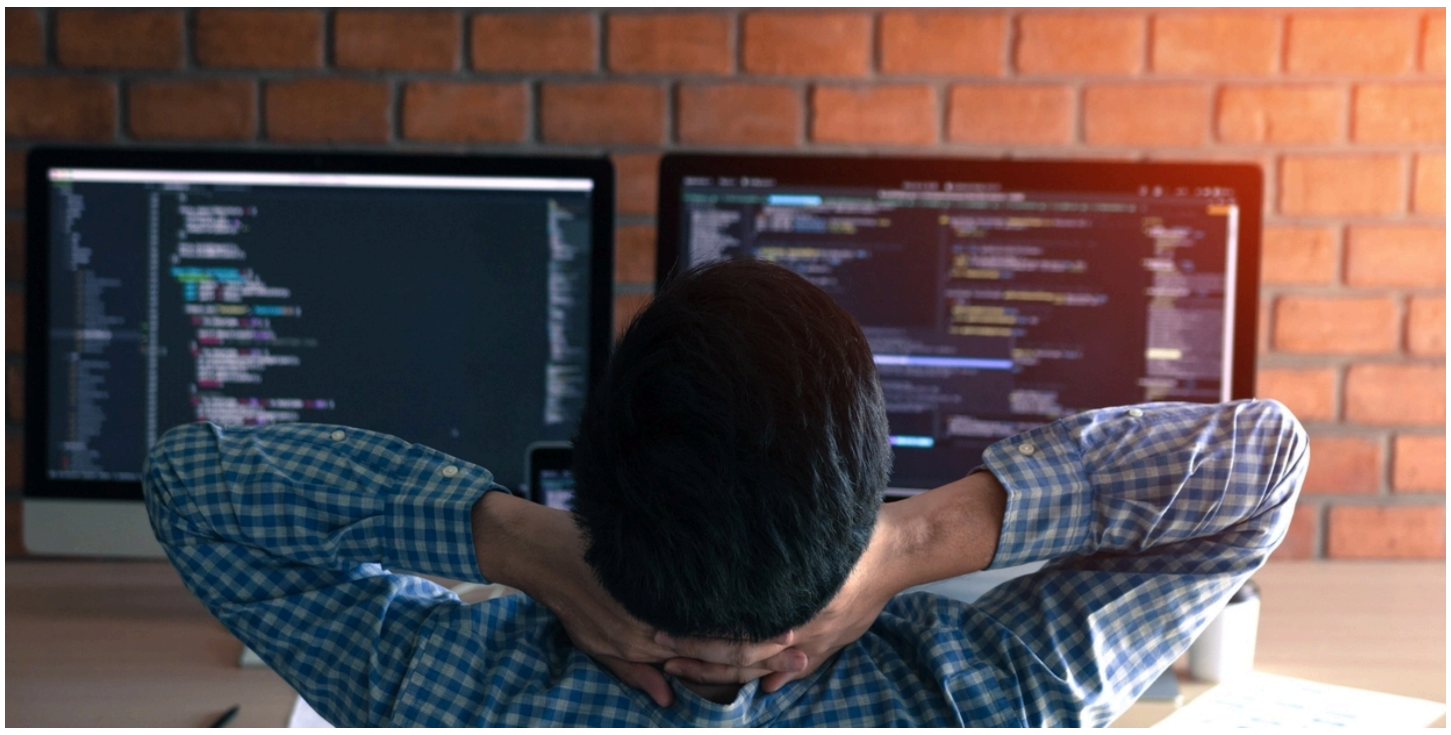
PYTHON & DJANGO

Testing Your Django App With Pytest

Mar 26, 2024 / 15 min read

#TESTING



Serhii Bolilyi
Senior Project Manager

> Helps you write better programs.
> — pytest

Many developers from Python community heard of and used unit testing to test their projects and knew about boilerplate code with Python and Django unittest module. But Pytest and Django suggest much more pythonic tests without boilerplate.

In this comprehensive tutorial, we are talking about PyTest in Python. We have sufficient experience in this topic (what is our portfolio worth), so if you need help or additional advice, please contact us.

Why You Should Use Pytest?

While you can get by without it when testing in Python and Django, Pytest provides a new approach for writing tests, namely, functional testing for applications and libraries. Below I'll list some pros and cons about this framework. We often use this in Django development.

Pros of Using Pytest:

– Assert statements (no need to remember `self.assert*` names)
– Detailed info on failures
– Fixtures (explicit, modular, scalable)
– Additional features of fixtures (auto-use, scope, request object, nested, finalizers, etc.)
– Auto-discovery of test modules and functions
– Marks
– Parametrizing
– Less boilerplate code: just create file, write function with assert and run (simple is better than complex)
– No Camel case as PyUnit
– Plugins with over 736+external plugins and thriving community
– Can run unittest and nose test suites out of the box
– Python 3.5+ and PyPy 3

Cons of Using Pytest:

– Requires a bit more advanced Python knowledge than using unittest, like decorators and simple generators
– The need for a separate installation of the module. But it can be an advantage as well, because you don't depend on Python version. If you want new features, you just need to update pytest package.

Read more: Python list generator

Short Introduction to Pytest

First of all, I would like to make a small intro to pytest philosophy and base syntax. I will do it in the format of the most frequently asked questions and answers. This is a very short intro for pytest with basic functionality, but make sure to check it as we will use it in the next parts.

Q1: What are Pytest Fixtures?

Fixtures are functions that run before and after each test, like setUp and tearDown in unitest and labelled pytest killer feature. Fixtures are used for data configuration, connection/disconnection of databases, calling extra actions, etc.

All fixtures have scope argument with available values:

- **function** run once per test
- **class** run once per class of tests
- **module** run once per module
- **session** run once per session

  Note: Default value of scope is *function*.

Example of simple fixture creation:

```
import pytest


@pytest.fixture
def function_fixture():
    print('Fixture for each test')
    return 1


@pytest.fixture(scope='module')
def module_fixture():
    print('Fixture for module')
    return 2
```

conftest.py hosted with ❤ by GitHub view raw

Another kind of fixture is yield fixture which provides access to test before and after the run, analogous to `setUp` and `tearDown`.

Example of simple `yield` fixture creation:

```
import pytest

@pytest.fixture
def simple_yield_fixture():
    print('setUp part')
    yield 3
    print('tearDown part')
```

conftest.py hosted with ❤ by GitHub view raw

> **Note:** Normal fixtures can use `yield` directly so the `yield_fixture` decorator is no longer needed and is considered deprecated.

Q2: How to use Fixtures with tests in Pytest?

To use fixture in test, you can put fixture name as function argument:

```

def test_function_fixture(function_fixture):
  assert function_fixture == 1

def test_yield_fixture(simple_yield_fixture):
  assert simple_yield_fixture == 3
```

test_own_fixture.py hosted with ❤ by GitHub view raw

> **Note: Pytest automatically register our fixtures** and can have access to fixtures without extra imports.

Q3: What is Marks in Pytest?

Marks is a helper using which you can easily set metadata on your test functions, some builtin markers, for example:

- skip – always skip a test function
- xfail – produce an "expected failure" outcome if a certain condition is met

Example of used marks:

```python
import pytest


@pytest.mark.xfail
def test_some_magic_test():
    ...


@pytest.mark.skip
def test_old_functional():
    ...
```

test_with_marks.py hosted with ❤ by GitHub view raw

Q4: How to create custom Marks for Pytest?

The one way is to register your marks in `pytest.ini` file:

```ini
[pytest]
markers =
    slow: marks tests as slow
    serial
```

pytest.ini hosted with ❤ by GitHub view raw

> **Note**: Everything after the ":" is an optional description for mark

Q5: How to run test with Marks in Pytest?

You can run all tests `xfail` without slowing down marks with the next command:

```shell
pytest -m "xfail and not slow" --strict-markers
```

shell hosted with ❤ by GitHub view raw

> **Note:** when the '–strict-markers' command-line flag is passed, any unknown marks applied with the '@pytest.mark.name_of_the_mark' decorator will trigger an error.

Q6: What is Parametrize in Pytest?

`Parametrize` is a builtin mark and one of the killer features of pytest. With this mark, you can perform multiple calls to the same test function.

Example of simple `parametrize` in test:

```python
import pytest

@pytest.mark.parametrize(
    'text_input, result', [('5+5', 10), ('1+4', 5)]
)
def test_sum(text_input, result):
    assert eval(text_input) == result
```

test_simple_parametrize.py hosted with ❤ by GitHub view raw

That's pretty much it. Further, we'll work with these basics to set up pytest for your Django project.

Setting up Pytest for Django Project

For testing our Django applications with pytest we won't reinvent the wheel and will use existing plugin pytest-django, that provides a set of useful tools for testing Django apps and projects. Let's start with configuration plugin.

1. Installation

Pytest can be installed with `pip`

```
pip install pytest-django
```

shell hosted with ❤ by GitHub view raw

Installing pytest-django will also automatically install the latest version of pytest. pytest-django uses pytest's plugin system and can be used right away after installation, there is nothing more to configure.

2. Point your Django settings to pytest

You need to tell pytest which Django settings should be used for test runs. The easiest way to achieve this is to create a pytest configuration file with this information. Create a file called `pytest.ini` in your project root directory that contains:

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

pytest.ini hosted with ❤ by GitHub view raw

When using Pytest, Django settings can also be specified by setting the `DJANGO_SETTINGS_MODULE` environment variable or specifying the `--ds=yourproject.settings` command-line flag when running the tests. See the full documentation on Configuring Django settings.

Read also: Configuring Django Settings: Best Practices

Optionally, also add the following line to the `[pytest]` section to instruct pytest to collect tests in Django's default app layouts too.

```
[pytest]
DJANGO_SETTINGS_MODULE = yourproject.settings
python_files = tests.py test_*.py *_tests.py
```

pytest.ini hosted with ❤ by GitHub view raw

3. Run your test suite

Tests are invoked directly with the pytest command, instead of manage.py test that you might be used to:

```shell
pytest
```

shell hosted with ❤ by GitHub view raw

Specific test files or directories or single test can be selected by specifying the test file names directly on the command line:

```shell
pytest a_directory                    # directory
pytest test_something.py              # tests file
pytest test_something.py::single_test  # single test function
```

shell hosted with ❤ by GitHub view raw

> **Note:** You may wonder “why would I use this instead of Django `manage.py` test command”? It’s easy. Running the test suite with `pytest` for Django offers some features that are not present in Django standard test mechanism:

- Less boilerplate: no need to import unittest, create a subclass with methods. Just write tests as regular functions.
- Manage test dependencies with fixtures.
- Run tests in multiple processes for increased speed.
- There are a lot of other nice plugins available for pytest.
- Easy switching: Existing unittest-style tests will still work without any modifications.

For now, we are configured and ready for writing first test with `pytest` and Django.

Read also: Python Rule Engine: Logic Automation & Examples

Django Testing with Pytest

1. Database Helpers

To gain access to the database pytest-django get `django_db` mark or request one of the `db`, `transactional_db` or `django_db_reset_sequences` fixtures.

> **Note:** all these database access methods automatically use `django.test.TestCase`
>
> **django_db:** to get access to the Django test database, each test will run in its own transaction that will be rolled back at the end of the test. Just like it happens in `django.test.TestCase`. We’ll use it constantly, because Django needs access to DB.

```python
import pytest

```

```python
from django.contrib.auth.models import User


@pytest.mark.django_db
def test_user_create():
  User.objects.create_user('john', 'lennon@thebeatles.com', 'johnpassword')
  assert User.objects.count() == 1
```

test_helpers.py hosted with ❤ by GitHub view raw

If you want to get access to the Django database inside a fixture this marker will not help even if the function requesting your fixture has this marker applied. To access the database in a fixture, the fixture itself will have to request the `db`, `transactional_db` or `django_db_reset_sequences` fixture. Below you may find a description of each one.

> **db:** This fixture will ensure the Django database is set up. Only required for fixtures that want to use the database themselves. A test function should

> **transactional_db:** This fixture can be used to request access to the database including transaction support. This is only required for fixtures which need database access themselves. A test function should normally use the `pytest.mark.django_db` mark with `transaction=True`.

> **django_db_reset_sequences:** This fixture provides the same transactional database access as `transactional_db`, with additional support for reset of auto increment sequences (if your database supports it). This is only required for fixtures which need database access themselves. A test function should normally use the `pytest.mark.django_db` mark with `transaction=True` and `reset_sequences=True`.

2. Client

The more frequently used thing in Django unit testing is `django.test.client`, because we use it for each request to our app, pytest-django has a build-in fixture client:

```python
import pytest

from django.urls import reverse

@pytest.mark.django_db
def test_view(client):
   url = reverse('homepage-url')
   response = client.get(url)
   assert response.status_code == 200
```

test_views.py hosted with ❤ by GitHub view raw

3. Admin Client

To get a view with superuser access, we can use `admin_client`, which gives us client with login `superuser`:

```python
import pytest

from django.urls import reverse


@pytest.mark.django_db
def test_unauthorized(client):
   url = reverse('superuser-url')
   response = client.get(url)
   assert response.status_code == 401


@pytest.mark.django_db
def test_superuser_view(admin_client):
   url = reverse('superuser-url')
   response = admin_client.get(url)
   assert response.status_code == 200
```

test_superuser_views.py hosted with ❤ by GitHub view raw

4. Create User Fixture

To create a user for our test we have two options:

> 1) Use Pytest Django Fixtures:

```python
import pytest

from django.urls import reverse


@pytest.mark.django_db
def test_user_detail(client, django_user_model):
   user = django_user_model.objects.create(
       username='someone', password='password'
   )
   url = reverse('user-detail-view', kwargs={'pk': user.pk})
   response = client.get(url)
   assert response.status_code == 200
   assert 'someone' in response.content
```

test_user_detail.py hosted with ❤ by GitHub view raw

> **django_user_model:** pytest-django helper for shortcut to the User model configured for use by the current Django project, like `settings.AUTH_USER_MODEL`

Cons of this option:

- must be copied for each test
- doesn't allow to set difference fields, because fixture creates User instance instead of us

```python
import pytest

from django.urls import reverse


@pytest.mark.django_db
def test_superuser_detail(client, admin_user):
   url = reverse(
       'superuser-detail-view', kwargs={'pk': admin_user.pk}
   )
   response = client.get(url)
   assert response.status_code == 200
   assert 'admin' in response.content
```

test_superuser_detail.py hosted with ❤ by GitHub view raw

> **admin_user:** pytest-django helper instance of a superuser, with username "admin" and password "password" (in case there is no "admin" user yet).

> 2) Create own Fixture:

To fix the disadvantages listed above we create our own custom fixture:

```python
import uuid

import pytest


@pytest.fixture
def test_password():
    return 'strong-test-pass'


@pytest.fixture
def create_user(db, django_user_model, test_password):
    def make_user(**kwargs):
        kwargs['password'] = test_password
        if 'username' not in kwargs:
            kwargs['username'] = str(uuid.uuid4())
        return django_user_model.objects.create_user(**kwargs)
    return make_user
```

conftest.py hosted with ❤ by GitHub view raw

> **Note:** Create user with call to local functions to pass extra arguments as kwargs, because pytest fixture can't accept arguments.

Re-write tests above:

```python
import pytest

from django.urls import reverse


@pytest.mark.django_db
def test_user_detail(client, create_user):
    user = create_user(username='someone')
    url = reverse('user-detail-view', kwargs={'pk': user.pk})
    response = client.get(url)
    assert response.status_code == 200
    assert 'someone' in response.content


@pytest.mark.django_db
def test_superuser_detail(client, create_user):
    admin_user = create_user(
        username='custom-admin-name',
        is_staff=True, is_superuser=True
    )
    url = reverse(
        'superuser-detail-view', kwargs={'pk': admin_user.pk}
    )
    response = client.get(url)
    assert response.status_code == 200
    assert 'custom-admin-name' in response.content
```

test_detail_views.py hosted with ❤ by GitHub view raw

> **create_user:** basic example how you can make own fixture, we can expand this fixture with two other for example, like `create_base_user` (for base user) and `create_superuser` (with fillable is_staff, is_superuser and etc.fields).

5. Auto Login Client

Let's test some authenticated endpoint:

```
import pytest

from django.urls import reverse


@pytest.mark.django_db
def test_auth_view(client, create_user, test_password):
   user = create_user()
   url = reverse('auth-url')
   client.login(
       username=user.username, password=test_password
   )
   response = client.get(url)
   assert response.status_code == 200
```

test_auth_views.py hosted with ❤ by GitHub view raw

The major disadvantage of this method is that we must copy the login block for each test.
Let's create our own fixture for auto login user:

```
import pytest


@pytest.fixture
def auto_login_user(db, client, create_user, test_password):
   def make_auto_login(user=None):
       if user is None:
           user = create_user()
       client.login(username=user.username, password=test_password)
       return client, user
   return make_auto_login
```

conftest.py hosted with ❤ by GitHub view raw

> **auto_login_user:** own fixture, that takes user as parameter or creates a new one and logins it to client fixture. And at the end it returns client and user back for the future actions.

Use our new fixture for the test above:

```
import pytest

from django.urls import reverse


@pytest.mark.django_db
def test_auth_view(auto_login_user):
   client, user = auto_login_user()
   url = reverse('auth-url')
   response = client.get(url)
```

```
    assert response.status_code == 200
```

test_auth_views.py hosted with ❤ by GitHub view raw

6. Parametrizing your tests with Pytest

Let’s say we must test very similar functionality, for example, different languages. Previously, you had to do single tests, like:

```
...
def test_de_language():
   ...
def test_gr_language():
   ...
def test_en_language():
   ...
```

test_languages.py hosted with ❤ by GitHub view raw

> It’s very funny to copy paste your test code, but not for a long time.
> — Andrew Svetlov

To fix it, pytest has parametrizing fixtures feature. After upgrade we had next tests:

```
import pytest

from django.urls import reverse


@pytest.mark.django_db
@pytest.mark.parametrize([
   ('gr', 'Yasou'),
   ('de', 'Guten tag'),
   ('fr', 'Bonjour')
])
def test_languages(language_code, text, client):
   url = reverse('say-hello-url')
   response = client.get(
       url, data={'language_code': language_code}
   )
   assert response.status_code == 200
   assert text == response.content
```

test_languages.py hosted with ❤ by GitHub view raw

You can see how much easier and less boilerplate our code has become.

7. Test Mail Outbox with Pytest

For testing your mail outbox pytest-django has a built-in fixture `mailoutbox`:

```
import pytest

from django.urls import reverse


@pytest.mark.django_db
def test_send_report(auto_login_user, mailoutbox):
   client, user = auto_login_user()
   url = reverse('send-report-url')
```

```python
    response = client.post(url)
    assert response.status_code == 201
    assert len(mailoutbox) == 1
    mail = mailoutbox[0]
    assert mail.subject == f'Report to {user.email}'
    assert list(mail.to) == [user.email]
```

test_send_report.py hosted with ❤ by GitHub view raw

For this test we use our own `auto_login_user` fixture and `mailoutbox` pytest built-in fixture.
To summarize the advantages of the approach demonstrated above: pytest teaches us how to setup our tests easily, so we could be more focused on testing main functionality.

Testing Django REST Framework with Pytest

1. API Client

The first thing to do here is to create your own fixture for API Client of PyTest-Django REST Framework:

```python
import pytest


@pytest.fixture
def api_client():
   from rest_framework.test import APIClient
   return APIClient()
```

conftest.py hosted with ❤ by GitHub view raw

Now we have `api_client` for our tests:

```python
import pytest

from django.urls import reverse


@pytest.mark.django_db
def test_unauthorized_request(api_client):
   url = reverse('need-token-url')
   response = api_client.get(url)
   assert response.status_code == 401
```

test_api.py hosted with ❤ by GitHub view raw

2. Get or Create Token

For getting authorized, your API `users` usually use Token. Let's create fixture to get or create token for a user:

```python
import pytest

from rest_framework.authtoken.models import Token


@pytest.fixture
def get_or_create_token(db, create_user):
   user = create_user()
   token, _ = Token.objects.get_or_create(user=user)
   return token
```

conftest.py hosted with ❤ by GitHub view raw

> **get_or_create_token:** inheritance `create_user`

```python
import pytest

from django.urls import reverse


@pytest.mark.django_db
def test_unauthorized_request(api_client, get_or_create_token):
   url = reverse('need-token-url')
   token = get_or_create_token()
   api_client.credentials(HTTP_AUTHORIZATION='Token ' + token.key)
   response = api_client.get(url)
   assert response.status_code == 200
```

test_token_create.py hosted with ❤ by GitHub view raw

3. Auto Credentials

The test demonstrated above is a good example, but setting credentials for each test will end up in a boilerplate code. And we can use other APIClient method to bypass pytest authentication entirely.
We can use `yield` feature to extend new fixture:

```python
import pytest


@pytest.fixture
def api_client_with_credentials(
   db, create_user, api_client
):
   user = create_user()
   api_client.force_authenticate(user=user)
   yield api_client
   api_client.force_authenticate(user=None)
```

conftest.py hosted with ❤ by GitHub view raw

> **api_client_with_credentials:** inheritance `create_user` and `api_client` fixtures and also clear our credential after every test.

```python
import pytest

from django.urls import reverse


@pytest.mark.django_db
def test_authorized_request(api_client_with_credentials):
   url = reverse('need-auth-url')
   response = api_client_with_credentials.get(url)
   assert response.status_code == 200
```

test_authorized_request.py hosted with ❤ by GitHub view raw

4. Data Validation with Pytest Parametrizing

Most tests for your API endpoint constitute and focus on data validation. You have to create the same tests without counting the difference in several values. We can use pytest `parametrizing fixture` for such solution:

```python
import pytest


@pytest.mark.django_db
@pytest.mark.parametrize(
   'email, password, status_code', [
       (None, None, 400),
       (None, 'strong_pass', 400),
       ('user@example.com', None, 400),
       ('user@example.com', 'invalid_pass', 400),
       ('user@example.com, 'strong_pass', 201),
   ]
)
def test_login_data_validation(
   email, password, status_code, api_client
):
   url = reverse('login-url')
   data = {
       'email': email,
       'password': password
   }
   response = api_client.post(url, data=data)
   assert response.status_code == status_code
```

test_login.py hosted with ❤ by GitHub view raw

By that mean, we test many cases with one test function thanks to this outstanding pytest feature.

5. Mock Extra Action in your Views

Let's demonstrate how `unittest.mock` can be used with our test use-case. I'd rather use 'unittest.mock' than 'monkeypatch' fixture. Alternatively, you can use pytest-mock package as it has some useful built-in methods like: `assert_called_once()` , `assert_called_with(*args,**kwargs)` , `assert_called()` and `assert_not_called()` .
If you want to take a closer look at `monkeypatch` fixture you may check official documentation page.
For example, we have a third-party service call after we saved our data:

```python
from rest_framework import generics

from .services import ThirdPartyService


class CreateEventView(generics.CreateAPIView):
   ...
def perform_create(self, serializer):
       event= serializer.save()
       ThirdPartyService.send_new_event(event_id=event.id)
```

views.py hosted with ❤ by GitHub view raw

We want to test our endpoint without extra request to service and we can use mock.patch as decorator with Pytest test:

```python
import pytest

from unittest import mock

```

```python

@pytest.fixture
def default_event_data():
    return {'name': 'Wizz Marathon 2019', 'event-type': 'sport'}


@pytest.mark.django_db
@mock.patch('service.ThirdPartyService.send_new_event')
def test_send_new_event_service_called(
    mock_send_new_event, default_event_data, api_client
):
    response = api_client.post(
        'create-service', data=default_event_data
    )
    assert response.status_code == 201
    assert response.data['id']
    mock_send_new_event.assert_called_with(
        event_id=response.data['id']
    )
```

test_creat_event.py hosted with ❤ by GitHub   view raw

Useful Tips for Pytest

1. Using Factory Boy as fixtures for testing your Django model

There are several ways to create Django Model instance for test and example with fixture:

- **Create object manually** — traditional variant: *"create test data by hand and support it by hand"*

```python
import pytest


from django.contrib.auth.models import User
@pytest.fixture
def user_fixture(db):
    return User.objects.create_user(
        'john', 'lennon@thebeatles.com', 'johnpassword'
    )
```

conftest.py hosted with ❤ by GitHub   view raw

If you want to add other fields like relation with Group, your fixture will get more complex and every new required field will change your fixture:

```python
import pytest

from django.contrib.auth.models import User, Group


@pytest.fixture
def default_group_fixture(db):
    default_group, _ = Group.objects.get_or_create(name='default')
    return default_group

@pytest.fixture
def user_with_default_group_fixture(db, default_group_fixture):
    user = User.objects.create_user(
        'john', 'lennon@thebeatles.com', 'johnpassword',
        groups=[default_group_fixture]
    )
    return user
```

conftest.py hosted with ❤ by GitHub view raw

- **Django fixtures** — *slow and hard to maintain… avoid them!*

Below I provide an example for comparison:

```
[
 {
 "model": "auth.group",
 "fields": {
   "name": "default",
   "permissions": [
     29,45,46,47,48
   ]
 }
},
{
 "model": "auth.user",
 "pk": 1,
 "fields": {
   "username": "simple_user",
   "first_name": "John",
   "last_name": "Lennon",
   "groups": [1],
 }
},
// create permissions here
]
```

fixture.json hosted with ❤ by GitHub view raw

Create fixture that loads fixture data to your session:

```
import pytest

from django.core.management import call_command


@pytest.fixture(scope='session')
def django_db_setup(django_db_setup, django_db_blocker):
   with django_db_blocker.unblock():
       call_command('loaddata', ‘fixture.json')
```

conftest.py hosted with ❤ by GitHub view raw

- **Factories** — *a solution for creation of your test data in a simple way.*
  I'd prefer to use pytest-factoryboy plugin and factoryboy alongside with Pytest. Alternatively, you may use model mommy.

  1) Install the plugin:

```
pip install pytest-factoryboy
```

shell hosted with ❤ by GitHub view raw

  2) Create User Factory:

```
import factory

from django.contrib.auth.models import User, Group

```

```python

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

    username = factory.Sequence(lambda n: f'john{n}')
    email = factory.Sequence(lambda n: f'lennon{n}@thebeatles.com')
    password = factory.PostGenerationMethodCall(
        'set_password', 'johnpassword'
    )

    @factory.post_generation
    def has_default_group(self, create, extracted, **kwargs):
        if not create:
            return
        if extracted:
            default_group, _ = Group.objects.get_or_create(
                name='group'
            )
            self.groups.add(default_group)
```

factories.py hosted with ❤ by GitHub view raw

3) Register Factory:

```python
from pytest_factoryboy import register

from factories import UserFactory


register(UserFactory)  # name of fixture is user_factory
```

conftest.py hosted with ❤ by GitHub view raw

Note: Name **convention** is a lowercase-underscore class name

4) Test your Factory:

```python
import pytest


@pytest.mark.django_db
def test_user_user_factory(user_factory):
    user = user_factory(has_default_group=True)
    assert user.username == 'john0'
    assert user.email == 'lennon0@thebeatles.com'
    assert user.check_password('johnpassword')
    assert user.groups.count() == 1
```

test_user_factory.py hosted with ❤ by GitHub view raw

You may read more about pytest-factoryboy and factoryboy.

2. Improve your Parametrizing tests

Let’s improve parametrizing test above with some features:

```python
import pytest


@pytest.mark.django_db
@pytest.mark.parametrize(
    'email, password, status_code', [
        ('user@example.com', 'invalid_pass', 400),
        pytest.param(
            None, None, 400,
            marks=pytest.mark.bad_request
        ),
        pytest.param(
            None, 'strong_pass', 400,
            marks=pytest.mark.bad_request,
            id='bad_request_with_pass'
        ),
        pytest.param(
            'some@magic.email', None, 400,
            marks=[
                pytest.mark.bad_request,
                pytest.mark.xfail
            ],
            id='incomprehensible_behavior'
        ),
        pytest.param(
            'user@example.com', 'strong_pass', 201,
            marks=pytest.mark.success_request
        ),
    ]
)
def test_login_data_validation(
    email, password, status_code, api_client
):
    url = reverse('login-url')
    data = {
        'email': email,
        'password': password
    }
    response = api_client.post(url, data=data)
    assert response.status_code == status_code
```

test_login.py hosted with ❤ by GitHub   view raw

pytest.param: pytest object for setting extra arguments like marks and ids

marks: argument for setting pytest mark

id: argument for setting unique indicator for test

success_request and **bad_request:** custom pytest marks

Let's run our test with some condition:

```shell
pytest -m bad_request

============== test session starts =================
collecting ... collected 5 items / 2 deselected / 3 selected
test_login.py::test_login_data_validation[None-None-400] PASSED              [ 33%]
test_login.py::test_login_data_validation[bad_request_with_pass] PASSED      [ 66%]
test_login.py::test_login_data_validation[incomprehensible_behavior] XFAIL   [100%]
```

shell hosted with ❤ by GitHub   view raw

As a result we have:
– Collected test with one of bad_request marks
– Ignore test without pytest.param object, because that don't have marks parameters
– Show test with custom ID in console

3. Mocking your Pytest test with fixture

Using pytest-mock plugin is another way to mock your code with pytest approach of naming fixtures as parameters.

1) Install the plugin:

```shell
pip install pytest-mock
```

shell hosted with ❤ by GitHub view raw

2) Re-write example above:

```python
import pytest


@pytest.mark.django_db
def test_send_new_event_service_called(
   mocker, default_event_data, api_client
):
   mock_send_new_event = mocker.patch(
       'service.ThirdPartyService.send_new_event'
   )
   response = api_client.post(
       'create-service', data=default_event_data
   )

   assert response.status_code == 201
   assert response.data['id']
   mock_send_new_event.assert_called_with(
       event_id=response.data['id']
   )
```

test_creat_event.py hosted with ❤ by GitHub view raw

The mocker is a fixture that has the same API as `mock.patch` and supports the same methods as:

- mocker.patch
- mocker.patch.object
- mocker.patch.multiple
- mocker.patch.dict
- mocker.stopall

4. Running tests simultaneously

To speed up your tests, you can run them simultaneously. This can result in significant speed improvements on multi core/multi CPU machines. It's possible to realize with pytest-xdist plugin which expands pytest functionality

1) Install the plugin:

```shell
pip install pytest-xdist
```

shell hosted with ❤ by GitHub view raw

2) Running test with multiprocessing:

```shell
pytest -n <number_of_processes>
```

shell hosted with ❤ by GitHub view raw

Notes:

– Avoid output and stdout executions in your tests, this will result in considerable speed-ups

– When tests are invoked with xdist, pytest-django will create a separate test database for each process. Each test database will be given a suffix (something like **gw0**, **gw1**) to map to a xdist process. If your database name is set to **foo**, the test database with xdist will be **test_foo_gw0**, **test_foo_gw1**, etc.

5. Config pytest.ini file

Example of `pytest.ini` file for your Django project:

```ini
[pytest]
DJANGO_SETTINGS_MODULE = yourproject.settings
python_files = tests.py test_*.py *_tests.py
addopts = -p no:warnings --strict-markers --no-migrations --reuse-db
norecursedirs = venv old_tests
markers =
    custom_mark: some information of your mark
    slow: another one slow tes
```

pytest.ini hosted with ❤ by GitHub view raw

`DJANGO_SETTINGS_MODULE` and **`python_files`** we discussed at the beginning of the article, let's discover other useful options:

- **addopts**
  Add the specified OPTS to the set of command-line arguments as if they had been specified by the user. We've specified next options:

  `--p no:warnings` — disables warning capture entirely (this might be useful if your test suites handle warnings using an external system)

  `--strict-markers` — typos and duplication in function markers are treated as an error

  `--no-migrations` will disable Django migrations and create the database by inspecting all Django models. It may be faster when there are several migrations to run in the database setup.

  `--reuse-db` reuses the testing database between test runs. It provides much faster startup time for tests.

Exemplary workflow with `--reuse-db` and `--create-db`:

– run tests with `pytest`; on the first run the test database will be created. On the next test run it will be reused.

– when you alter your database schema, run `pytest --create-db` to force re-creation of the test database.

- **norecursedirs**
  Set the exclusion of directory basename patterns when recursing for test discovery. This will tell pytest not to look into `venv` and `old_testsdirectory`

  Note: Default patterns are `'.*', 'build', 'dist', 'CVS', '_darcs', '{arch}', '*.egg', 'venv'`

- **markers**
  You can list additional markers in this setting to add them to the whitelist and use them in your tests.

Run all tests with mark **slow**:

```
pytest -m slow
```

sh hosted with ❤ by GitHub view raw

6. Show your coverage of the test

To check the Django pytest coverage of your Python app you can use pytest-cov plugin

1) Install plugin:

```
pip install pytest-cov
```

shell hosted with ❤ by GitHub view raw

2) Coverage of your project and example of report:

```
pytest --cov

-------------------- coverage: ... ---------------------
Name                 Stmts   Miss  Cover
----------------------------------------
proj/__init__          2      0    100%
proj/apps              257    13   94%
proj/proj              94     7    92%
----------------------------------------
TOTAL                  353    20   94%
```

shell hosted with ❤ by GitHub view raw

To wrap up, using this guide now you can avoid boilerplate code in your tests and make them smoother with Pytest. I hope that this post helped you to explore possibilities of Pytest deeper and bring your coding skills to the next level. **And if you need consulting or a dedicated team extension, contact Django Stars.**

FREQUENTLY ASKED QUESTIONS

Can I use PyTest with Django?

What is the advantage of Pytest?

What are the features of Pytest?

How do I run test py in Django?

How to test Django REST Framework with Pytest?

Have an idea? Let's discuss!

CONTACT US

Rate this article!

17 ratings, average: 4.53 out of 5

1	2	3	4	5
Very bad				Very good

TAG #TESTING

SHARE

RELATED ARTICLE

PYTHON & DJANGO

Unit Testing in React: Full Guide on Jest and Enzyme Testing

Alyona Pysarenko
Frontend Engineer

TECH CONSULTATION

Why You Need QA and How It Can Save Your Money

Iryna Meshchankina
Operations Manager

PYTHON & DJAN

Python De Pdb Modu

Vady
Softv

← →

COMMENTS 12

SUBSCRIBE US

LATEST ARTICLES
RIGHT IN
YOUR INBOX

Your email address *

Your name *

By clicking "SUBSCRIBE" you consent to the processing of your data by Django Stars company for marketing purposes, including sending emails. For details, check our Privacy Policy .

Subscribe

Find

Here at Django Stars we apply our extensive knowledge in both tech and business domains to help our partners build products from scratch, go through digital transformation, and scale.

REVIEWED ON

57 REVIEWS

2810 N Church St, Ste 27948
Wilmington, DE 19802-444

Mail
info@djangostars.com

Phone
+1 276 873 0823

Book a Meeting
Get free consultation

SERVICES

Software Product Development

Product Design
Software Architecture
Web Development
Mobile Development
Quality Assurance
Support & Maintenance
Cloud & DevOps

Team Extension

Python Development
Django Development
Node.js Development
JavaScript Development
React Native Development
Vue.js Development

IT Consulting & Advisory

Technology Consulting
Digital Transformation
CTO As a Service
For Startups
Product Discovery Phase
Software Reengineering

ML And AI Services

Machine Learning
Computer Vision
Data Science
Data Engineering
Virtual assistants

EXPERTISE

Industries

Fintech
Logistics
Travel
E-commerce
EdTech
PropTech
HealthCare

Technologies

Python
Django
Node.js
React.js
Vue.js
React Native
Flutter
iOS
Android

Case Studies

MoneyPark
PADI Travel
Molo
SAIB
BillionToOne
Boa Lingua
Illumidesk

Company

About Us
Reviews
Clients
Achievements
How We Work
Careers
Blog

Privacy Policy

Our statement on war 🇺🇦