
# Django Class Notes

Clarusway

## Testing in Django Rest Framework

Nice to have VSCode Extentions:

- Djaneiro - Django Snippets

Needs

- Python
- pip
- virtualenv

Summary

- Testing in Django Rest Framework
- Spin up the project
- Tests with Django Rest Framework classes
    - Test folder structure and test labels
    - APITestCase class
    - Testing list API endpoint
    - Creating test requests with the APIRequestFactory
    - Responses
        - Creating responses
    - Assertions
    - Testing create API endpoint
    - Status Codes
    - Testing update API endpoint

    - Testing delete API endpoint
- Pytest
    - Testing reservation views
    - Test client object
    - (Optional) Creating testing objects using packages
- Seperate testing database
- TDD vs. Testing After
- Code Coverage
- Next steps

# Testing in Django Rest Framework

REST framework includes a few helper classes that extend Django's existing test framework, and improve support for making API requests.

Automated testing is an extremely useful bug-killing tool for the modern Web developer. You can use a collection of tests – a test suite – to solve, or avoid, a number of problems:

- When you're writing new code, you can use tests to validate your code works as expected.
- When you're refactoring or modifying old code, you can use tests to ensure your changes haven't affected your application's behavior unexpectedly.

With Django's test-execution framework and assorted utilities, you can simulate requests, insert test data, inspect your application's output and generally verify your code is doing what it should be doing.

# Spin up the project

- (Optional) Shorten your powershell terminal prompt:

```
Function Prompt { "Code: " }
```

- Create a working directory, name it as you wish, cd to new directory.
- Pull the latest changes from Clarusway repo, copy materials, paste them to your newly created folder.
- Create virtual environment as a best practice:

```
python3 -m venv env # for Windows or
python -m venv env # for Windows
virtualenv env # for Mac/Linux or;
virtualenv env -p python3 # for Mac/Linux
```

- Activate scripts:

```
.\env\Scripts\activate  # for Windows
source env/bin/activate  # for MAC/Linux
```

- See the (env) sign on the left side of your command prompt.
- Install dependencies:

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

- Create .env file on root directory. We will collect our variables in this file. Those varibles may change with project.

```
SECRET_KEY=django2-insecure-...
SQL_DATABASE=flightApp
SQL_USER=postgres
SQL_PASSWORD=postgres
SQL_HOST=localhost
SQL_PORT=5432
ENV_NAME=DEV
DJANGO_LOG_LEVEL=INFO
```

- Migrate db's:

```
python manage.py migrate
```

- Check your project if it's installed correctly:

```
python manage.py runserver
py -m manage.py runserver
```

- Create a super user:

```
python manage.py createsuperuser
```

# Tests with Django Rest Framework classes

- First we need to create a class in flight/tests.py
- There may be some alternatives to create folder structure for tests.

## Test folder structure and test labels

The default startapp template creates a tests.py file in the new application. This might be fine if you only have a few tests, but as your test suite grows you will likely want to restructure it into a tests package so you can split your tests into different submodules such as:

- test_models.py,
- test_views.py,
- test_serializers.py, etc.

Feel free to pick whatever organizational scheme you like.

## APITestCase class

REST framework includes the following test case classes, that mirror the existing Django's test case classes, but use APIClient instead of Django's default Client.

- APISimpleTestCase
- APITransactionTestCase
- APITestCase
- APILiveServerTestCase

API classes provide some API specific methods like;

- `.format` method to more simply send encoded data.
- `.force_authenticate` method to provide authentication credentials on every request.

Otherwise there's no difference with regular TestCase class.

The docs note that we shouldn't use APISimpleTestCase when doing any testing with the database.

The APITestCase wraps the tests with 2 atomic() blocks, one for the whole test class and one for each test within the class. This essentially stops tests from altering the database for other tests as the transactions are rolled back at the end of each test. By having this second atomic() block around the whole test class, specific database transaction behaviour can be hard to test and hence you'd want to drop back to using APITransactionTestCase.

We will use APITestCase in our tests.

## Testing list API endpoint

```
from rest_framework.test import APITestCase
from rest_framework.test import APIRequestFactory
from django.contrib.auth.models import AnonymousUser
from .views import FlightView

# End the class name with TestCase, nothing special with the prefix name
class FlightTestCase(APITestCase):

    # Before the tests run we want to set the environment up.
    # The setUp is part of initialization, this method will get called
```

```
    # before every test function which you are going to write in this
    # test case class.
    def setUp(self):
        # Every test needs access to the request factory.
        # APIRequestFactory provides a way to generate a request instance
        self.factory = APIRequestFactory()

    # Give a descriptive name to your test functions
    # This is the test case method. The test case method should always
    # start with characters `test`.
    def test_flight_list_as_non_authenticated_user(self):
        request = self.factory.get('/flight/flights/')
        # You can simulate an anonymous user by setting
        # request.user to an AnonymousUser instance.
        request.user = AnonymousUser()
        # While using viewsets, we need to bind this viewset into separate
        # views like {'get': 'list'}, {'get': 'retrieve'},
        # {'post': 'create'}, {'delete': 'destroy'}, {'put': 'update'}
        response = FlightView.as_view({'get': 'list'})(request)
        self.assertEqual(response.status_code, 200)from rest_framework.test import
APITestCase
from rest_framework.test import APIRequestFactory
from django.contrib.auth.models import AnonymousUser
from .views import FlightView

# End the class name with TestCase, nothing special with the prefix name
class FlightTestCase(APITestCase):

    # Before the tests run we want to set the environment up.
    # The setUp is part of initialization, this method will get called
    # before every test function which you are going to write in this
    # test case class.
    def setUp(self):
        # Every test needs access to the request factory.
        # APIRequestFactory() allows you to create requests with any http method,
        # which you can then pass on to any view method and compare responses.
        self.factory = APIRequestFactory()

    # Give a descriptive name to your test functions
    # This is the test case method. The test case method should always
    # start with characters `test`.
    def test_flight_list_as_non_authenticated_user(self):
        request = self.factory.get('/flight/flights/')
        # You can simulate an anonymous user by setting
        # request.user to an AnonymousUser instance.
        request.user = AnonymousUser()
        # While using viewsets, we need to bind this viewset into separate
        # views like {'get': 'list'}, {'get': 'retrieve'},
        # {'post': 'create'}, {'delete': 'destroy'}, {'put': 'update'}
        response = FlightView.as_view({'get': 'list'})(request)
        # We will discuss response object and assertions later.
        self.assertEqual(response.status_code, 200)
```

- Time to run the test!

Test discovery is based on the unittest module's built-in test discovery. By default, this will discover tests in any file named `test*.py` under the current working directory.

You can specify particular tests to run by supplying any number of `test labels` to `manage.py test`. Each test label can be a full Python dotted path to a package, module, TestCase subclass, or test method. For instance:

```
# Run all the tests found within the 'animals' app package
$ ./manage.py test animals

# Run all the tests in the animals.tests py module
$ ./manage.py test animals.tests

# Run just one test case class
$ ./manage.py test animals.tests.AnimalTestCase

# Run just one test method function
$ ./manage.py test animals.tests.AnimalTestCase.test_animals_can_speak
```

```
python manage.py test
```

- It is possible to run the test again with verbose flag;

```
# By default verbose level is 1, you can increase it 2 or 3
python manage.py test -v 2
```

## Creating test requests with the APIRequestFactory

The RequestFactory shares the same API as the test client. However, instead of behaving like a browser, the RequestFactory provides a way to generate a request instance that can be used as the first argument to any view. This means you can test a view function the same way as you would test any other function – as a black box, with exactly known inputs, testing for specific outputs.

The API for the RequestFactory is a slightly restricted subset of the test client API:

- It only has access to the HTTP methods get(), post(), put(), delete(), head(), options(), and trace().
- These methods accept all the same arguments except for follow. Since this is just a factory for producing requests, it's up to you to handle the response.
- It does not support middleware. Session and authentication attributes must be supplied by the test itself if required for the view to function properly.

## Responses

REST framework supports HTTP content negotiation by providing a Response class which allows you to return content that can be rendered into multiple content types, depending on the client request.

Using the Response class simply provides a nicer interface for returning content-negotiated Web API responses, that can be rendered to multiple formats.

### Creating responses

```
Response(data, status=None, template_name=None, headers=None, content_type=None)
```

Unlike regular HttpResponse objects, you do not instantiate Response objects with rendered content. Instead you pass in unrendered data, which may consist of any Python primitives.

The renderers used by the Response class cannot natively handle complex datatypes such as Django model instances, so you need to serialize the data into primitive datatypes before creating the Response object.

You can use REST framework's Serializer classes to perform this data serialization, or use your own custom serialization.

Arguments:

- **data**: The serialized data for the response.
- **status**: A status code for the response. Defaults to 200. See also status codes.
- **template_name**: A template name to use if HTMLRenderer is selected.
- **headers**: A dictionary of HTTP headers to use in the response.
- **content_type**: The content type of the response. Typically, this will be set automatically by the renderer as determined by content negotiation, but there may be some cases where you need to specify the content type explicitly.

## Assertions

The TestCase class provides several assert methods to check for and report failures.

For more info about assertions on django:

- Assertions
- The list of the most commonly used assert methods in python

## Testing create API endpoint

The APIRequestFactory class supports an almost identical API to Django's standard RequestFactory class. This means that the standard .get(), .post(), .put(), .patch(), .delete(), .head() and .options() methods are all available.

- Let's try post method to test creation of a flight. Write tests ensuring only staff users can create a flight. First test an anonymous user can not create a flight:

```
    def test_flight_create_as_anonymous_user(self):
        request = self.factory.post('/flight/flights/')
        request.user = AnonymousUser()
        response = FlightView.as_view({'post': 'create'})(request)
```

```
        # HTTP_401_UNAUTHORIZED
        self.assertEqual(response.status_code, 401)
        self.assertTrue(status.is_success(response.status_code))
```

## Status Codes

Using bare status codes in your responses isn't recommended. REST framework includes a set of named constants that you can use to make your code more obvious and readable.

- Next, test a logged in user can not create a flight:

```
from rest_framework.test import force_authenticate
from rest_framework.authtoken.models import Token
from django.contrib.auth.models import User

    def setUp(self):
        # Create a test user and token:
        self.user = User.objects.create_user(
            username='jacob',
            email='jacob@mail.com',
            password='secret123#'
            )
        self.token = Token.objects.get(user=self.user)

    def test_flight_create_as_logged_in_user(self):
        # We need to pass token to our request to generate a logged in user
        request = self.factory.post('/flight/flights/', HTTP_AUTHORIZATION='Token
{}'.format(self.token))
        force_authenticate(request, user=self.user)
        response = FlightView.as_view({'post': 'create'})(request)
        # HTTP_403_FORBIDDEN
        self.assertEqual(response.status_code, 403)
```

- And test if staff user can create a flight:

```
from rest_framework import status

    def test_flight_create_as_staff_user(self):
        # Without sending data it gives 400 Bad Request
        data = {
            "flight_number": "TK100",
            "airlines": "THY",
            "departure_city": "Antalya",
            "arrival_city": "Amsterdam",
            "date_of_departure": "2022-12-03",
            "etd": "15:27:22"
        }
        request = self.factory.post('/flight/flights/', data,
HTTP_AUTHORIZATION='Token {}'.format(self.token))
```

```
        # And make this user a staff, do not forget to save it
        self.user.is_staff = True
        self.user.save()
        force_authenticate(request, user=self.user)
        response = FlightView.as_view({'post': 'create'})(request)
        self.assertEqual(response.status_code, 201)
        # Another way to check everything is ok
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
```

## Testing update API endpoint

- Test if you can update a flight:

```
from .models import Flight


    def setUp(self):
        # To update a flight, we need to have one.
        self.flight = Flight.objects.create(
            id=1,
            flight_number="AA100",
            airlines="AHY",
            departure_city="Antalya",
            arrival_city="Amsterdam",
            date_of_departure="2022-12-01",
            etd="15:00:00")


def test_flight_update_as_staff_user(self):
        # Without sending data it gives 400 Bad Request
        data = {
            "flight_number": "UK100",
            "airlines": "UHY",
            "departure_city": "Untalya",
            "arrival_city": "Umsterdam",
            "date_of_departure": "2022-12-03",
            "etd": "15:27:22"
        }
        request = self.factory.put('/flight/flights/1/', data,
HTTP_AUTHORIZATION='Token {}'.format(self.token))
        self.user.is_staff = True
        self.user.save()
        force_authenticate(request, user=self.user)
        response = FlightView.as_view({'put': 'update'})(request, pk='1')
        self.assertEqual(response.status_code, 200)
        self.assertEqual(Flight.objects.count(), 1)
        self.assertEqual(Flight.objects.get().flight_number, 'UK100')
```

## Testing delete API endpoint

- Lastly test delete function.

```
    def test_flight_delete_as_staff_user(self):
        request = self.factory.delete('/flight/flights/1/',
HTTP_AUTHORIZATION='Token {}'.format(self.token))
        self.user.is_staff = True
        self.user.save()
        force_authenticate(request, user=self.user)
        response = FlightView.as_view({'delete': 'destroy'})(request, pk='1')
        self.assertEqual(response.status_code, 204)
        self.assertEqual(Flight.objects.count(), 0)
```

# Pytest

The pytest framework makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries.

- To use pytest first, install it:

```
pip install pytest
# pytest-django is a plugin for pytest that provides a set of useful
# tools for testing Django applications and projects.
pip install pytest-django
pip freeze > requirements.txt
```

- To set up pytest-django, create a new file named `pytest.ini` at the same place with manage.py file.

```
[pytest]
DJANGO_SETTINGS_MODULE = main.settings
# -- recommended but optional:
python_files = tests.py test_*.py *_tests.py
```

- Ready to test with;

```
pytest
```

- At this time, it is logical to seperate test files for pytest. Create a folder under flight app named tests. Create a new empty file named `__init__.py` which is showing this is a python package. Create another file named `test_pytest.py` for our new test functions. And, copy existing tests.py file by renaming as `test_django.py` under tests folder.

- Let us write a basic test function;

```
def test_initial():
  assert True
```

- Test! See pytest also compatible with Django default test classes.
- Specific test files or directories can be selected by specifying the test file names directly on the command line:

```
pytest flight/tests/test_pytest.py
```

- Let's write a failing test function;

```
def test_failing():
  assert 1 == 2
```

- Test and see it is failing. It is a lot of information for a tiny error but it helps when you have a lot of code.

## Testing reservation views

Some programmers follow a discipline called “test-driven development”; they actually write their tests before they write their code. This might seem counter-intuitive, but in fact it’s similar to what most people will often do anyway: they describe a problem, then create some code to solve it.

- Delete first two tests and write a test for reservation view:

```
from rest_framework.authtoken.models import Token
from django.contrib.auth.models import User
import pytest
from rest_framework.test import APIClient

class TestReservationEndpoints():

  # This is used to mark a test function as requiring the database.
  # It will ensure the database is set up correctly for the test.
  # Each test will run in its own transaction which will be rolled back
  # at the end of the test.
  @pytest.mark.django_db
  def test_get_reservation_view_as_authenticated_user(self):
    '''
    This test ensures an authorized user can see reservations.
    '''

    user = User.objects.create_user(
              username='jacob',
              email='jacob@mail.com',
```

```
                password='secret123#'
                )
    token = Token.objects.get(user=user)
    client = APIClient()
    # The credentials method is appropriate for testing APIs that require
    # authentication headers, such as basic authentication, and simple
    # token authentication schemes.
    client.credentials(HTTP_AUTHORIZATION='Token ' + token.key)
    # client.force_authenticate(user=user)
    response = client.get(path='/flight/reservations/')
    assert 200 == response.status_code
    assert [] == response.data
```

- `@pytest.mark.django_db` decorator tells pytest to enable db access.
- Adding a docstring to our test functions is also logical.
- Writing expected response first is much more elegant way.
- Create fixture for the code pieces you are using more than once.

```
@pytest.fixture
def logged_user(client):
  user = User.objects.create_user(
            username='jacob',
            email='jacob@mail.com',
            password='secret123#'
            )
  client.login(username=user.username, password='password')
  return user
```

- And use fixture func name next to client.

## Test client object

- The test `client` is a Python class that acts as a dummy web browser, allowing you to test your views and interact with your Django-powered application programmatically. The test client
- The test client does not require the web server to be running.
- When retrieving pages, remember to specify the path of the URL, not the whole domain.
- Run the test

## (Optional) Creating testing objects using packages

- (Optional) We can use factory boy package to create objects especially complicated ones.

```
pip install factory-boy
```

- Create factories.py, and create test objects inside;

```
import factory
from django.contrib.auth.models import User
from django.contrib.auth.hashers import make_password

class UserFactory(factory.django.DjangoModelFactory):
  class Meta:
    model = User

    username = factory.Sequence(lambda n: f'user_{n:04}')  # user_0000, user_0001
...
    email = factory.LazyAttribute(lambda user: f'{user.username}@example.com')
    password = factory.LazyFunction(lambda: make_password('password'))
```

- Then use them in your tests;

```
from .factories import UserFactory

user = UserFactory()
```

- (Optional) For texts use FuzzyText to generate texts.

# Seperate testing database

- Even in development, we need to isolate testing db, for not to mess up with the development db.
- Go to the settings.py add test db under DATABASES, copy from previous db, change it, and add:

```
'TEST_NAME': BASE_DIR / 'test_db.sqlite3',
```

- In this way, anytime we initialize a test on db, django will create a test db and destroy it after test.

# TDD vs. Testing After

- Tests can take away joy, speed, and motivation
- When a project is young, it changes too much for testing
  - You will often delete features, and tests associated to them
  - If others aren't using it, why test?
- When adding tests later:
  - Wait until you have version 1.0
  - Create tests for those things you find crucial

# Code Coverage: How do I know what to test?

Use code coverage tool to see what to test, where to test, and if your tests enough to meet requirements.

```
# Install the package in your Django project:
pip install coverage
# Run the tool inside the project folder:
coverage run --omit='*/env/*' manage.py test
# After the first pass you can get a coverage report with:
coverage report
# You can also generate an HTML report with (a new folder called htmlcov will
appear inside the project root):
coverage html
```

- Open the index.html on a browser to check coverage report.

## Next steps

- Try tests on your own projects
- Shine on your interviews with your testing knowledge!

☺ **Happy Coding!** ✍

Clarusway