Food Delivery App

This is a microservice-based food delivery app built with Flask.

Setup

1. Clone the repository.
2. Create a virtual environment and activate it.
3. For windows

```
python3 -m venv venv
.\venv\Scripts\activate.bat
```

4. For Linux

```
python3 -m venv venv
source venv/bin/activate
```

5. Install the dependencies:

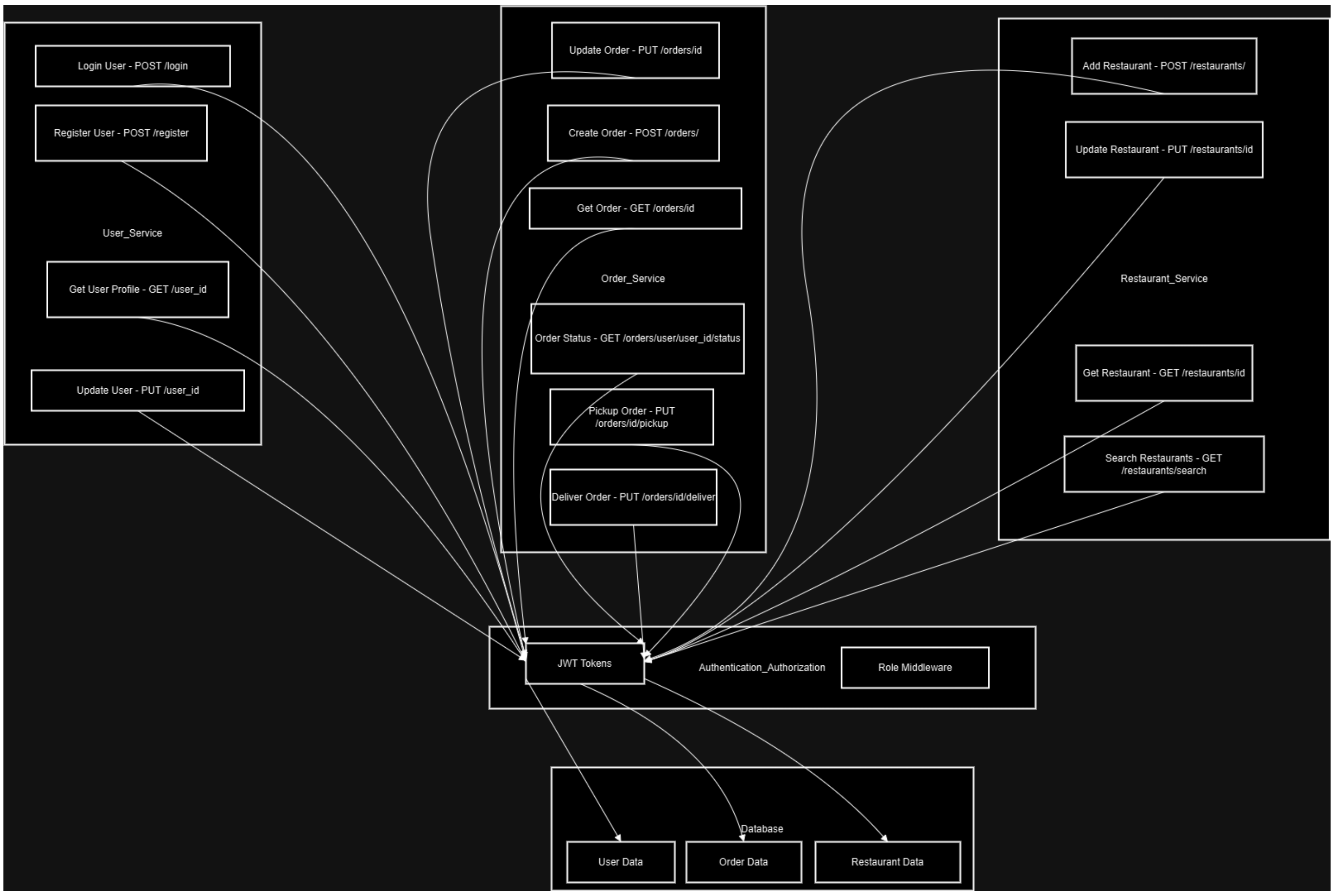
   ```
   pip install -r requirements.txt
   ```

6. Create a `.env` file and add your environment variables.
7. Run the application:

   ```
   python -m app.app
   ```

Architecture

The application is structured as a microservices-based architecture with the following components:


User_Service
Login User - POST /login
Register User - POST /register
Get User Profile - GET /user_id
Update User - PUT /user_id
Order_Service
Update Order - PUT /orders/id
Create Order - POST /orders/
Get Order - GET /orders/id
Order Status - GET /orders/user/user_id/status
Pickup Order - PUT /orders/id/pickup
Deliver Order - PUT /orders/id/deliver
Restaurant_Service
Add Restaurant - POST /restaurants/
Update Restaurant - PUT /restaurants/id
Get Restaurant - GET /restaurants/id
Search Restaurants - GET /restaurants/search
Authentication_Authorization
JWT Tokens
Role Middleware
Database
User Data
Order Data
Restaurant Data

Database Structure ### Users Table

```
+---------------------+
|       users         |
+---------------------+
| id (String, PK)     |
| username (String)   |
| email (String)      |
| password (String)   |
| role (String)       |
| phone (String)      |
| delivery_address    |
| payment_info        |
+---------------------+
```

Orders Tables

```
+---------------------+
|       orders        |
+---------------------+
| id (String, PK)     |
| user_id (String)    |
| restaurant_id       |
| items (PickleType)  |
| total_price (Float) |
| status (String)     |
+---------------------+
```

Restaurant Tables

```
+---------------------+
|    restaurants      |
+---------------------+
| id (String, PK)     |
| name (String)       |
| address (String)    |
| cuisine (String)    |
| menu (PickleType)   |
| work_hours (String) |
+---------------------+
```

As you can see the application is structured into three main microservices:

1. **User Service:** Handles user-related operations such as registration, login, profile retrieval, and profile updates.

2. **Order Service:** Manages order-related operations such as creating orders, updating order statuses, and retrieving order details.
3. **Restaurant Service:** Handles restaurant-related operations such as adding restaurants, updating restaurant details, and searching for restaurants.

API Architecture Style

The API follows the RESTful (Representational State Transfer) architecture style. This includes:

1. **Resource-Based URLs:** The API endpoints are structured around resources such as users, orders, and restaurants.
2. **HTTP Methods:** The API uses standard HTTP methods to perform CRUD (Create, Read, Update, Delete) operations.
3. **Statelessness:** Each API request contains all the information needed to process the request, typically through the use of tokens for authentication and authorization.
4. **Use of JSON:** The API uses JSON as the format for request and response bodies.
5. **Clear Separation of Concerns:** The application is structured with separate services for users, orders, and restaurants, each with its own set of endpoints and responsibilities.

Roles and Permissions

Roles

1. **User:** Can register, login, view and update their profile, place orders, and view order statuses.
2. **Restaurant Owner:** Can add, update, and view restaurant details, and update order statuses to "Accepted" or "Rejected".
3. **Delivery Agent:** Can update order statuses to "Picked Up" and "Delivered".

Permissions

1. **User:** Access to user-related endpoints.
2. **Restaurant Owner:** Access to restaurant-related endpoints and order status updates.
3. **Delivery Agent:** Access to order status updates for delivery.

Order Statuses

The following order statuses are used to track the progress of an order:

1. **Pending:** The default status when an order is created.
2. **Accepted:** The order has been accepted by the restaurant.
3. **Rejected:** The order has been rejected by the restaurant.
4. **Picked Up:** The order has been picked up by the delivery agent.
5. **Delivered:** The order has been delivered to the customer.

Data Validation

The application includes basic data validation for email and password fields:

1. The `validate_email` function ensures that the email format is valid using a regular expression.

2. The `validate_password` function ensures that the password is strong by checking its length and the presence of digits.

Endpoints

User Endpoints

- **POST /api/users/register**: Register a new user.
  - **Request Body:**

    ```
    {
      "username": "testuser",
      "email": "testuser@example.com",
      "password": "password",
      "role": "user"
    }
    ```

  - **Response:**

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

- **POST /api/users/login**: User login.
  - **Request Body:**

    ```
    {
      "username": "testuser",
      "password": "password"
    }
    ```

  - **Response:**

    ```
    {
      "token": "jwt_token"
    }
    ```

- **GET /api/users/<user_id>**: Get user profile.
  - **Response:**

```
{
  "username": "testuser",
  "email": "testuser@example.com",
  "phone": "1234567890",
  "delivery_address": "123 Test St",
  "payment_info": "Payment Info"
}
```

- **PUT /api/users/<user_id>**: Update user details.
  - **Request Body:**

    ```
    {
      "email": "newemail@example.com",
      "phone": "1234567890",
      "delivery_address": "123 New St",
      "payment_info": "New Payment Info"
    }
    ```

  - **Response:**

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

Order Endpoints

- **POST /api/orders/**: Create a new order.
  - **Request Body:**

    ```
    {
      "user_id": "testuser",
      "restaurant_id": "testrestaurant",
      "items": ["item1", "item2"],
      "total_price": 100.0
    }
    ```

  - **Response:**

    ```
    {
      "message": "Order created successfully"
    }
    ```

- **GET /api/orders/<order_id>**: Get order details.
  - **Response:**

    ```
    {
      "user_id": "testuser",
      "restaurant_id": "testrestaurant",
      "items": ["item1", "item2"],
      "total_price": 100.0,
      "status": "Pending"
    }
    ```

- **GET /api/orders/user/<user_id>/status**: Get the status of orders for a user.
  - **Response:**

    ```
    [
      {
        "user_id": "testuser",
        "restaurant_id": "testrestaurant",
        "items": ["item1", "item2"],
        "total_price": 100.0,
        "status": "Pending"
      }
    ]
    ```

- **PUT /api/orders/<order_id>/status**: Update order status (for restaurant owners).
  - **Request Body:**

    ```
    {
      "status": "Accepted"
    }
    ```

  - **Response:**

    ```
    {
      "message": "Order status updated successfully"
    }
    ```

- **PUT /api/orders/<order_id>/pickup**: Update order status to picked up (for delivery agents).
  - **Request Body:**

```
{
  "status": "Picked Up"
}
```

  - **Response:**

```
{
  "message": "Order picked up successfully"
}
```

- **PUT /api/orders/<order_id>/deliver**: Update order status to delivered (for delivery agents).

  - **Request Body:**

```
{
  "status": "Delivered"
}
```

  - **Response:**

```
{
  "message": "Order delivered successfully"
}
```

Restaurant Endpoints

- **POST /api/restaurants/**: Add a new restaurant (for restaurant owners).

  - **Request Body:**

```
{
  "name": "Test Restaurant",
  "address": "123 Test St",
  "cuisine": "Test Cuisine",
  "menu": [
    {
      "name": "item1",
      "price": 10.0
    },
    {
      "name": "item2",
      "price": 15.0
    }
  ],
  "work_hours": "9 AM - 9 PM"
}
```

  - **Response:**

```
  {
    "message": "Restaurant added successfully"
  }
```

- **GET /api/restaurants/<restaurant_id>**: Get restaurant details.
  - **Response:**

```
  {
    "name": "Test Restaurant",
    "address": "123 Test St",
    "cuisine": "Test Cuisine",
    "menu": [
      {
        "name": "item1",
        "price": 10.0
      },
      {
        "name": "item2",
        "price": 15.0
      }
    ],
    "work_hours": "9 AM - 9 PM"
  }
```

- **PUT /api/restaurants/<restaurant_id>**: Update restaurant details (for restaurant owners).
  - **Request Body:**

```
  {
    "address": "123 New St",
    "cuisine": "New Cuisine",
    "menu": [
      {
        "name": "item1",
        "price": 12.0
      },
      {
        "name": "item2",
        "price": 18.0
      }
    ],
    "work_hours": "10 AM - 10 PM"
  }
```

  - **Response:**

```
  {
    "message": "Restaurant details updated successfully"
  }
```

- **GET /api/restaurants/search**: Search for restaurants.
  - **Query Parameters:**
    - `cuisine` (optional): Filter by cuisine.
    - `max_price` (optional): Filter by maximum price.
  - **Response:**

```
  [
    {
      "name": "Test Restaurant",
      "address": "123 Test St",
      "cuisine": "Test Cuisine",
      "menu": [
        {
          "name": "item1",
          "price": 10.0
        },
        {
          "name": "item2",
          "price": 15.0
        }
      ],
      "work_hours": "9 AM - 9 PM"
    }
  ]
```

Running Tests

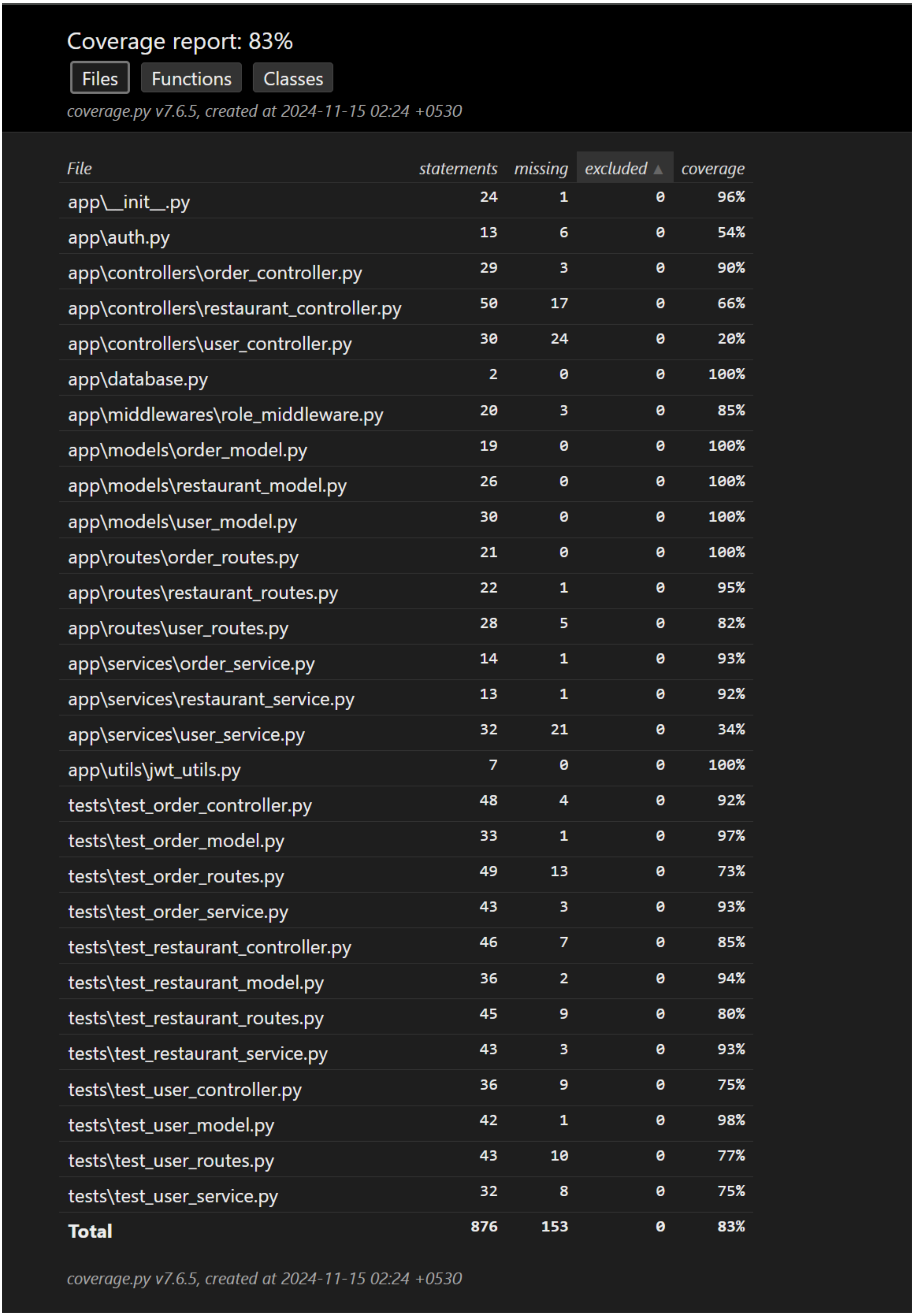
To ensure everything is working correctly, run your automated tests using `unittest`:

Using `unittest`

```
python -m unittest <test>
```

Test Coverage

The following is a screenshot of the coverage report:



Coverage report: 83%

coverage.py v7.6.5, created at 2024-11-15 02:24 +0530

File	statements	missing	excluded ▲	coverage
app\__init__.py	24	1	0	96%
app\auth.py	13	6	0	54%
app\controllers\order_controller.py	29	3	0	90%
app\controllers\restaurant_controller.py	50	17	0	66%
app\controllers\user_controller.py	30	24	0	20%
app\database.py	2	0	0	100%
app\middlewares\role_middleware.py	20	3	0	85%
app\models\order_model.py	19	0	0	100%
app\models\restaurant_model.py	26	0	0	100%
app\models\user_model.py	30	0	0	100%
app\routes\order_routes.py	21	0	0	100%
app\routes\restaurant_routes.py	22	1	0	95%
app\routes\user_routes.py	28	5	0	82%
app\services\order_service.py	14	1	0	93%
app\services\restaurant_service.py	13	1	0	92%
app\services\user_service.py	32	21	0	34%
app\utils\jwt_utils.py	7	0	0	100%
tests\test_order_controller.py	48	4	0	92%
tests\test_order_model.py	33	1	0	97%
tests\test_order_routes.py	49	13	0	73%
tests\test_order_service.py	43	3	0	93%
tests\test_restaurant_controller.py	46	7	0	85%
tests\test_restaurant_model.py	36	2	0	94%
tests\test_restaurant_routes.py	45	9	0	80%
tests\test_restaurant_service.py	43	3	0	93%
tests\test_user_controller.py	36	9	0	75%
tests\test_user_model.py	42	1	0	98%
tests\test_user_routes.py	43	10	0	77%
tests\test_user_service.py	32	8	0	75%
Total	876	153	0	83%

coverage.py v7.6.5, created at 2024-11-15 02:24 +0530

Deployed API - Food Delivery App

This is the backend API for the Food Delivery App, deployed on Render with Supabase as the PostgreSQL database. This API allows you to manage users, restaurants, and orders for a food delivery service.

Deployment Setup

Render Deployment

To deploy the app on Render, follow these steps:

1. **Create a Render Account**: Sign up or log in to Render.
2. **Create a New Web Service**:
   - Go to **Dashboard** and click on **New > Web Service**.
   - Connect your GitHub repository containing the Food Delivery App.
3. **Add a `render.yaml` File**:
   - In the root directory of your repository, add a `render.yaml` file to specify the deployment settings.
   - Here's a sample `render.yaml` file:

     ```
     services:
       - type: web
         name: food-delivery-app
         env: python
         region: oregon
         buildCommand: "pip install -r requirements.txt"
         startCommand: "flask run --host=0.0.0.0 --port=10000"
         envVars:
           - key: DATABASE_URL
             value: "YOUR_SUPABASE_POSTGRES_URL"
     ```

   - Replace `"YOUR_SUPABASE_POSTGRES_URL"` with the actual Supabase PostgreSQL connection URL.
4. **Deploy the Service**: Render will automatically build and deploy the app based on the settings in `render.yaml`.

Supabase Setup for PostgreSQL

To use Supabase as the PostgreSQL database backend:

1. **Create a Supabase Account**: Sign up or log in to Supabase.
2. **Create a New Project**:
   - Once logged in, click on **New Project**.
   - Choose a name, region, and database password, then click **Create new project**.
3. **Get the Database URL**:

  - Go to **Settings > Database** in your project and copy the `DATABASE_URL`.
  - Paste this URL into your `render.yaml` file under `DATABASE_URL` as shown above.
4. **Set Up Tables**:
  - Use the SQL Editor in Supabase to create tables (like `users`, `orders`, and `restaurants`) based on your app's data model.

Base URL

After deploying, all API requests are made to the following base URL:

https://food-delivery-app-oyfl.onrender.com/

Testing with Postman

To test these APIs using Postman:

1. **Set Up Authorization**: Use the `/api/users/login` endpoint to get a token. Copy the token and set it as a Bearer token in Postman:

- Go to the **Authorization** tab.
- Choose **Bearer Token** as the type.
- Paste your token in the field provided.

2. **Create and Test Requests**:

- Create new requests for each endpoint.
- Set the request method (e.g., GET, POST) as required.
- For POST requests, go to the **Body** tab, select **raw** and choose **JSON** as the format, and then add the payload.
- Send the request and observe the response.

3. **Use the Base URL**:

- All requests should start with the base URL `https://food-delivery-app-oyfl.onrender.com/`.

4. **Check Authorization Requirements**:

- Ensure you add the token in the **Authorization** header for endpoints that require it.

Database

The app uses Supabase as a PostgreSQL database backend.

License

This project is licensed under the MIT License.