Online Food Ordering System

Api ,Testing and DB

Members:

1. Hema Krishna Anjan Vankayala
2. Sakshi Kumari
3. Aakash D S
4. Hansakotte
5. Chander Thakur

Overview of our work

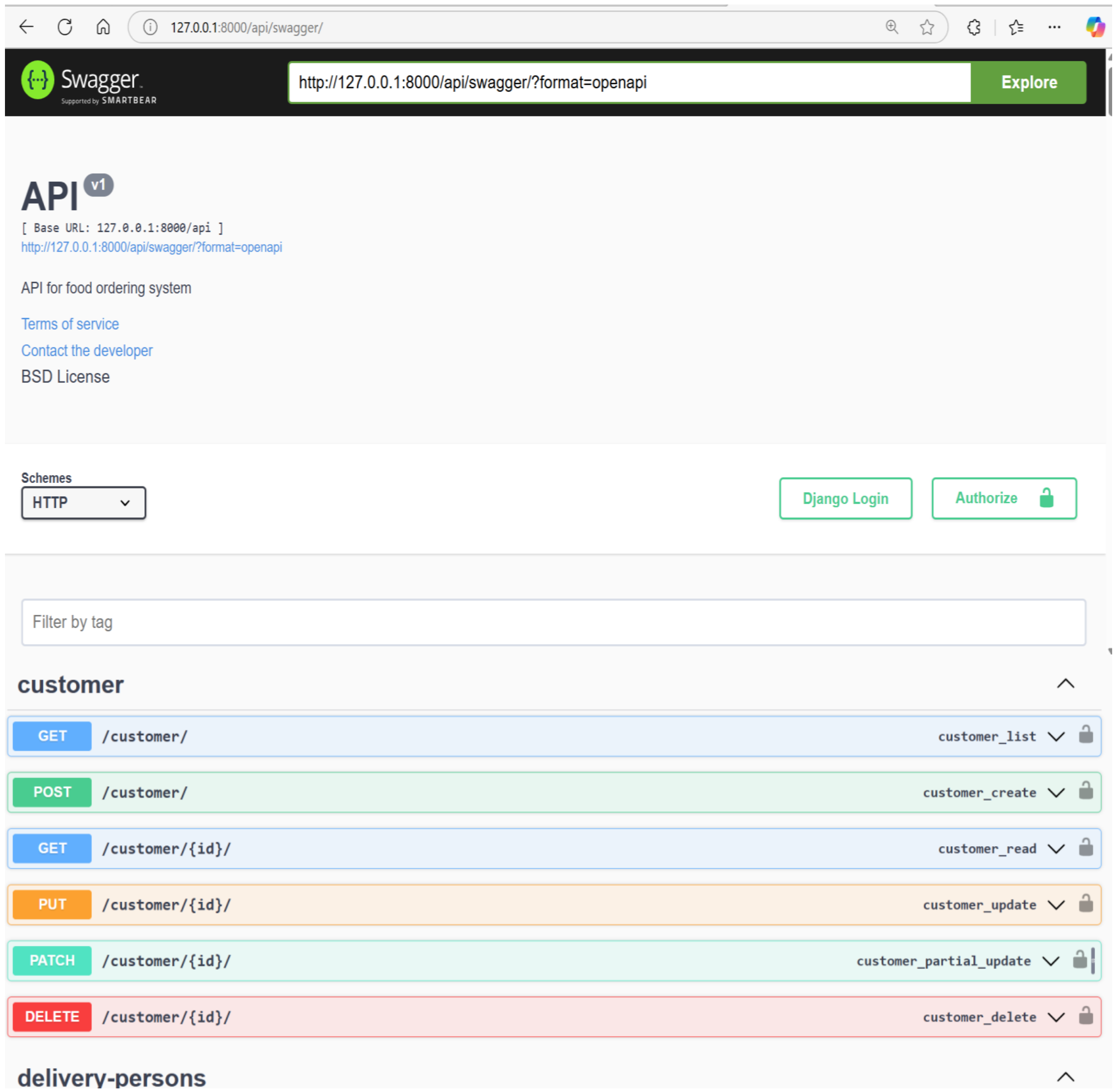
1. **API** - Developed a robust API using Django REST framework to facilitate seamless communication between the frontend and backend. Implemented token-based authentication for secure access and utilized Swagger for comprehensive API documentation and testing.

2. **Testing** - Conducted thorough testing to ensure the reliability and functionality of the system.

3. **Database** - Utilized SQLite as the database for development purposes, chosen for its simplicity and ease of integration. Designed a comprehensive database schema to manage users, restaurants, menu items, and orders efficiently.

API Overview

- APIs are created using the Django rest framework to handle CRUD operations efficiently for Apps like Customer, Restaurant , Comments , Payments etc.
- **Integrating Swagger** for interactive API documentation, allowing developers to explore, test, and visualize APIs directly from a user-friendly interface.
- **Token Authentication**: Apply the defined security scheme to the relevant API endpoints in the Swagger specification. This ensures that any requests to these endpoints require a valid token, enhancing the security of APIs.


127.0.0.1:8000/api/swagger/

Swagger

http://127.0.0.1:8000/api/swagger/?format=openapi

Explore

API v1

[Base URL: 127.0.0.1:8000/api]

http://127.0.0.1:8000/api/swagger/?format=openapi

API for food ordering system

Terms of service

Contact the developer

BSD License

Schemes: HTTP

Django Login

Authorize

Filter by tag

customer

GET	/customer/	customer_list
POST	/customer/	customer_create
GET	/customer/{id}/	customer_read
PUT	/customer/{id}/	customer_update
PATCH	/customer/{id}/	customer_partial_update
DELETE	/customer/{id}/	customer_delete

delivery-persons

Swagger Integration

- Swagger provides an interactive UI for testing API endpoints, which enhances development efficiency and accuracy.
- Endpoint to access swagger: */api/swagger/*

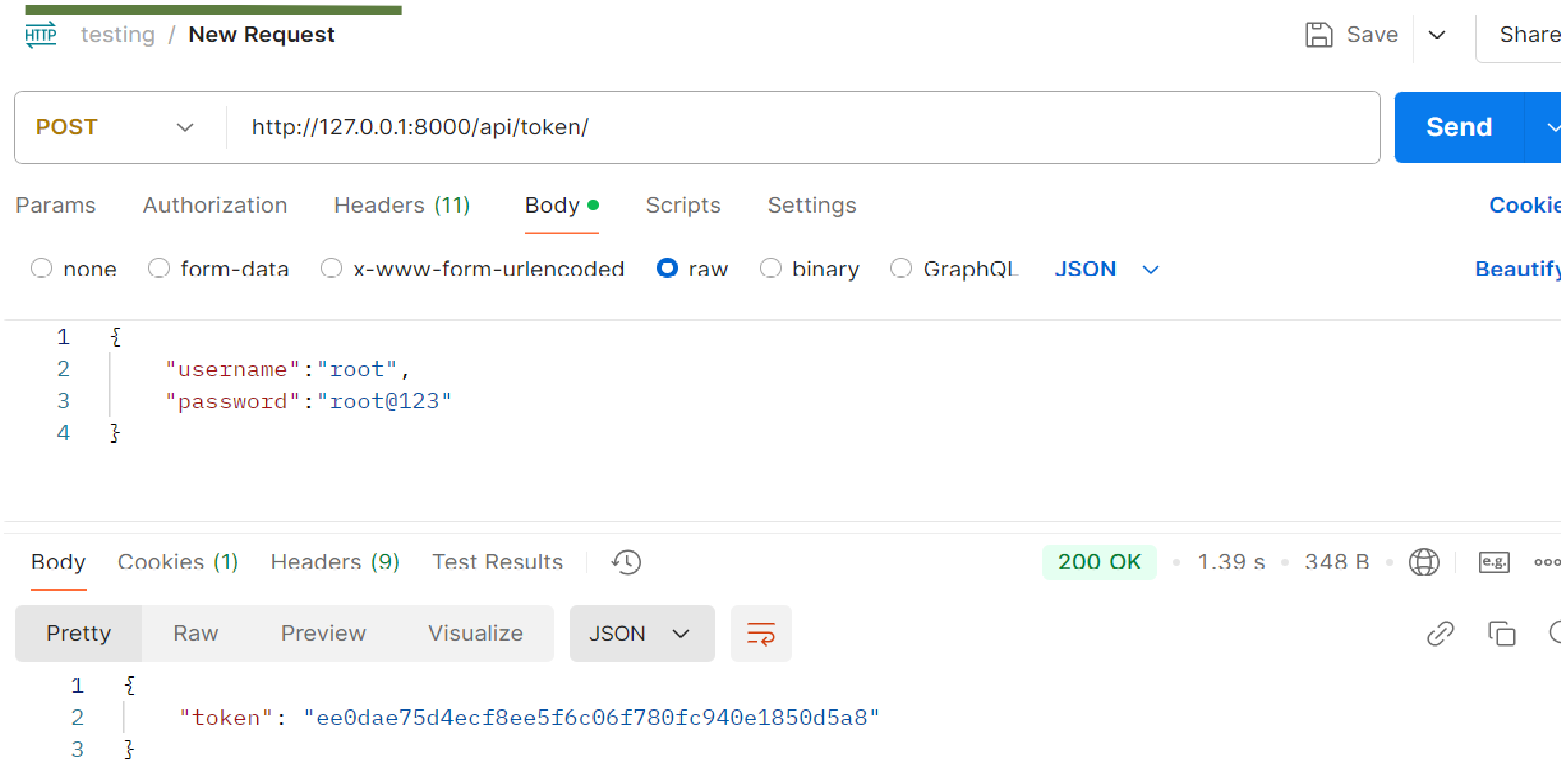
receive a token.

Token Authentication

- Token Authentication is a security mechanism to authenticate users via tokens.
- Implemented using Django REST framework’s **rest_framework.authtoken module .**
- User send a post request to **/api/token/** with username and password to receive the token.
- **Token Validity** - Using the default behavior of Django REST framework, tokens do not have an expiration mechanism. Once a token is generated, it remains valid indefinitely unless manually deleted or refreshed.

Token Authentication

On providing a **username** and **password** a token is generated for the user.



POST http://127.0.0.1:8000/api/token/

```
{
    "username":"root",
    "password":"root@123"
}
```

200 OK • 1.39 s • 348 B

JSON

```
{
    "token": "ee0dae75d4ecf8ee5f6c06f780fc940e1850d5a8"
}
```

API Endpoints

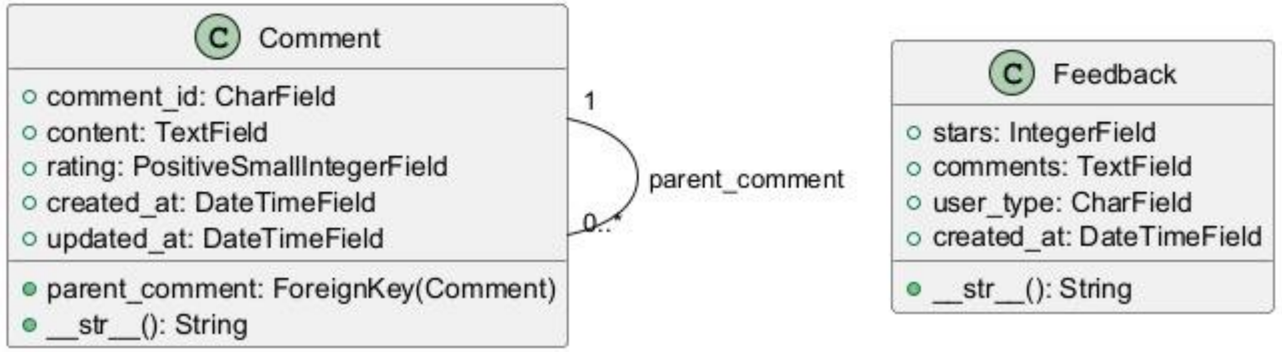
List of endpoints

- /api/token/
- /api/customer/
- /api/restaurantusers/
- /api/delivery-users/
- /api/foodItems/
- /api/order/
- /api/comments/
- /api/contacts/
- /api/feedback/
- /api/delivery-locations/
- /api/state/
- /api/place/
- /api/city/

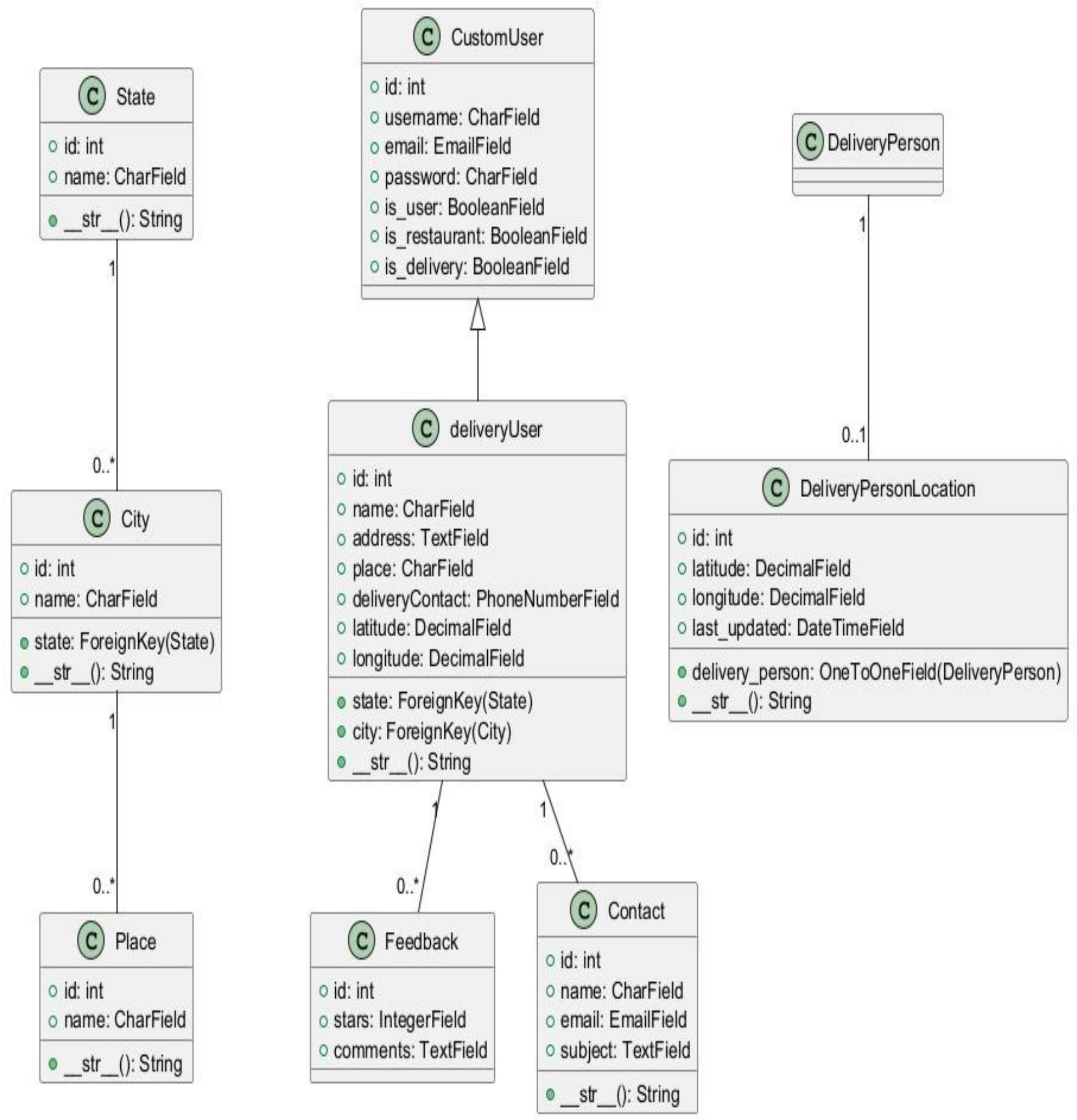
Database Overview

- SQLite was chosen for its simplicity and ease of integration during development.
- The schema includes tables for customer, restaurants, menu items, orders ,delivery, feedback, state, city and place.“

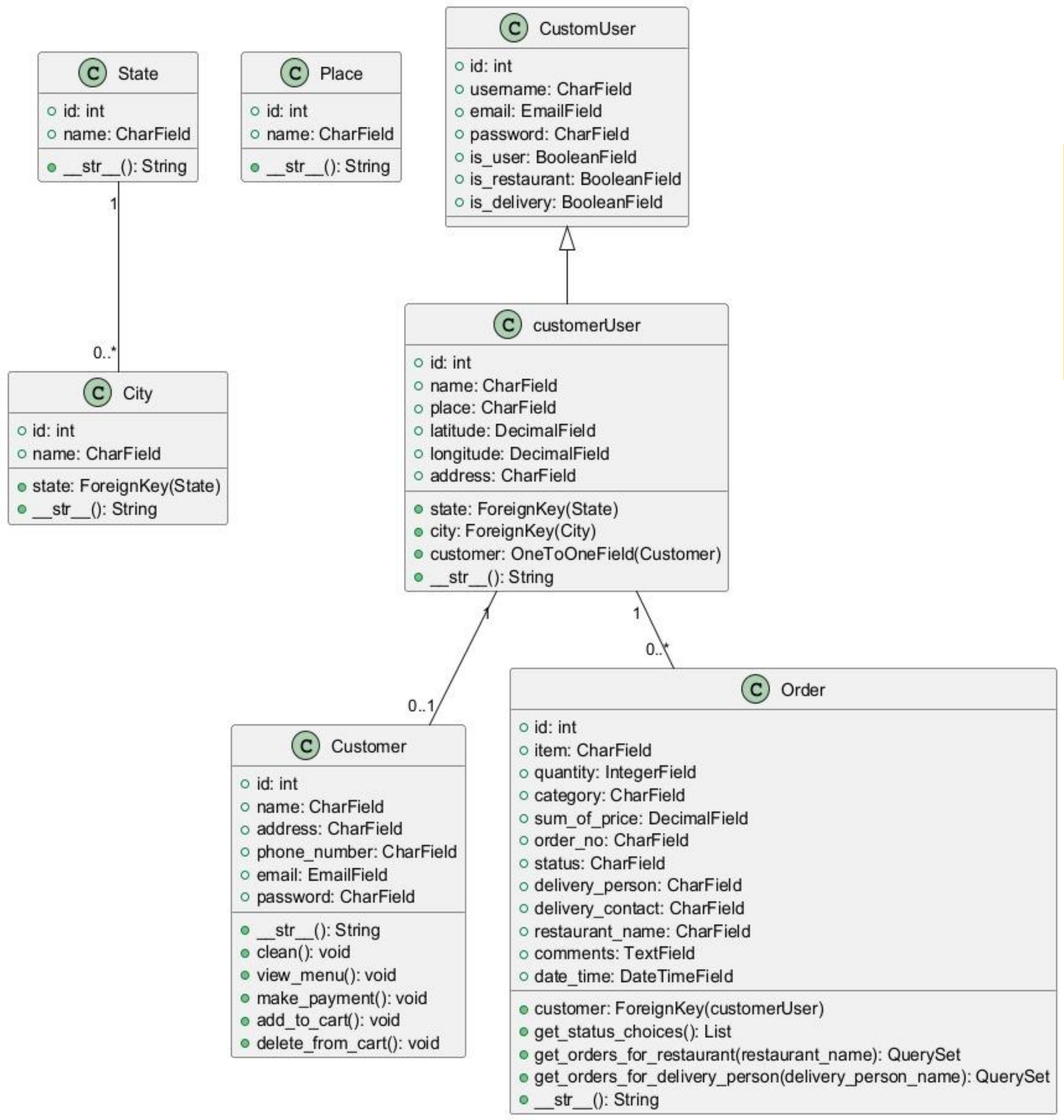
UML Diagram for Comment and Feedback Models



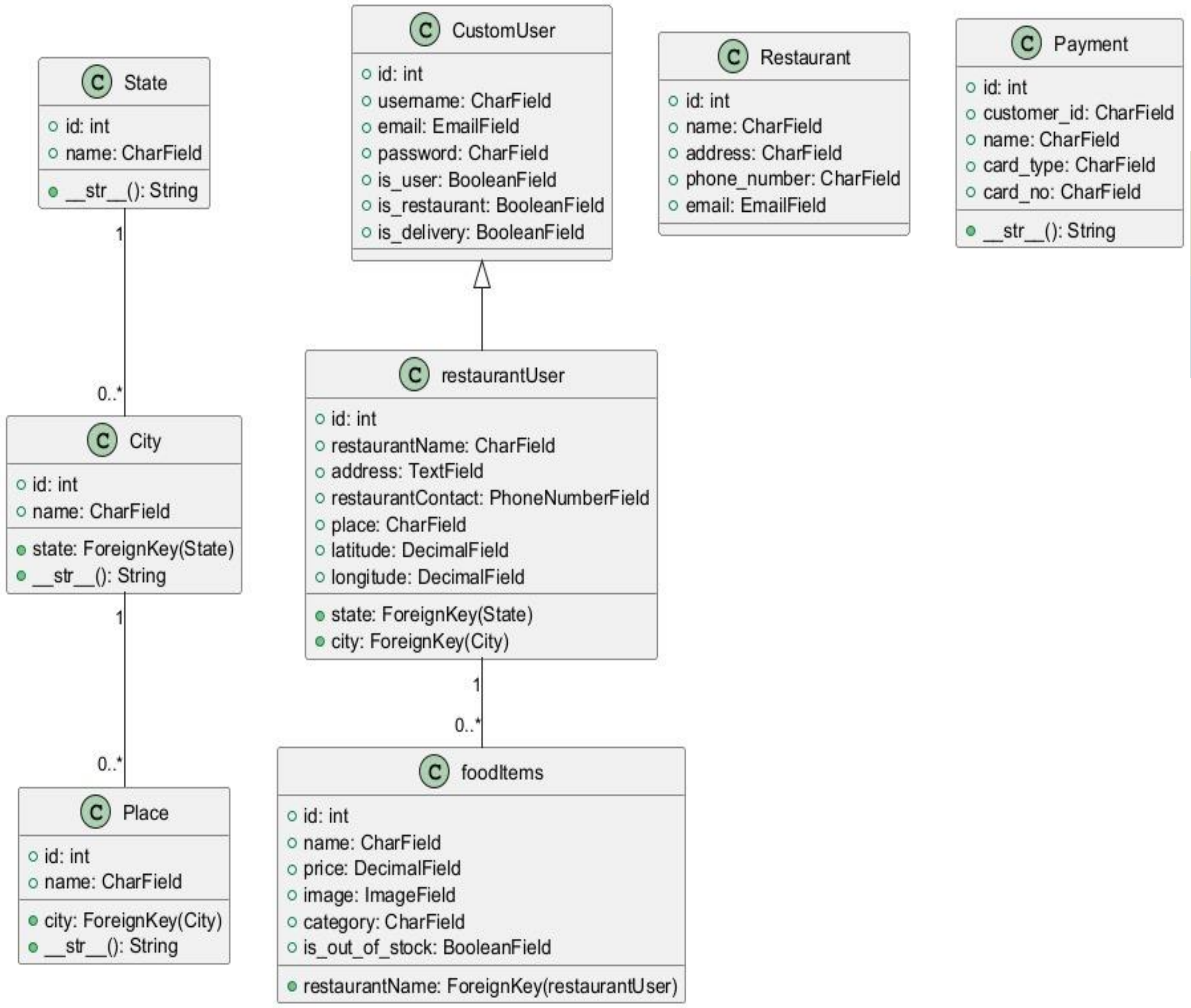
UML Diagram for Delivery Models



UML Diagram for Django Models



UML Diagram for Restaurant Models



Database Models

- CustomerUser
- RestaurantUser
- DeliveryUser
- State
- City
- Place
- Feedback
- Comment
- Delivery Person Location
- Order
- Food Items

Testing Overview

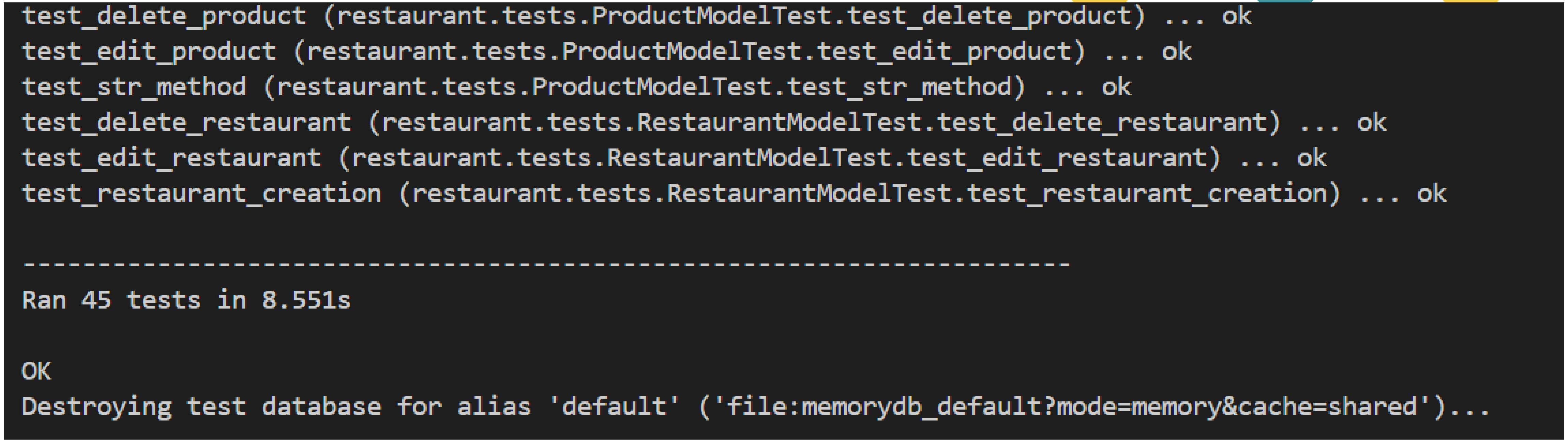
- Created Comprehensive Tests: Wrote tests for creating, editing, and deleting model instances.
- Tested Custom Validation: Ensured that custom validation logic is correctly enforced.
- Used Django's TestCase: Leveraged Django's TestCase class for setting up and running tests.

Steps used to write the tests

- Setup Method: Created initial data for the tests in the setUp method.
- Test Creation: Verified that model instances are created correctly.
- Test String Representation: Ensured that the __str__ method returns the expected string.
- Test Validation: Checked that invalid data raises ValidationError.
- Test Editing: Updated model instances and verified that the changes are saved.
- Test Deletion: Deleted model instances and confirmed that they no longer exist in the database.

Output

- Command Used : python manage.py test –verbosity=2



```
test_delete_product (restaurant.tests.ProductModelTest.test_delete_product) ... ok
test_edit_product (restaurant.tests.ProductModelTest.test_edit_product) ... ok
test_str_method (restaurant.tests.ProductModelTest.test_str_method) ... ok
test_delete_restaurant (restaurant.tests.RestaurantModelTest.test_delete_restaurant) ... ok
test_edit_restaurant (restaurant.tests.RestaurantModelTest.test_edit_restaurant) ... ok
test_restaurant_creation (restaurant.tests.RestaurantModelTest.test_restaurant_creation) ... ok

----------------------------------------------------------------------
Ran 45 tests in 8.551s

OK
Destroying test database for alias 'default' ('file:memorydb_default?mode=memory&cache=shared')...
```

Thank you