuthContext.jsx > ...
1 // src/context/AuthContext.jsx
2 import { createContext, useState, useContext } from 'react';
3
4 const AuthContext = createContext();
5
6 export const AuthProvider = ({ children }) => {
7   const [user, setUser] = useState(null); // null = not logged in
8
9   const login = (userData) => setUser(userData);
10  const logout = () => setUser(null);
11
12  return (
13    <AuthContext.Provider value={{ user, login, logout }}>
14      {children}
15    </AuthContext.Provider>
16  );
17}
18
19 export const useAuth = () => useContext(AuthContext);
20

```

```

const Login = () => {
  const { login } = useAuth();
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const navigate = useNavigate();

  const handleLogin = async (e) => {
    e.preventDefault();
    try {
      const res = await axios.post('http://localhost:5000/api/auth/user/login', { email, password });
      login(res.data.user); // store user info in context
      navigate('/');
    } catch (err) {
      alert('Login failed: ' + (err.response?.data?.message || 'Server Error'));
    }
  };
}


```

Register:

```

const Register = () => {
  const { login } = useAuth();
  const navigate = useNavigate();
  const [form, setForm] = useState({ userName: '', email: '', password: '' });

  const handleRegister = async (e) => {
    e.preventDefault();
    try {
      const res = await axios.post('http://localhost:5000/api/auth/user/register', form);
      login(res.data.user); // auto-login after registration
      navigate('/');
    } catch (err) {
      alert('Registration failed: ' + (err.response?.data?.message || 'Server error'));
    }
  };
}


```

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:

```
const ProductList = () => {
  const [products, setProducts] = useState([]);
  const location = useLocation();
  const navigate = useNavigate();

  // Extract query parameters
  const queryParams = new URLSearchParams(location.search);
  const restaurantId = queryParams.get('restaurantId');
  const category = queryParams.get('category');

  useEffect(() => {
    const fetchProducts = async () => {
      try {
        const query = [];
        if (restaurantId) query.push(`restaurantId=${restaurantId}`);
        if (category) query.push(`category=${category}`);
        const queryString = query.join('&');

        const res = await axios.get(`http://localhost:5000/api/products/fetch?${queryString}`);
        setProducts(res.data);
      } catch (err) {
        console.error('Error fetching products:', err);
      }
    };

    fetchProducts();
  }, [restaurantId, category]);
```

Backend

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

```
JS productController.js X
code > server > controllers > JS productController.js > ...
41
42 const fetchProduct = async (req, res) => {
43   try {
44     const { restaurantId, category } = req.query;
45
46     // Build dynamic filter
47     const filter = {};
48     if (restaurantId) filter.restaurantId = restaurantId;
49     if (category) filter.category = category;
50     if (req.query.id) filter._id = req.query.id;
51
52     const products = await Product.find(filter);
53
54     return res.status(200).json(products);
55
56   } catch (error) {
57     return res.status(500).json({ message: "Server error", error: error.message });
58   }
59 }
60
61
```

Add product to cart:

Frontend

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

```

ProductDetail.jsx ×
ode > client > src > pages > ProductDetail.jsx > ...
10  const ProductDetail = () => {
11
12    const handleAddToCart = async (productId) => {
13      try {
14
15        if (!user) {
16          navigate('/login');
17          return;
18        }
19
20
21        await axios.post('http://localhost:5000/api/cart/addItem', {
22          userId: user._id,
23          productId,
24          quantity
25        });
26        alert(`Added ${quantity} item(s) to cart`);
27      } catch (err) {
28        console.error('Error adding item:', err);
29        alert('Cart items can only contain items from same restaurant');
30      }
31    };
32
33  
```

- Backend

Add product to cart:

```

cartController.js ×
code > server > controllers > cartController.js > ...
1  import Cart from "../models/Cart.js";
2  import Product from "../models/Product.js";
3
4  const addToCart = async (req, res) => {
5    try {
6      const { userId, productId, quantity } = req.body;
7
8      const product = await Product.findById(productId);
9      if (!product) {
10        return res.status(404).json({ message: "Product not found" });
11      }
12
13      // Check for existing cart
14      let cart = await Cart.findOne({ userId });
15
16      if (cart) {
17        // Check if same restaurant
18        if (cart.restaurantId !== product.restaurantId) {
19          return res.status(400).json({ message: "Cart contains items from another restaurant. Clear cart first." });
20        }
21
22        // Check if product already in cart
23        const itemIndex = cart.items.findIndex(item => item.productId === productId);
24        if (itemIndex > -1) {
25          cart.items[itemIndex].quantity += 1;
26        } else {
27          cart.items.push({ productId, quantity });
28        }
29
30        // Recalculate price
31        let total = 0;
32        for (const item of cart.items) {
33          const p = await Product.findById(item.productId);
34          total += p.price * item.quantity;
35        }
36      }
37
38      res.json({ message: "Item added to cart" });
39    } catch (err) {
40      console.error('Error adding item:', err);
41      res.status(500).json({ message: "Internal server error" });
42    }
43  };
44
45  
```

```

50
51
52
53
54
55
56
57
58

```

```

    cart.totalPrice = total;
    await cart.save();

} else {
    // Create new cart
    cart = new Cart({
        userId,
        restaurantId: product.restaurantId,
        items: [{ productId, quantity}],
        totalPrice: product.price
    });

    await cart.save();
}

return res.status(200).json({ message: "Added to cart", cart });
} catch (error) {
    return res.status(500).json({ message: "Server Error", error: error.message });
}
};

}

```

Order products:

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

- Frontend

```

Cart.jsx  X
code > client > src > pages > Cart.jsx > ...
9  const Cart = () => {
53  const handlePlaceOrder = async () => {
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98

```

```

    return;
}

// Get current position
navigator.geolocation.getCurrentPosition(async (position) => {
    const { latitude, longitude } = position.coords;
    const now = new Date();
    const createdAt = `latitude : ${latitude} longitude : ${longitude} ${now.toDateString()} ${now.toLocaleTimeString()}`;

    const orderData = {
        userId: user._id,
        items: cart.items.map((item) => ({
            productId: item.productId,
            quantity: item.quantity
        })),
        createdAt
    };

    try {
        // Step 1: Place the order
        await axios.post('http://localhost:5000/api/orders/place', orderData);
        alert("Order placed successfully!");

        // Step 2: Delete each item from the cart
        for (const item of cart.items) {
            await axios.post('http://localhost:5000/api/cart/deleteItem', {
                userId: user._id,
                productId: item.productId
            });
        }

        // Step 3: Refresh the cart
        fetchCart(); // This will set cart to null if empty
    } catch (err) {
        console.error("Failed to place order:", err);
        alert("Failed to place order");
    }
}, (error) => {
    console.error("Location access denied or failed:", error);
    alert("Please enable location access to place the order.");
});

```

- Backend

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

```
js orderController.js ×
code > server > controllers > js orderController.js > ...
4  const placeOrder = async (req, res) => {
5    try {
6      const { userId, items, createdAt } = req.body;
7
8      let totalPrice = 0;
9      let finalPrice = 0;
10
11     for (const item of items) {
12       const product = await Product.findById(item.productId);
13
14       if (!product) {
15         return res.status(404).json({ message: `Product with ID ${item.productId} not found` });
16       }
17
18       const itemTotal = product.price * item.quantity;
19       const itemDiscount = (product.discount / 100) * itemTotal;
20
21       totalPrice += itemTotal;
22       finalPrice += itemTotal - itemDiscount;
23     }
24
25     const newOrder = new Order({
26       userId,
27       items,
28       totalPrice,
29       finalPrice,
30       createdAt
31     });
32
33     await newOrder.save();
34
35     return res.status(200).json({ message: "Order placed successfully", orderId: newOrder._id });
36
37   } catch (error) {
38     return res.status(500).json({ message: "Server Error", error: error.message });
39   }
40 }
```

In 1 Col 1 Spaces: 2 UFT-8 CRLF

Add new product:

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

- Frontend:

```
code > client > src > pages > restaurant > RestaurantDashBoard.jsx > ...
10  const RestaurantDashboard = () => {
99    const handleAddProduct = async () => {
100      try {
101        await axios.post('http://localhost:5000/api/products/insert', {
102          ...newProduct,
103          restaurantId: restaurant._id,
104          price: Number(newProduct.price),
105          discount: Number(newProduct.discount),
106        });
107        alert('Product added');
108        setNewProduct({ productName: '', description: '', price: '', discount: '', category: '', imageUrl: '' });
109        fetchProducts();
110      } catch (err) {
111        alert('Failed to add product');
112      }
113    };
114  }
```

Backend:

```
js productController.js > ...
code > server > controllers > productController.js > ...
4  const insertProduct = async (req, res) => {
5    try {
6      const {
7        productName,
8        description,
9        restaurantId,
10       price,
11       discount,
12       category,
13       imageUrl
14     } = req.body;
15
16    const restaurant = await Restaurant.findById(restaurantId);
17    if (!restaurant) {
18      return res.status(404).json({ message: "Restaurant not found" });
19    }
20
21    const newProduct = new Product({
22      productName,
23      description,
24      restaurantId,
25      price,
26      discount,
27      category,
28      imageUrl,
29      rating: 0,
30      totalRatings: 0
31    });
32
33    await newProduct.save();
34    return res.status(201).json({ message: "Product inserted successfully", product: newProduct });
35
36  } catch (error) {
37    return res.status(500).json({ message: "Server error", error: error.message });
38  }
39}
40
```

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

5.Project Implementation & Execution

Restaurants :

 [Home](#) [Cart \(2\)](#) [My Orders](#) [Logout](#)

Popular Restaurants

 Dosa Hub ★ 4.4 South Indian 20 mins View Menu	 Sri Tiffins ★ 4.3 Tiffins 18 mins View Menu	 Paradise Biryani ★ 4.6 Biryani 30 mins View Menu	 Andhra Meals House ★ 4.5 Andhra Meals 25 mins View Menu	 North Spice ★ 4.4 North Indian 28 mins View Menu
				

Restaurant Menu :

 [Home](#) [Cart \(2\)](#) [My Orders](#) [Logout](#)

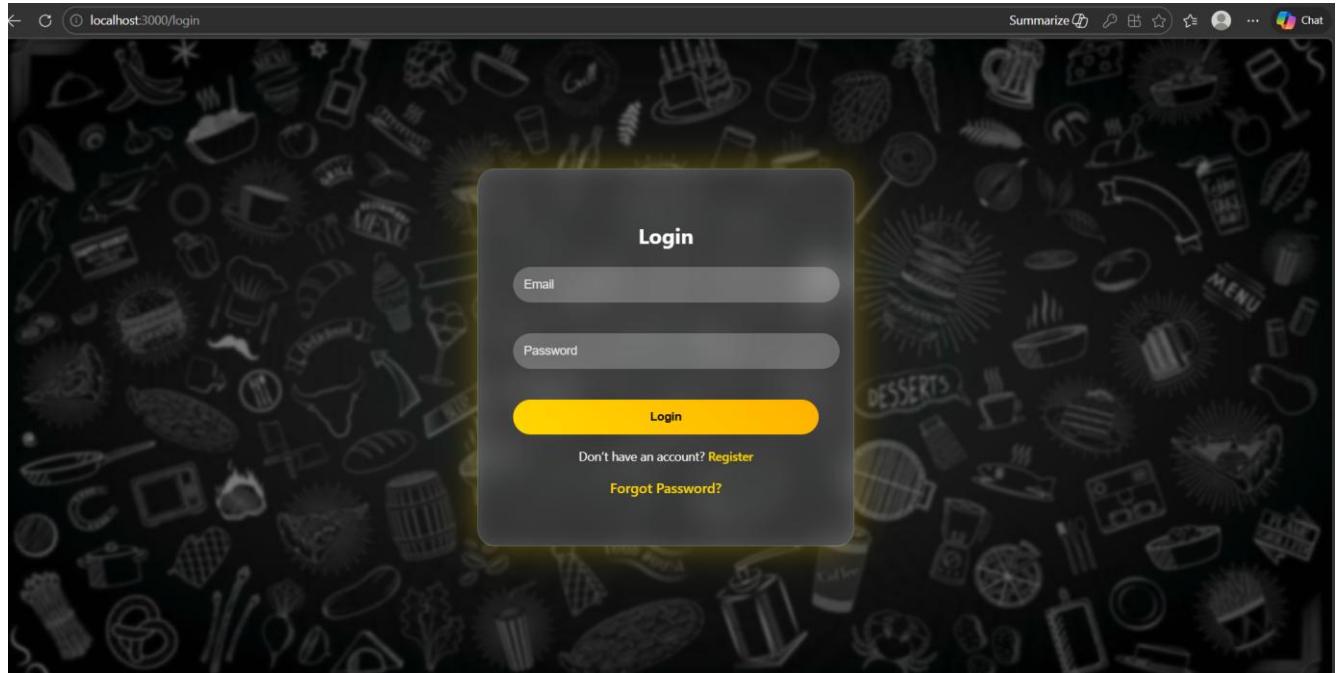
Sri Tiffins

★ 4.3 • 18 mins

MENU

 Idly (2 pcs) Soft steamed idlies ₹ 40 Add	 Vada (2 pcs) Crispy medu vada ₹ 50 Add	 Poori (3 pcs) Puffy poori with curry ₹ 70 Add	 Upma Traditional rava upma ₹ 55 Add	 Pongal Ghee pongal special ₹ 65 Add
				

Authentication:



Orders :

The screenshot shows a web interface for managing orders. At the top, there's a header with the logo 'OrderOnTheGo', a search bar containing 'Search for food...', and navigation links for 'Home', 'Cart (2)', 'My Orders', and 'Logout'. Below the header, the main content area is titled 'My Orders'. It shows a single pending order with the following details:

Status: Pending	
Date: 28/2/2026, 4:06:18 pm	
	Idly (2 pcs) ₹40 × 1
	Filter Coffee ₹30 × 1
Total: ₹70	

Cart:

 OrderOnTheGo

Search for food...

Home Cart (2) My Orders Logout

Your Cart

Vada (2 pcs)



Crispy medu vada

Price: ₹50

Quantity: 1

[+] [-]

Poori (3 pcs)



Puffy poori with curry

Price: ₹70

Quantity: 1

[+] [-]

Total: ₹120

Place Order