Full Stack Development with MERN

Project Documentation format

1. Introduction

- **Project Title:** Food Mine [Ordering food]
- **Team Members:**
    1. K Yoshitha
    2. Y Jayasri
    3. V Indira
    4. Ch Raghu Ram

2. Project Overview

- **Purpose:**

  The Food Mine project aims to streamline the process of ordering food from various restaurants. It provides a user-friendly platform for browsing menus, selecting items, and placing orders. The project seeks to enhance convenience for customers by offering seamless payment and delivery options. It integrates features such as real-time order tracking and personalized recommendations. Restaurants benefit from increased visibility and efficient order management. The ultimate goal is to improve the overall dining experience by leveraging technology to simplify and expedite food ordering. This project targets both individual customers and restaurants to create a mutually beneficial ecosystem.

  Features:

1. **User-friendly Interface**: Easy navigation for browsing menus and selecting items.
2. **Real-time Order Tracking**: Customers can track their orders from preparation to delivery.
3. **Personalized Recommendations**: Tailored suggestions based on user preferences and order history.
4. **Secure Payment Options**: Multiple payment methods with robust security for safe transactions.
5. **Efficient Order Management**: Restaurants can manage orders seamlessly, from acceptance to preparation and delivery.
6. **Customer Reviews and Ratings**: Users can leave feedback and read reviews to make informed choices.
7. **Promotions and Discounts**: Offers and discounts to attract and retain customers, enhancing the overall value proposition.

3. Architecture

- **Frontend:**

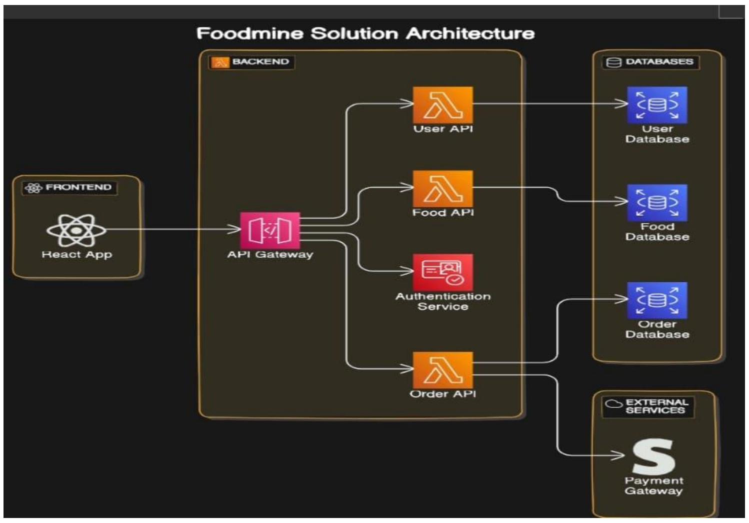
  The frontend architecture for the Food Menu system using React features a component-based structure with reusable modules like `Menu`, `Cart`, and `Order-Summary`. State management is handled using Redux or Context API for global state, with React Router for seamless navigation. Axios or Fetch API manages backend integration for dynamic data fetching. Libraries like Material-UI enhance the UI/UX, and performance optimizations include code splitting and lazy loading. Form handling is streamlined with React Hook Form or Formic, ensuring robust and efficient user input management.

Overview:

1. **Component-based Structure**: Modular design
2. **State Management**: Global state
3. **Routing**: Seamless navigation
4. **API Integration**: Dynamic data
5. **UI/UX Enhancements**: Visual consistency
6. **Form Handling**: Input validation
7. **Performance Optimization**: Load efficiency
8. **Testing**: Reliability assurance

Key Components:

1. **User Registration**: Account creation
2. **Restaurant Listings**: Browse options
3. **Menu Browsing**: Item selection
4. **Cart Management**: Order customization
5. **Order Placement**: Checkout process
6. **Payment Integration**: Secure transactions
7. **Order Tracking**: Real-time updates



Backend:

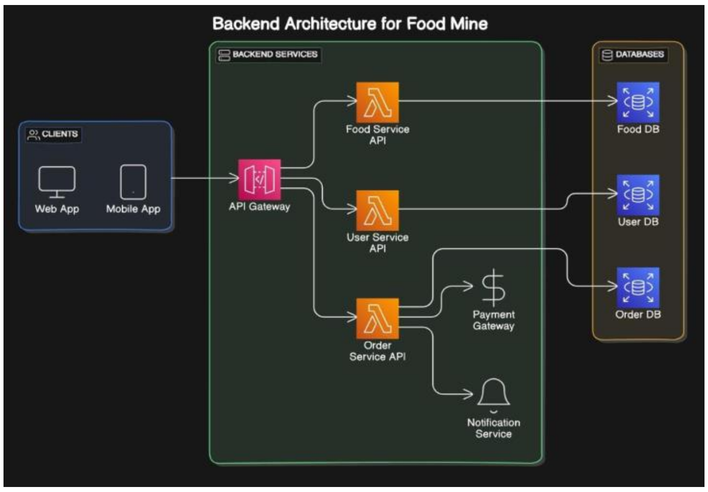
The backend architecture for the Food Mining system complements the React frontend by utilizing a Node.js and Express.js server to handle API requests and responses. It manages user authentication, interacts with a database (e.g., MongoDB or PostgreSQL) to store and retrieve data, and processes business logic for order management. The server ensures secure payment transactions through integration with payment gateways. Real-time updates and notifications are handled using WebSocket or similar technologies for seamless user experience

Overview:

1. **API Development**: Create RESTful or GraphQL APIs for client-server communication.
2. **Database Management**: Use MongoDB for storing and managing user, restaurant, menu, and order data.
3. **Authentication**: Implement user authentication and authorization for secure access.
4. **Order Processing**: Handle order placement, updates, and status tracking.
5. **Payment Integration**: Integrate with payment gateways for secure transactions.
6. **Real-time Updates**: Provide real-time notifications and updates on order status.
7. **Scalability**: Ensure the backend is scalable to handle varying loads and traffic efficiently.

Key Concepts:

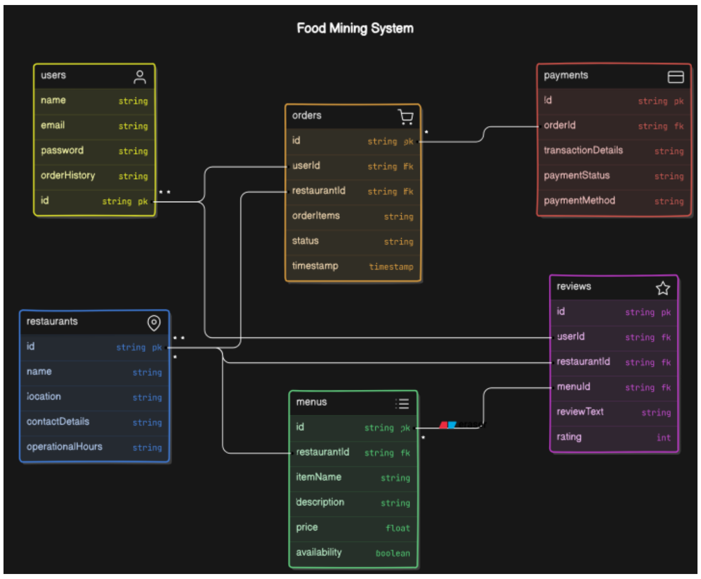
1. **API Endpoints**: Define routes for user interactions, order management, and data retrieval.
2. **Database Schema**: Structure collections for users, restaurants, menus, orders, and reviews.
3. **Authentication & Authorization**: Implement secure user login, registration, and role management.
4. **Order Management**: Process and track orders from placement to delivery.
5. **Payment Processing**: Integrate with payment systems for handling transactions securely



Database:

The database schema for a Food Mining system using MongoDB typically includes collections for users, restaurants, menus, orders, and reviews. The users collection stores account information and order history. The restaurants collection includes details such as location and menu items. The menus collection holds item details, prices, and availability. Orders are tracked in the orders collection, capturing user, restaurant, and status details, while the reviews collection contains user feedback on restaurants and items. Interactions involve CRUD operations to manage these collections, using MongoDB's flexible schema design to adapt to evolving requirements. In a Food Mining system, the database architecture typically includes the following key collections:

1. **Users**: Stores user profiles, including personal details, login credentials, and order history.
2. **Restaurants**: Contains restaurant information such as name, location, contact details, and operational hours.
3. **Menus**: Lists menu items for each restaurant, including item names, descriptions, prices, and availability.
4. **Orders**: Tracks details of each order, including user information, restaurant details, order items, status, and timestamps.
5. **Reviews**: Records user feedback and ratings for restaurants and menu items, including review text and rating scores.
6. **Payments**: Manages payment information related to orders, including transaction details, payment status, and method used.



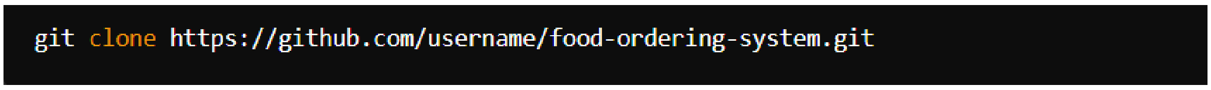
5. Setup Instructions

Prerequisites:

1. React
2. Redux
3. GIT

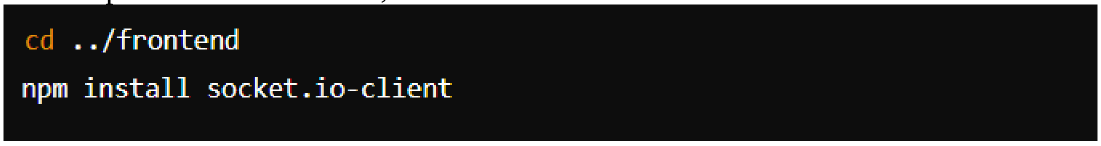
Installation:

1. Clone the repository:

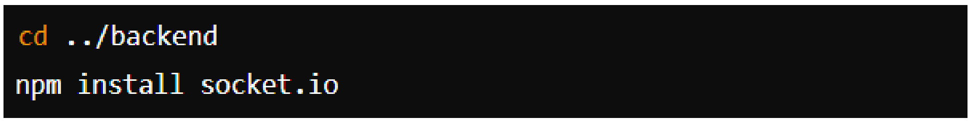


```
git clone https://github.com/username/food-ordering-system.git
```

2. Install dependencies for client , server and socket

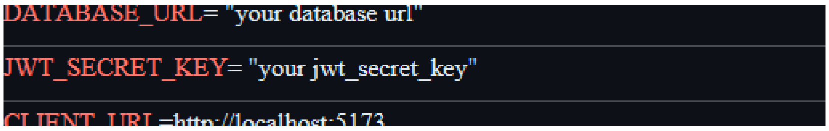


```
cd ../frontend
npm install socket.io-client
```



```
cd ../backend
npm install socket.io
```

3. Set up environment variables:
Create a .env file in the server directory (api) with following environment variables



```
DATABASE_URL= "your database url"
JWT_SECRET_KEY= "your jwt_secret_key"
CLIENT_URL=http://localhost:5173
```

5. Folder Structure

- **Client:**

```
client/
├── public/
│   └── vite.svg
├── src/
│   ├── assets/
│   │   └── images/
│   │       └── logo.png
│   ├── components/
│   │   ├── Navbar.jsx
│   │   └── Sidebar.jsx
│   ├── pages/
│   │   ├── Add/
│   │   │   ├── Add.jsx
│   │   │   └── Add.css
│   │   ├── List/
│   │   │   ├── List.jsx
│   │   │   └── List.css
│   │   ├── Orders/
│   │   │   ├── Orders.jsx
│   │   │   └── Orders.css
│   │   ├── Cart/
│   │   │   ├── Cart.jsx
│   │   │   └── Cart.css
│   │   ├── Home/
│   │   │   ├── Home.jsx
│   │   │   └── Home.css
│   │   ├── MyOrders/
│   │   │   ├── MyOrders.jsx
│   │   │   └── MyOrders.css
│   │   ├── Profile/
│   │   │   ├── Profile.jsx
│   │   │   └── Profile.css
│   │   ├── Login/
│   │   │   ├── Login.jsx
│   │   │   └── Login.css
│   │   ├── Register/
│   │   │   ├── Register.jsx
│   │   │   └── Register.css
│   ├── App.jsx
│   ├── index.css
│   ├── main.jsx
│   └── vite.config.js
├── .eslintrc.cjs
├── .gitignore
├── index.html
├── package-lock.json
├── package.json
└── vite.config.js
```

- **Server:**

```
server/
├── api/
│   ├── controllers/
│   │   ├── authController.js
│   │   ├── cartController.js
│   │   ├── orderController.js
│   │   ├── postController.js
│   │   └── userController.js
│   ├── middleware/
│   │   └── verifyToken.js
│   ├── models/
│   │   ├── User.js
│   │   ├── Post.js
│   │   ├── Order.js
│   │   └── Cart.js
│   ├── routes/
│   │   ├── authRoutes.js
│   │   ├── cartRoutes.js
│   │   ├── orderRoutes.js
│   │   ├── postRoutes.js
│   │   └── userRoutes.js
│   ├── utils/
│   │   └── errorHandler.js
│   └── prisma.js
├── server.js
└── package.json
```

6. Running the Application

commands to start the frontend and backend servers locally.

- **Frontend:** `npm start` in the client directory.
- **Backend:** `npm start` in the server directory.

7. API Documentation

1. Food Management

• **GET /api/foods**

- Description: Retrieves all food items.

- Example Response:

```
[
  {
    "id": "FOOD_ID",
    "name": "Food Name",
    "description": "Food Description",
    "price": 9.99
  }
]
```

POST /api/foods

- **Description**: Adds a new food item.
- **Parameters**:
    - name (string, required)
    - description (string, required)
    - price (number, required)
    - Example Response:

```
{
"message": "Food item added successfully"
}
```

PUT /api/foods/{id}

- **Description**: Updates a food item by ID.
- **Parameters**:
    - name (string, optional)
    - description (string, optional)
    - price (number, optional)
- Example Response

```
{
"message": "Food item updated successfully"
}
```

DELETE /api/foods/{id}

- **Description**: Deletes a food item by ID.
- Example Response

```
{
  "message": "Food item deleted successfully"
}
```

2.Cart Management

• **GET /api/cart**

- **Description**: Retrieves the user's cart.
- Example Response

```
    {
  "items": [
{
  "foodId": "FOOD_ID",
   "name": "Food Name",
      "quantity": 2,
    "price": 19.98
}
  ],
 "total": 19.98
}
```

POST /api/cart

- **Description**: Adds a food item to the cart.
- **Parameters**:
    - foodId (string, required)
    - quantity (number, required)
- Example Response

```
 {
  "message": "Item added to cart successfully"
}
```

PUT /api/cart/{foodId}

- **Description**: Updates the quantity of a food item in the cart.
- **Parameters**:
  - quantity (number, required)
  - Example Response

```
{
  "message": "Cart updated successfully"
}
```

DELETE /api/cart/{foodId}

- **Description**: Removes a food item from the cart.
- Example Response

```
{
  "message": "Item removed from cart successfully"
}
```

3.Order Management

• POST /api/orders

- **Description**: Places a new order.
- **Parameters**:
  - cartId (string, required)
  - address (string, required)
  - paymentMethod (string, required)
- **Example Response**:

json

Copy code

```
{
  "message": "Order placed successfully",
  "orderId": "ORDER_ID"
}
```

• GET /api/orders

- **Description**: Retrieves all orders for the logged-in user.
- **Example Response**:

  json

  Copy code

```
[
  {
    "id": "ORDER_ID",
    "items": [
      {
        "foodId": "FOOD_ID",
        "name": "Food Name",
        "quantity": 2,
        "price": 19.98
      }
    ],
    "total": 19.98,
    "status": "Pending",
    "address": "User Address",
    "paymentMethod": "Credit Card",
    "createdAt": "2024-07-20T12:34:56.789Z"
  }
```

• **GET /api/orders/{id}**

- **Description**: Retrieves an order by ID.
- **Example Response**:

  json

  Copy code

```
{
  "id": "ORDER_ID",
  "items": [
    {
      "foodId": "FOOD_ID",
      "name": "Food Name",
      "quantity": 2,
      "price": 19.98
    }
  ],
  "total": 19.98,
  "status": "Pending",
  "address": "User Address",
  "paymentMethod": "Credit Card",
  "createdAt": "2024-07-20T12:34:56.789Z"
}
```

4.Home Management

• GET /api/home

- **Description**: Retrieves the home page data, including featured foods and promotions.
- Example Response:

  json

  Copy code

```
{
  "featuredFoods": [
    {
      "id": "FOOD_ID",
      "name": "Food Name",
      "description": "Food Description",
      "price": 9.99
    }
  ],
  "promotions": [
    {
      "id": "PROMOTION_ID",
      "title": "Promotion Title",
      "description": "Promotion Description"
    }
  ]
}
```

5.User Authentication and Management

• POST /api/auth/register

- **Description**: Registers a new user.
- **Parameters**:
  - username (string, required)
  - email (string, required)
  - password (string, required)
- **Example Response**:

  json

  Copy code

```
{
  "message": "User registered successfully"
}
```

• POST /api/auth/login

- **Description**: Authenticates a user and returns a token.
- **Parameters**:
  - email (string, required)
  - password (string, required)
- Example Response:

  json

Copy code

```
{
  "token": "JWT_TOKEN_HERE"
}
```

• POST /api/auth/logout

- **Description**: Logs out the user.
- Example Response:

json

Copy code

```
{
  "message": "User logged out successfully"
}
```

• GET /api/users

- **Description**: Retrieves all users.
- Example Response:

json

Copy code

```
[
  {
    "id": "USER_ID",
    "username": "USERNAME",
    "email": "EMAIL"
  }
]
```

• PUT /api/users/{id}

- **Description**: Updates user information by ID.
- **Parameters**:

    - username (string, optional)
    - email (string, optional)
- Example Response:

  json

  Copy code

  ```
  {
    "message": "User updated successfully"
  }
  ```

• DELETE /api/users/{id}

- **Description**: Deletes a user by ID.
- Example Response:

  json

  Copy code

  ```
  {
    "message": "User deleted successfully"
  }
  ```

• POST /api/users/save

- **Description**: Saves a post for the user.
- **Parameters**:
    - postId (string, required)
- Example Response:

  json

  Copy code

  ```
  {
    "message": "Post saved successfully"
  }
  ```

• GET /api/users/profilePosts

- **Description**: Retrieves posts saved by the user.

- Example Response:

  json

  Copy code

  ```
  [
    {
      "id": "POST_ID",
      "title": "Post Title",
      "description": "Post Description"
    }
  ]
  ```

• GET /api/users/notification

- **Description**: Retrieves the number of notifications for the user.
- Example Response:

  json

  Copy code

  ```
  {
    "notifications": 3
  }
  ```

8. Authentication

In our Online Food Ordering System, authentication and authorization are managed through a secure token-based system using JSON Web Tokens (JWT). Below are the key aspects of how this process is handled:

1. Authentication Flow

User Registration:

- Endpoint: `POST /api/auth/register`
- Users register by providing their username, email, and password.
- Passwords are hashed before being stored in the database for security.

User Login:
- Endpoint: `POST /api/auth/login`
- Users authenticate by providing their email and password.
- If credentials are correct, a JWT token is generated and sent back to the client.

User Logout:
- Endpoint: `POST /api/auth/logout`
- Users can log out, which invalidates the token on the client side.

2. Token Handling

JWT Generation:
- Upon successful login, a JWT is generated using a secret key.
- The token contains the user's ID and expiration information.

Token Storage:
- The token is stored on the client side, typically in localStorage or cookies.
- For API requests requiring authentication, the token is included in the request headers.

Token Verification:
- Each protected route on the backend uses middleware to verify the JWT.
- Middleware checks the validity of the token. If the token is valid, the request is allowed to proceed; otherwise, it is rejected with an error.

3. Authorization

Role-Based Access Control (RBAC):
- Certain actions are restricted based on user roles (e.g., admin, vendor, customer).
- Middleware checks user roles before allowing access to specific routes.

Protected Routes:
- Routes that require authentication use the middleware to ensure only authenticated users can access them.
- Examples of protected routes: Creating, updating, or deleting orders; managing user profile information; accessing order history and saved items.

4. Security Measures

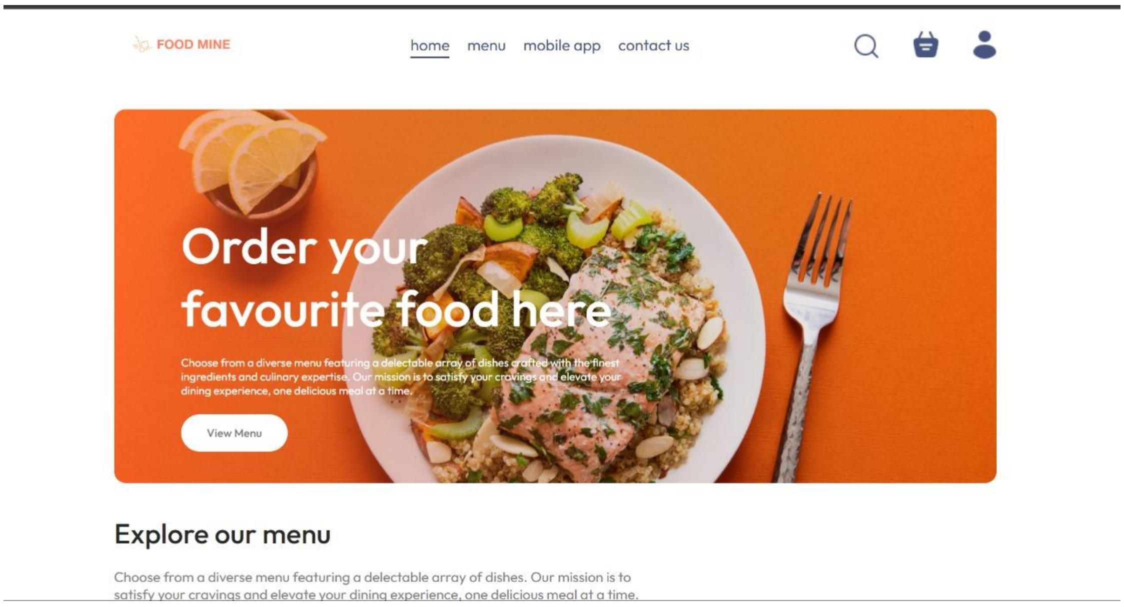
Password Hashing:
- User passwords are hashed using bcrypt before being stored in the database, ensuring that plain text passwords are never stored.

Token Expiration:
- JWT tokens have an expiration time set to limit their validity, reducing the risk of token misuse.

Secure Token Storage:
- Storing tokens in secure, http-only cookies to protect against cross-site scripting (XSS) attacks.

HTTPS:
- Ensure that all communications between the client and server are encrypted using HTTPS to protect sensitive data in transit.

9. User Interface

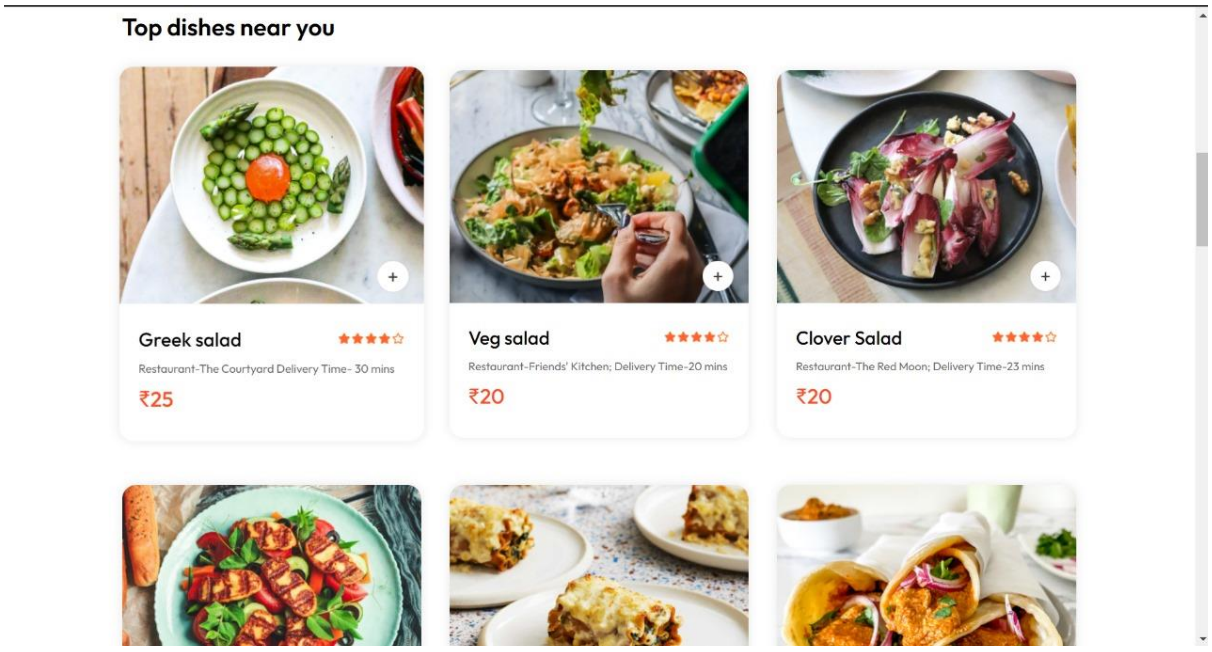


Order your favourite food here

Choose from a diverse menu featuring a delectable array of dishes crafted with the finest ingredients and culinary expertise. Our mission is to satisfy your cravings and elevate your dining experience, one delicious meal at a time.

Explore our menu

Choose from a diverse menu featuring a delectable array of dishes. Our mission is to satisfy your cravings and elevate your dining experience, one delicious meal at a time.

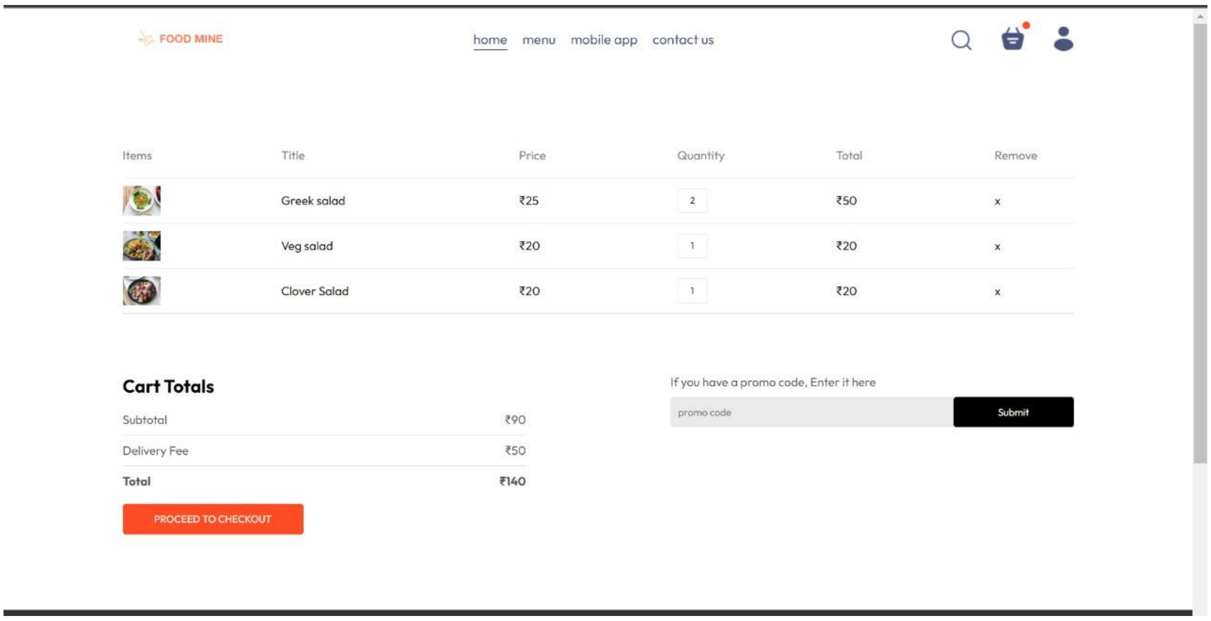


Top dishes near you

Greek salad
Restaurant-The Courtyard Delivery Time- 30 mins
₹25

Veg salad
Restaurant-Friends' Kitchen; Delivery Time-20 mins
₹20

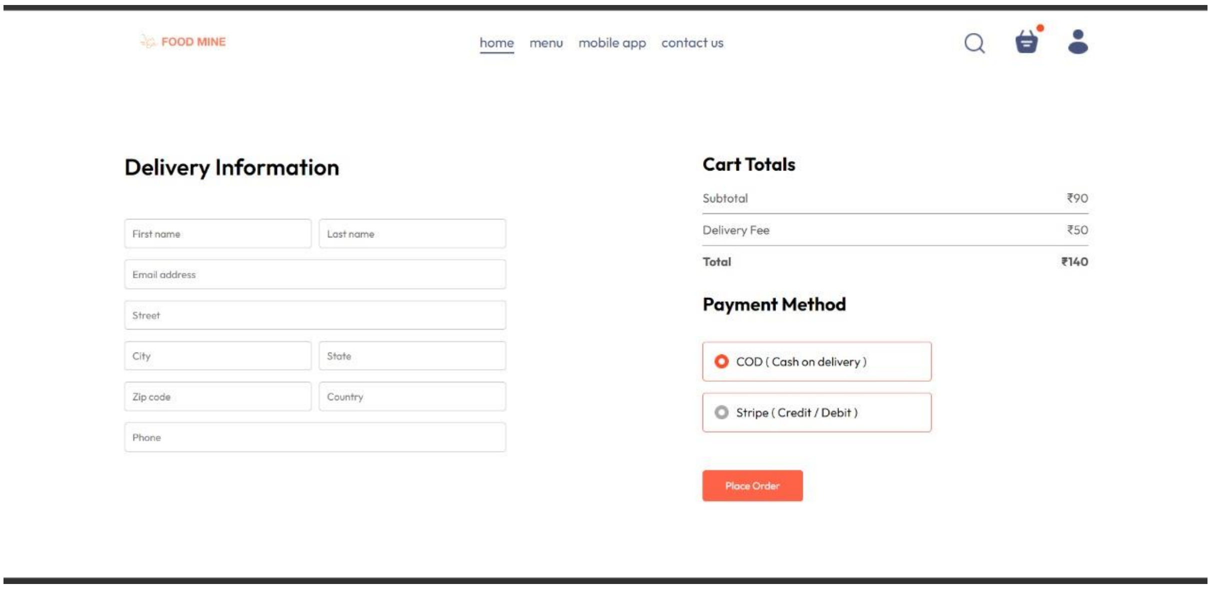
Clover Salad
Restaurant-The Red Moon; Delivery Time-23 mins
₹20



Items	Title	Price	Quantity	Total	Remove
	Greek salad	₹25	2	₹50	x
	Veg salad	₹20	1	₹20	x
	Clover Salad	₹20	1	₹20	x

Cart Totals

Subtotal	₹90
Delivery Fee	₹50
Total	**₹140**



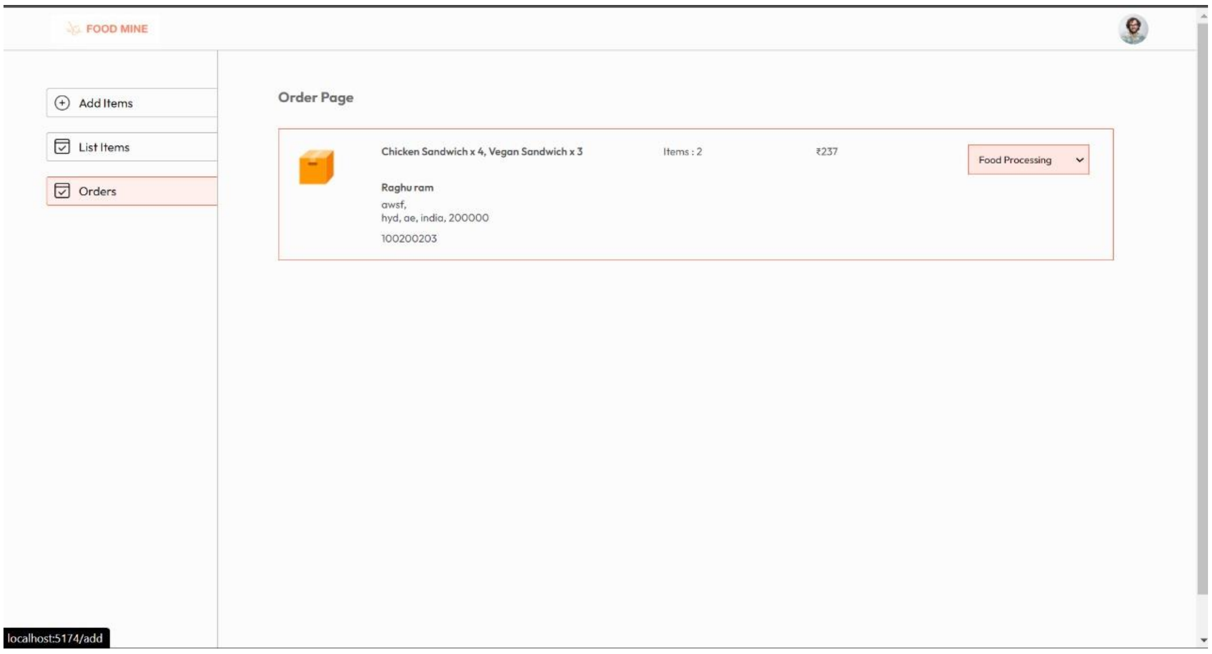
Delivery Information

Cart Totals

Subtotal	₹90
Delivery Fee	₹50
Total	**₹140**

Payment Method

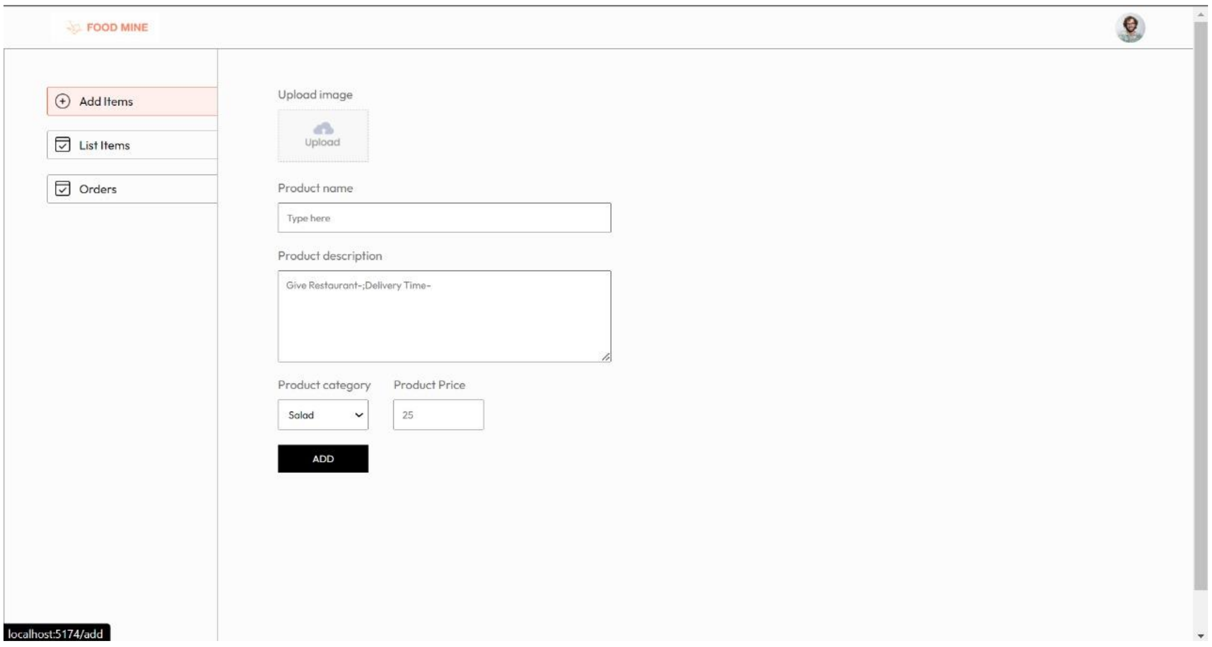
[✓ COD (Cash on delivery)]
Stripe (Credit / Debit)



Order Page

Chicken Sandwich x 4, Vegan Sandwich x 3 Items : 2 ₹237 Food Processing

Raghu ram
awsf,
hyd, ae, india, 200000
100200203



Upload image

Product name: Type here

Product description: Give Restaurant-;Delivery Time-

Product category: Salad

Product Price: 25

10. Testing

Testing Strategy and Tools Used

Our testing strategy for the online food ordering system focused on ensuring functionality, performance, and usability across various components. We employed a combination of manual testing by team members and automated testing using Postman. The goal was to validate both frontend and backend functionalities, ensuring seamless integration and user experience.

Manual Testing

- Responsiveness and Visual Consistency:

  - Team members conducted thorough manual testing across different browsers and device sizes to ensure the system is responsive and visually consistent.

- **Key Functionalities Tested:**
  - User Registration: Ensured new users can register successfully.
  - Login/Logout:Verified that users can log in and log out without issues.
  - Menu Browsing:Checked that users can browse food menus and view details of individual items.
  - Cart Management: Tested adding and removing items from the cart.
  - Order Placement:Ensured the process of placing an order is smooth and error-free.
  - Order History: Verified users can view their past orders.
  - User Interactions: Checked interactions such as adding reviews and ratings.

Automated Testing with Postman

- **API Testing:**
  - We utilized Postman for testing the API endpoints to validate backend functionalities.
- **Key Areas Tested:**
  - CRUD Operations for Menu Items:Ensured the create, read, update, and delete operations for menu items function correctly.
  - User Authentication: Validated user registration, login, and logout processes.
  - Order Management:Tested creating, viewing, and updating orders.
  - Data Validation: Checked that all required data fields are validated correctly and edge cases are handled effectively.
- Expected Responses: Ensured that API endpoints return the expected responses, including appropriate error messages for invalid requests.

11. Screenshots or Demo

DemoLink:
https://drive.google.com/file/d/1R6jCRyzyM6XFO5tefd0HyHq7pX97OUhV/view?usp=sharing

12. Known Issues

At present, there are no known issues with the developed features of the food ordering system. The team has conducted comprehensive testing on all existing functionalities, and everything is performing as expected.

However, it's important to note that, as with any software project, the introduction of new features or an increase in user interactions may reveal new issues. We will continue to monitor the system closely and address any potential concerns promptly to ensure a seamless user experience.

13. Future Enhancements

Outline potential future features or improvements that could be made to the project.

Enhanced Search Functionality

- Semantic Search
- Personalized Recommendations

User Reviews and Ratings

- Detailed Feedback
- Rating Aggregation

Booking and Scheduling

- Order Scheduling
- Reservation Management

Admin Dashboard

- Real-Time Analytics
- Order and Menu Management

Multi-language Support

- Localize Interfaces
- Automatic Translation