GeekBrains

+Django REST Framework

Тестирование API. Фабрики данных

На этом уроке

1. Узнаем, какие виды тестов для API существуют.
2. Научимся писать тесты для API на стороне сервера.
3. Научимся генерировать тестовые данные с помощью Mixer.

Оглавление

Тестирование API

- Введение
- APIRequestFactory
  - force_authenticate
  - Применение
- APIClient
  - Применение
- APISimpleTestCase
  - Применение
- APITestCase

Фабрика данных Mixer

- Введение
- Полное создание объекта
- Указание нужных полей
- Указание полей связанной модели

Итоги

Глоссарий

Дополнительные материалы

Используемые источники

Практическое задание

Тестирование API

Введение

Покрытие проекта тестами не обязательно, но это важная часть разработки и средство улучшения качества кода, архитектуры и системы в целом.

В Django-проектах есть встроенный механизм тестирования, который можно расширить, используя DRF. Так как DRF добавляет возможность создания REST API, тесты позволяют протестировать этот API.

В DRF есть несколько возможностей создания тестов для API, которые описаны в официальной документации. На этом занятии мы разберёмся, в каких случаях применяется тот или иной инструмент, и напишем тесты для нашего демонстрационного проекта.

Мы рассмотрим следующие основные классы для написания тестов:

- `APIRequestFactory`;
- `APIClient`;
- `APISimpleTestCase`;
- `APITestCase`.

Также познакомимся с библиотекой Mixer для быстрой генерации тестовых данных.

APIRequestFactory

Все тесты мы будем писать в файле tests.py приложения mainapp.

Внимание! *Повторяющийся код тестов можно вынести в метод setUp. Чтобы лучше усвоить материал, мы не будем этого делать и оставим в коде тестов некоторое дублирование.*

В файле tests.py напишем следующий код нашего первого теста:

```
import json
from django.test import TestCase
from rest_framework import status
from rest_framework.test import APIRequestFactory, force_authenticate,
APIClient, APISimpleTestCase, APITestCase
from mixer.backend.django import mixer
from django.contrib.auth.models import User
from .views import AuthorViewSet
from .models import Author, Book


class TestAuthorViewSet(TestCase):
```

```
    def test_get_list(self):
        factory = APIRequestFactory()
        request = factory.get('/api/authors/')
        view = AuthorViewSet.as_view({'get': 'list'})
        response = view(request)
        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

/library/mainapp/tests.py

Рассмотрим этот код по частям:

```
import json
from django.test import TestCase
from rest_framework import status
from rest_framework.test import APIRequestFactory, force_authenticate,
APIClient, APISimpleTestCase, APITestCase
from mixer.backend.django import mixer
from django.contrib.auth.models import User
from .views import AuthorViewSet
from .models import Author, Book
```

Сразу сделаем все импорты, чтобы не возвращаться к ним в будущем:

- модуль `json` нужен для чтения содержимого ответа от сервера;
- `TestCase` — базовый класс для создания Django-теста;
- `status` содержит константы для ответов сервера;
- `APIRequestFactory` — фабрика для создания запросов;
- `force_authenticate` — функция для авторизации пользователя;
- `APIClient` — клиент для удобной отправки REST-запросов;
- `APISimpleTestCase` — класс для создания простых test cases;
- `APIITestCase` — класс для создания test cases для REST API;
- `Mixer` — библиотека для генерации тестовых данных;
- `User` — модель пользователя;
- `AuthorViewSet` — view set для работы с моделью Author;
- `Author`, `Book` — модели автора и книги.

```
class TestAuthorViewSet(TestCase):

    def test_get_list(self):
        factory = APIRequestFactory()
        request = factory.get('/api/authors/')
        view = AuthorViewSet.as_view({'get': 'list'})
        response = view(request)
        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

Мы создали класс `TestAuthorViewSet`, который наследуется от `TestCase`, то есть это стандартный тест для Django-проектов.

Метод `test_get_list` будет проверять страницу со списком авторов.

Далее подробно рассмотрим код теста с использованием `APIRequestFactory`.

```
factory = APIRequestFactory()
request = factory.get('/api/authors/')
```

`APIRequestFactory` служит для создания запросов. Идея в том, чтобы не отправлять реальный запрос на сервер, а создать похожий запрос и передать его во `view`. Для этого требуется сначала создать объект класса `APIRequestFactory`, а затем с его помощью создать нужный тип запроса на определённый адрес.

Мы создали GET-запрос на адрес `/api/authors/`:

```
view = AuthorViewSet.as_view({'get': 'list'})
response = view(request)
self.assertEqual(response.status_code, status.HTTP_200_OK)
```

После создания запроса мы передаём его во `view`. Это позволит протестировать `view` независимо от других частей приложения.

После передачи запроса получаем ответ `response` и проверяем, что статус ответа `200 OK`.

Тесты запускаем стандартным способом:

```
python manage.py test
```

Напишем ещё один тест:

```
    def test_create_guest(self):
        factory = APIRequestFactory()
        request = factory.post('/api/authors/', {'name': 'Пушкин',
'birthday_year': 1799}, format='json')
        view = AuthorViewSet.as_view({'post': 'create'})
        response = view(request)
        self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)
```

Метод `test_create_guest` будет проверять запрос на создание автора, который отправляет неавторизованный пользователь.

Теперь с помощью объекта `factory` мы отправляем POST-запрос и передаём в него данные автора в формате JSON. Код ответа должен быть 401 UNAUTHORIZED, так как у гостя нет прав на создание данных, ему доступен только просмотр.

force_authenticate

Чтобы написать тест с использованием `APIRequestFactory` для авторизованного пользователя, можно использовать функцию `force_authenticate`. Напишем ещё один тест:

```
    def test_create_admin(self):
        factory = APIRequestFactory()
        request = factory.post('/api/authors/', {'name': 'Пушкин',
'birthday_year': 1799}, format='json')
        admin = User.objects.create_superuser('admin', 'admin@admin.com',
'admin123456')
        force_authenticate(request, admin)
        view = AuthorViewSet.as_view({'post': 'create'})
        response = view(request)
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
```

Он аналогичен предыдущему, но перед отправкой запроса во `view` мы создаём суперпользователя и передаём его вместе с запросом в `force_authenticate`:

```
force_authenticate(request, admin)
```

После этого статус ответа от сервера должен быть 201 Created, так как у администратора есть права на создание модели Author.

Применение

`APIRequestFactory` «подменяет» объект запроса, который мы потом передаём во `view`. Этот класс:

- возвращает объект запроса;
- не делает настоящий запрос;
- используется для изолированной проверки `view`;
- используется редко, обычно для нестандартных случаев.

APIClient

`APIClient` позволяет удобно отправлять запросы на определённый адрес и сразу получать ответ. Этот класс используется наиболее часто.

Добавим тест для проверки страницы с детальной информацией об авторе:

```
def test_get_detail(self):
        author = Author.objects.create(name='Пушкин', birthday_year=1799)
        client = APIClient()
        response = client.get(f'/api/authors/{author.id}/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

Для проверки детальной информации нам сначала нужно создать объект автора:

```
author = Author.objects.create(name='Пушкин', birthday_year=1799)
```

После этого создаём объект класса `APIClient` и отправляем GET-запрос с помощью этого объекта:

```
client = APIClient()
response = client.get(f'/api/authors/{author.id}/')
```

После этого проверяем статус ответа, он должен быть 200 OK.

По коду видно, что использовать `APIClient` удобнее, чем `APIRequestFactory`.

Добавим ещё один тест для редактирования автора неавторизованным пользователем:

```
    def test_edit_guest(self):
        author = Author.objects.create(name='Пушкин', birthday_year=1799)
        client = APIClient()
        response = client.put(f'/api/authors/{author.id}/', {'name':'Грин',
'birthday_year': 1880})
        self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)
```

Он похож на предыдущий, но мы отправляем PUT-запрос с данными и ожидаем код ответа 401 UNAUTHORIZED.

Для авторизации пользователя можно использовать метод `login` самого `APIClient`. Рассмотрим это на примере теста для редактирования автора авторизованным пользователем:

```
    def test_edit_admin(self):
        # 6.
        author = Author.objects.create(name='Пушкин', birthday_year=1799)
        client = APIClient()
        admin = User.objects.create_superuser('admin', 'admin@admin.com',
'admin123456')
        client.login(username='admin', password='admin123456')
        response = client.put(f'/api/authors/{author.id}/', {'name':'Грин',
```

```
'birthday_year': 1880})
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        author = Author.objects.get(id=author.id)
        self.assertEqual(author.name, 'Грин')
        self.assertEqual(author.birthday_year, 1880)
        client.logout()
```

Перед отправкой запроса мы создаём пользователя и вызываем метод `login` с его логином и паролем:

```
admin = User.objects.create_superuser('admin', 'admin@admin.com', 'admin123456')
client.login(username='admin', password='admin123456')
```

После этого все запросы будут отправляться от имени этого пользователя.

Теперь мы ожидаем код ответа 200 OK.

После проверки статуса ответа дополнительно проверяем, что данные в модели автора изменились:

```
author = Author.objects.get(id=author.id)
self.assertEqual(author.name, 'Грин')
self.assertEqual(author.birthday_year, 1880)
```

Метод `logout` в классе `APIClient` позволяет совершить выход:

```
client.logout()
```

Применение

`APIClient` — класс для удобной отправки REST-запросов. Этот класс:

- отправляет запрос;
- обычно используется для большинства тестов.

APISimpleTestCase

Далее рассмотрим расширение класса `TestCase`. Класс `APISimpleTestCase` позволяет создавать класс для тестирования, не связанный с базой данных. Рассмотрим следующий пример:

```
class TestMath(APISimpleTestCase):
```

```
    def test_sqrt(self):
        import math
        self.assertEqual(math.sqrt(4), 2)
```

/library/mainapp/tests.py

Этот пример никак не связан с проектом и просто служит для демонстрации работы `APISimpleTestCase`. Сначала мы создаём наш тестовый класс на его основе с помощью наследования:

```
class TestMath(APISimpleTestCase):
```

После этого пишем тестовый метод `test_sqrt` для проверки работы функции `sqrt` из пакета `math`.

Применение

`APISimpleTestCase` применяется очень редко: в случаях, когда тест не связан с базой данных. Этот класс:

- не использует базу данных;
- удобен для тестирования внутренних функций;
- быстро исполняется.

APITestCase

`APITestCase` уже содержит в себе экземпляр класса `APIClient`, поэтому наиболее удобен для решения большинства задач.

С помощью `APITestCase` мы напишем тесты для `BookViewSet`. Перед написанием тестов внесём некоторые изменения в наш проект.

В файле serializers.py добавим новый сериализатор. Код файла приобретает следующий вид:

```
from rest_framework import serializers
from .models import Author, Book

class AuthorSerializer(serializers.ModelSerializer):
    class Meta:
        model = Author
        fields = '__all__'

class BookSerializerBase(serializers.ModelSerializer):
    class Meta:
```

```
        model = Book
        fields = '__all__'

class BookSerializer(serializers.ModelSerializer):
    author = AuthorSerializer()
    class Meta:
        model = Book
        fields = '__all__'
```

/library/mainapp/serializers.py

Мы добавили `BookSerializerBase` — второй сериализатор для модели `Book`. Дело в том, что в `BookSerializer` мы объявили поле `author` как `author = AuthorSerializer()`. Это удобно при выводе данных, так как данные автора будут представлены словарём, а не его id, но это неудобно при сохранении данных. Поэтому `BookSerializerBase` будет для сохранения данных, а `BookSerializer` — для вывода данных.

Далее изменим код в файле views.py следующим образом:

```
from rest_framework import viewsets, permissions
from .models import Author, Book
from .serializers import AuthorSerializer, BookSerializer, BookSerializerBase


class AuthorViewSet(viewsets.ModelViewSet):
    serializer_class = AuthorSerializer
    queryset = Author.objects.all()


class BookViewSet(viewsets.ModelViewSet):
    # permission_classes = [permissions.IsAuthenticated]
    serializer_class = BookSerializer
    queryset = Book.objects.all()

    def get_serializer_class(self):
        if self.request.method in ['GET']:
            return BookSerializer
        return BookSerializerBase
```

/library/mainapp/views.py

В `BookViewSet` мы убрали `permission_classes`, они были нам нужны на предыдущем занятии для проверки авторизации.

Также мы переопределили метод `get_serializer_class`. Этот метод используется для выбора сериализатора. Теперь при GET-запросах на просмотр данных мы будем использовать `BookSerializer`, а для всех остальных (на изменение данных) `BookSerializerBase`.

Всё готово для написания тестов. Создадим test case и первый тест для получения списка книг:

```
class TestBookViewSet(APITestCase):

    def test_get_list(self):
        response = self.client.get('/api/books/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

/library/mainapp/tests.py

Обратите внимание на эту строчку:

```
response = self.client.get('/api/books/')
```

При использовании `APITestCase` вместо `TestCase` в экземпляре класса у нас сразу есть свойство `client`. Это `APIClient`, который мы рассматривали ранее.

Напишем ещё один тест для проверки редактирования книги суперпользователем:

```
    def test_edit_admin(self):
        author = Author.objects.create(name='Пушкин', birthday_year=1799)
        book = Book.objects.create(name='Пиковая дама', author=author)
        admin = User.objects.create_superuser('admin', 'admin@admin.com',
'admin123456')
        self.client.login(username='admin', password='admin123456')
        response = self.client.put(f'/api/books/{book.id}/', {'name': 'Руслан и
Людмила', 'author': book.author.id})
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        book = Book.objects.get(id=book.id)
        self.assertEqual(book.name,'Руслан и Людмила')
```

Сначала мы создаём тестовые данные для книги, автора и пользователя:

```
author = Author.objects.create(name='Пушкин', birthday_year=1799)
book = Book.objects.create(name='Пиковая дама', author=author)
admin = User.objects.create_superuser('admin', 'admin@admin.com', 'admin123456')
```

После этого логиним пользователя:

```
self.client.login(username='admin', password='admin123456')
```

Далее отправляем PUT-запрос и проверяем статус ответа 200 OK:

```
response = self.client.put(f'/api/books/{book.id}/', {'name': 'Руслан и
Людмила', 'author': book.author.id})
        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

После этого получаем объект книги и проверяем, что данные изменились:

```
book = Book.objects.get(id=book.id)
self.assertEqual(book.name,'Руслан и Людмила')
```

Фабрика данных Mixer

Введение

В последнем тесте мы создали тестовые данные для книги и автора. Посмотрим на этот кусок кода ещё раз:

```
author = Author.objects.create(name='Пушкин', birthday_year=1799)
book = Book.objects.create(name='Пиковая дама', author=author)
```

Обратите внимание, что для создания объекта книги нам пришлось создать связанную с ним модель автора и заполнить её данными. Если бы модель автора тоже зависела от другой модели, нам бы пришлось создать уже три объекта.

Важный момент: для самого теста начальные данные книги и автора нам не важны, мы проверяем только изменённые данные:

```
self.assertEqual(book.name,'Руслан и Людмила')
```

Получается, что неважно, какими данными мы заполним объекты книги и автора, лишь бы они были созданы в базе.

Для быстрой генерации тестовых данных служит библиотека Mixer. Она позволяет создавать объекты и заполнять их тестовыми данными, а также создавать связанные объекты моделей.

Для работы с библиотекой Mixer установим её:

```
pip install mixer
```

Полное создание объекта

Перепишем последний тест с помощью библиотеки Mixer:

```
def test_edit_mixer(self):
        book = mixer.blend(Book)
        admin = User.objects.create_superuser('admin', 'admin@admin.com',
'admin123456')
        self.client.login(username='admin', password='admin123456')
        response = self.client.put(f'/api/books/{book.id}/', {'name': 'Руслан и
Людмила', 'author': book.author.id})
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        book = Book.objects.get(id=book.id)
        self.assertEqual(book.name,'Руслан и Людмила')
```

Код теста остался прежним, но создание тестовых данных свелось к одной простой строке:

```
book = mixer.blend(Book)
```

При таком вызове Mixer сам создаст объект, при этом:

- сам определит типы полей модели и заполнит их соответствующим типом случайных данных;
- сам определит все связанные модели и создаст соответствующие объекты со случайными данными.

Указание нужных полей

Иногда нам нужно указать некоторые значения полей модели, а остальные оставить случайными. Чтобы рассмотреть эту возможность, напишем тест на получение данных одной книги:

```
    def test_get_detail(self):
        book = mixer.blend(Book, name='Алые паруса')
        response = self.client.get(f'/api/books/{book.id}/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        response_book = json.loads(response.content)
        self.assertEqual(response_book['name'], 'Алые паруса')
```

При создании объекта мы указали, что `name` (название книги) равно `'Алые паруса'`. Mixer создаст объект с этим именем, а все остальные данные создаст случайно.

Далее мы проверяем код ответа 200 OK и название книги, которые сами задали при создании объекта.

Указание полей связанной модели

Иногда вместо указания поля самой модели нам требуется указать поле связанной модели.

Для демонстрации напишем ещё один тест:

```
    def test_get_detail_author(self):
        book = mixer.blend(Book, author__name='Грин')
        response = self.client.get(f'/api/books/{book.id}/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        response_book = json.loads(response.content)
        self.assertEqual(response_book['author']['name'], 'Грин')
```

Этот тест похож на предыдущий, мы проверяем отображение одного объекта модели. Но в этом случае при создании мы указываем имя автора книги, используя двойное подчёркивание:

```
book = mixer.blend(Book, author__name='Грин')
```

И в конце проверяем это имя:

```
self.assertEqual(response_book['author']['name'], 'Грин')
```

Итоги

На этом занятии мы рассмотрели, как создавать тесты для REST API с помощью DRF и как удобно создавать тестовые данные с помощью Mixer.

Глоссарий

`APIRequestFactory` — класс для генерации запросов.

`APIClient` — класс для удобной отправки REST-запросов.

`APISimpleTestCase` — класс для создания тестов, не связанных с базой данных.

`APITestCase` — класс для удобного написания тестов для REST API.

`Mixer` — библиотека для создания тестовых данных.

Дополнительные материалы

1. Mixer — официальная документация.
2. Примеры использования Mixer.
3. Тестирование DRF.

Используемые источники

1. Mixer — официальная документация.
2. Примеры использования Mixer.
3. Тестирование DRF.
4. Бэк Кент «Разработка через тестирование».

Практическое задание

Тестирование API. В этой самостоятельной работе мы тренируем умения:

- тестировать REST API;
- создавать тестовые данные.

Смысл: писать тесты для проектов с DRF.

Последовательность действий

1. Написать минимум один тест для API, используя `APIRequestFactory`.
2. Написать минимум один тест для API, используя `APIClient`.
3. Написать минимум один тест для API, используя `APITestCase`.
4. Данные для тестов удобно создавать, используя mixer.
5. * Написать минимум один Live test.