

Тестирование Django-приложений с использованием Pytest



Адилет
Асанкожоев



Адилет Асанкожоев

Python-разработчик в Makers.kg





План занятия

1. [Django и тестирование](#)
2. [Настройка Pytest для работы с Django](#)
3. [Организация тестов](#)
4. [Примеры](#)
5. [Параметризация входных данных](#)
6. [Написание фикстур](#)
7. [Фабрики для моделей](#)
8. [Покрывание кода тестами](#)



Django и тестирование

Тестирование – неотъемлемая часть разработки.

Во многих компаниях есть требования по покрытию кода тестами. Например, нельзя, чтобы покрытие тестами опускалось менее 80% от числа строк кода в проекте.

Повторение



Что такое pytest?
Чем pytest отличается от unittest?

Django и тестирование

В Django есть стандартный модуль для тестирования приложений, но он использует `unittest`.

<https://docs.djangoproject.com/en/3.1/topics/testing/overview/>

`pytest` значительно мощнее и удобнее, рекомендуется использовать его. При этом вы не теряете доступные библиотеки для тестирования Django, заменяется лишь стандартный test runner.



Настройка Pytest для работы с Django

Установка

```
$ pip install pytest  
$ pip install pytest-django
```

Совет по интеграции **pytest** с Pycharm:

Для удобного запуска из Pycharm, укажите **pytest** как **Default test runner** в настройках проекта:

<https://www.jetbrains.com/help/pycharm/pytest.html>

Конфигурация

Заведите в корне проекта файл `pytest.ini` и укажите там, какой файл настроек является основным для тестов:

```
[pytest]
DJANGO_SETTINGS_MODULE = django_testing.settings
```

Документация: <https://pytest-django.readthedocs.io/en/latest/tutorial.html>



Организация тестов

Организация тестов

Заведем директорию `tests` на уровне проекта:

```
tests/  
  conftest.py  
  orders/  
    __init__.py  
    test_api.py
```

Обратите внимание, что не нужно добавлять файл `__init__.py` в директорию `tests`, если вы располагаете тесты в отдельном модуле.

Подробнее:

<https://docs.pytest.org/en/stable/pythonpath.html#standalone-test-module-s-conftest-py-files>



Название тестов

Также обратите внимание на название тестов. По умолчанию, `pytest` собирает функции и классы, начинающиеся на `Test` / `test`. Это можно переопределить в конфиге `pytest`.

Логика внутри теста

При написании теста следует придерживаться определенной структуры.

Один из общепринятых подходов – **Arrange, Act, Assert**.

- **Arrange**: готовим данные;
- **Act**: совершаем действие, которое хотим протестировать;
- **Assert**: проверяем результат.

<https://github.com/testdouble/contributing-tests/wiki/Arrange-Act-Assert>



Примеры

Пример #1: запуск теста

Проверим, что мы все настроили правильно:

```
def test_something():  
    assert True
```

Ожидаемый результат:

```
$ pytest  
  
===== test session starts =====  
...  
django: settings: api_first_steps.settings (from ini)  
plugins: django-3.10.0  
collected 1 item  
  
tests/orders/test_api.py . [100%]  
  
===== 1 passed in 0.26s =====
```

Пример #2: запуск теста с API

```
# views.py
@api_view(['GET'])
def ping_view(request):
    return Response({"status": "Ok"})
```

```
# test_api.py
from rest_framework.test import APIClient

def test_ping():
    client = APIClient() # создание клиента для запросов
    url = reverse("ping")
    resp = client.get(url)
    assert resp.status_code == HTTP_200_OK
    resp_json = resp.json()
    assert resp_json["status"] == "Ok"
```


Пример #3: запуск теста, требующего базу данных

Для того, чтобы использовать базу данных, необходим декоратор `@pytest.mark.django_db`

```
from orders.models import Product

@pytest.mark.django_db
def test_product_create():
    product = Product.objects.create(name="Test", unit_price=50)
    assert product.id
    assert product.name == "Test"
```

Примечание

Проверять конкретные **id** считается плохой практикой – так как при автоматическом создании вы не знаете, какой именно **id** подставит база данных, так как они инкрементируются автоматически.

Например, в предыдущем тесте нельзя писать

```
assert product.id == 1
```

Потому что если перед этим в каком-то тесте создавался **Product**, то **id** может отличаться от запуска к запуску.

Пример #4: запуск теста с API и наполнением данных

```
class ProductViewSet(viewsets.ModelViewSet):  
    queryset = Product.objects.all()  
    serializer_class = ProductSerializer
```

```
router = routers.DefaultRouter()  
router.register("products", order_views.ProductViewSet,  
    basename="products")
```

```
urlpatterns = [  
    path('api/v1/', include(router.urls)),  
]
```


Мы задаем **basename**, чтобы можно было использовать **reverse** для получения адреса, на который необходимо сделать запрос.

Итоговый тест:

```
@pytest.mark.django_db
def test_products_list():
    client = APIClient()
    url = reverse("products-list")
    products = Product.objects.bulk_create([
        Product(unit_price=10, name='Test 1'),
        Product(unit_price=20, name='Test 2'),
    ])

    resp = client.get(url)

    assert resp.status_code == HTTP_200_OK
    resp_json = resp.json()
    assert resp_json["results"]
    results = resp_json["results"]
    assert len(results) == 2
    # получаем список id в результате и сверяем с созданными
    results_ids_set = {result["id"] for result in results}
    expected_ids_set = {product.id for product in products}
    assert results_ids_set == expected_ids_set
```



Параметризация входных данных

Параметризация входных данных

Иногда возникает потребность воспроизвести существующий тест, но с другими параметрами. Например, мы хотим протестировать разные параметры фильтров или граничные условия.

Опишем ограничения в модели:

```
class Product(models.Model):
    """Продукт в ассортименте."""
    ...
    unit_price = models.DecimalField(
        verbose_name="Цена за единицу",
        decimal_places=2,
        max_digits=16,
        validators=[
            validators.MinValueValidator(0),
            validators.MaxValueValidator(10000),
        ]
    )
```



И будем использовать **ModelSerializer**:

```
class ProductSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Product  
        fields = ("id", "name", "unit_price",)
```



Протестируем создание продукта через API:

```
@pytest.mark.django_db
def test_product_create_validation():
    client = APIClient()
    url = reverse("products-list")
    product_payload = {
        "name": "Test",
        "unit_price": '400',
    }
    resp = client.post(url, product_payload)
    assert resp.status_code == HTTP_201_CREATED
```

Как проверить с другой ценой за единицу (отрицательная цена, цена больше максимума)? Писать новый тест или усложнять логику внутри теста не хочется, это разбивает **Act-Arrange-Assert** паттерн и создает много дублирующегося кода.

Использование parametrize

Выход – использовать **parametrize**:

```
@pytest.mark.parametrize(
    ["unit_price", "expected_status"],
    (
        ("400", HTTP_201_CREATED),
        ("-100", HTTP_400_BAD_REQUEST),
        ("100000000", HTTP_400_BAD_REQUEST),
    )
)
@pytest.mark.django_db
def test_product_create_validation(unit_price, expected_status):
    client = APIClient()
    url = reverse("products-list")
    product_payload = {
        "name": "Test",
        "unit_price": unit_price,
    }
    resp = client.post(url, product_payload)
    assert resp.status_code == expected_status
```



Написание фикстур

Объявление фикстуры

Фикстуры очень удобны для переиспользуемого кода. Если вы видите, что код часто повторяется, является вспомогательным и перегружает тест, вынесите его в фикстуру.

На примере клиента. Заводим фикстуру в `conftest.py`:

```
import pytest
from rest_framework.test import APIClient

@pytest.fixture
def api_client():
    """Фикстура для клиента API."""
    return APIClient()
```

Использование фикстуры в тесте

```
def test_ping(api_client):  
    url = reverse("ping")  
    resp = api_client.get(url)  
    assert resp.status_code == HTTP_200_OK  
    resp_json = resp.json()  
    assert resp_json["status"] == "Ok"
```



Фабрики для моделей

Фабрики для моделей

Если у модели есть обязательные поля, то приходится каждый раз передавать их все, даже когда они не влияют на логику теста. Это усложняет код теста, поэтому для таких целей лучше использовать специальные библиотеки с фабриками для Django.

Самые популярные:

- FactoryBoy: <https://factoryboy.readthedocs.io/en/latest/>
- Model Bakery: https://github.com/model-bakers/model_bakery

Пример использования фабрики

Рассмотрим на примере Model Bakery, так как для неё не требуется дополнительная конфигурация.

Установка:

```
$ pip install model_bakery
```

Заведём фикстуру для фабрики (это полезно делать, так как вам может потребоваться подготовительная работа до/после создания моделей через фабрику):

```
from model_bakery import baker

@pytest.fixture
def product_factory():
    def factory(**kwargs):
        return baker.make("Product", **kwargs)
    return factory
```

Результат

Тест существенно упростился:

```
@pytest.mark.django_db
def test_products_list(api_client, product_factory):
    url = reverse("products-list")
    products = product_factory(_quantity=2)
    resp = api_client.get(url)
    assert resp.status_code == HTTP_200_OK
    resp_json = resp.json()
    assert resp_json["results"]
    results = resp_json["results"]
    assert len(results) == 2
```




Поккрытие кода тестами



Тестирование как метрика

Покрытие кода тестами считается как отношение количества строк кода, которые запускаются в тесте, к общему количеству строк кода.

Покрытие кода тестами – полезная метрика, которую используют для того чтобы понять, какое количество кода протестировано, а какое нет.

Во многих командах договариваются держать покрытие тестами на определенном уровне. Например, нельзя опускать его ниже 80%. Это можно валидировать на этапе открытия пулл-реквестов с новым кодом.

Установка библиотеки coverage

coverage — библиотека для Python, которая позволяет считать покрытие тестами Python-кода.

Можно установить как плагин для pytest:

```
$ pip install pytest-cov
```

Затем опишем параметры для **coverage** в корне проекта в файле **.coveragerc**:

```
[run]  
omit = tests/*
```

omit указывает, что не нужно считать покрытие тестами внутри модуля с тестами.

Запуск

Запустим `pytest` с генерацией отчета о покрытии в `html`:

```
pytest --cov-report=html --cov=your_project
```

Отчет будет лежать в директории `htmlcov` после того, как `pytest` закончит выполнение.

Итоги

Сегодня на занятии мы научились:

- писать тесты для Django, используя pytest;
- подставлять разные входные параметры в тесты;
- использовать фабрики моделей для упрощения создания сущностей;
- использовать фикстуры для переиспользования кода между тестами;
- считать покрытие кода тестами.



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаём в чате Slack!
- Задачи можно сдавать по частям.
- Зачёт по домашней работе проставляется после того, как приняты **все задачи**.

**Задавайте вопросы и
пишите отзыв о лекции!**

 [Адилет Асанкожоев](#)

Адилет Асанкожоев