

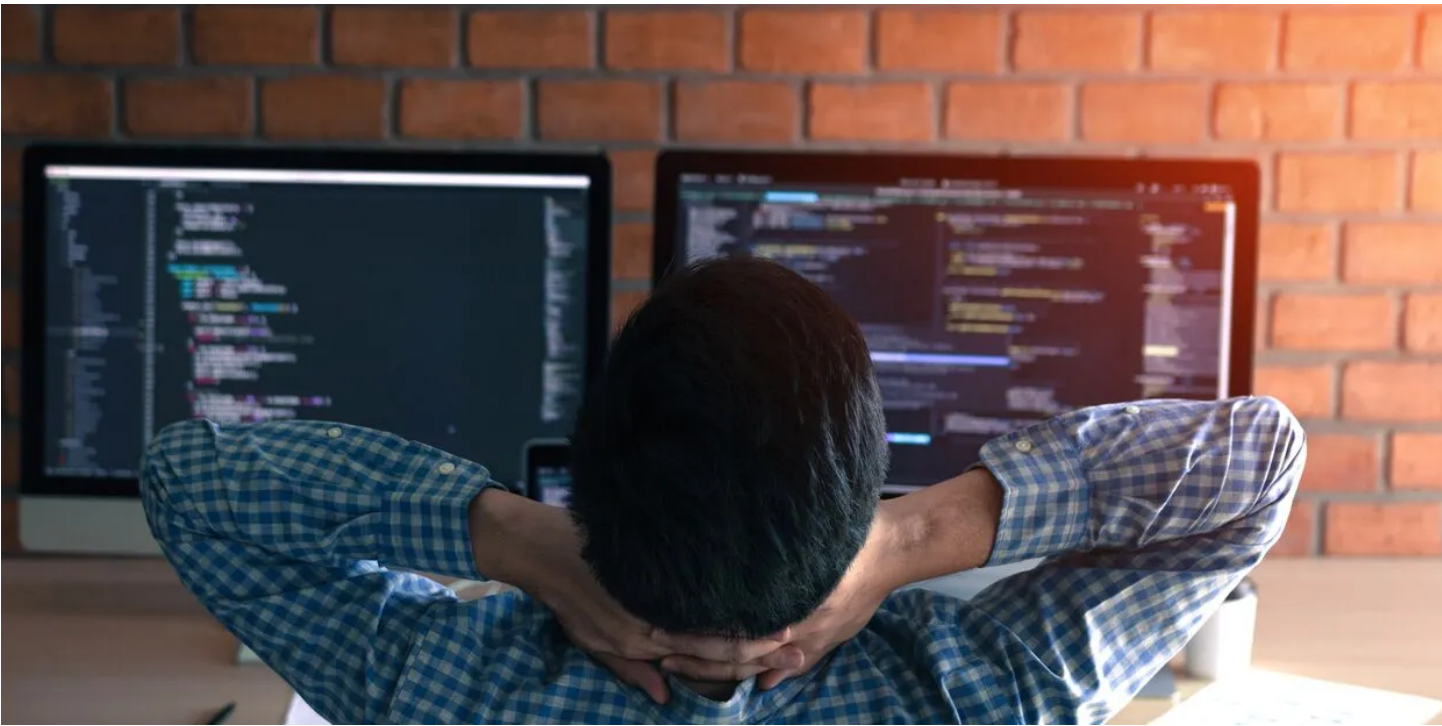


[PYTHON/DJANGO](#)

Testing Your Django App With Pytest

Aug 11, 2023 / 15 min read

#QA



 **Dmytro Chuvardynskyi**
Software Engineer

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](#).

We store cookies on your computer. We use these cookies to collect information about how you interact with our website and remember you. We use this information to improve and customize your browsing experience and for analytics and metrics about our visitors both on this website and other media. To find out more about the cookies we use, check our [Privacy Policy](#).

I agree

I do not agree



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 `unittest` 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:

ALL CATEGORIES ▾

🔍

- **session** run once per session

Note: Default value of scope is *function*.

Example of simple fixture creation:

```
1  import pytest
2
3
4  @pytest.fixture
5  def function_fixture():
6      print('Fixture for each test')
7      return 1
8
9
10 @pytest.fixture(scope='module')
11 def module_fixture():
12     print('Fixture for module')
13     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:

```
1  import pytest
2
3  @pytest.fixture
4  def simple_yield_fixture():
5      print('setUp part')
6      yield 3
7      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:

```
1
2  def test_function_fixture(function_fixture):
3      assert function_fixture == 1
4
5  def test_yield_fixture(simple_yield_fixture):
6      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.

Example of used marks:


```
1  import pytest
2
3
4  @pytest.mark.xfail
5  def test_some_magic_test():
6      ...
7
8
9  @pytest.mark.skip
10 def test_old_functional():
11     ...
```

test_with_marks.py hosted with  by [GitHub](#)[view raw](#)

Q4: How to create custom Marks for Pytest?

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

```
3     slow: marks tests as slow
4     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:

```
1  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:

```
1  import pytest
2
3  @pytest.mark.parametrize(
4      'text_input, result', [('5+5', 10), ('1+4', 5)]
5  )
6  def test_sum(text_input, result):
7      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

ALL CATEGORIES ▾

🔍

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`

```
1 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:

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

```
1 [pytest]
2 DJANGO_SETTINGS_MODULE = yourproject.settings
3 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:

```
1 pytest
```

`shell` hosted with ❤️ by [GitHub](#)[view raw](#)

— — — — —

We store cookies on your computer. We use these cookies to collect information about how you interact with our website and remember you. We use this information to improve and customize your browsing experience and for analytics and metrics about our visitors both on this website and other media. To find out more about the cookies we use, check our [Privacy Policy](#).

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.

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.

```
1 import pytest
2
3 from django.contrib.auth.models import User
4
5
6 @pytest.mark.django_db
7 def test_user_create():
8     User.objects.create_user('john', 'lennon@thebeatles.com', 'johnpassword')
9     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



normally use the `pytest.mark.django_db` mark to signal it needs the database.

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`:

```
1 import pytest
2
3 from django.urls import reverse
4
5 @pytest.mark.django_db
6 def test_view(client):
7     url = reverse('homepage-url')
8     response = client.get(url)
9     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`:

ALL CATEGORIES 

```
3 from django.urls import reverse
4
5
6 @pytest.mark.django_db
7 def test_unauthorized(client):
8     url = reverse('superuser-url')
9     response = client.get(url)
10    assert response.status_code == 401
11
12
13 @pytest.mark.django_db
14 def test_superuser_view(admin_client):
15     url = reverse('superuser-url')
16     response = admin_client.get(url)
17     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:

```
1 import pytest
2
3 from django.urls import reverse
4
5
6 @pytest.mark.django_db
7 def test_user_detail(client, django_user_model):
8     user = django_user_model.objects.create(
9         username='someone', password='password'
10    )
11    url = reverse('user-detail-view', kwargs={'pk': user.pk})
12    response = client.get(url)
13    assert response.status_code == 200
14    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

```
1 import pytest
2
3 from django.urls import reverse
4
```


ALL CATEGORIES 

12 **assert** response.status_code == 200

13 **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:

```
1  import uuid
2
3  import pytest
4
5
6  @pytest.fixture
7  def test_password():
8      return 'strong-test-pass'
9
10
11 @pytest.fixture
12 def create_user(db, django_user_model, test_password):
13     def make_user(**kwargs):
14         kwargs['password'] = test_password
15         if 'username' not in kwargs:
16             kwargs['username'] = str(uuid.uuid4())
17         return django_user_model.objects.create_user(**kwargs)
18     return make_user
```

[confest.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:

```
1  import pytest
2
3  from django.urls import reverse
4
5
6  @pytest.mark.django_db
7  def test_user_detail(client, create_user):
8      user = create_user(username='someone')
9      url = reverse('user-detail-view', kwargs={'pk': user.pk})
10     response = client.get(url)
11     assert response.status_code == 200
12     assert 'someone' in response.content
13
14
15 @pytest.mark.django_db
16 def test_superuser_detail(client, create_user):
17     admin_user = create_user(
18         username='custom-admin-name',
19         is_staff=True, is_superuser=True
20     )
21     url = reverse(
```

We store cookies on your computer. We use these cookies to collect information about how you interact with our website and remember you. We use this information to improve and customize your browsing experience and for analytics and metrics about our visitors both on this website and other media. To find out more about the cookies we use, check our [Privacy Policy](#).

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:


```
1  import pytest
2
3  from django.urls import reverse
4
5
6  @pytest.mark.django_db
7  def test_auth_view(client, create_user, test_password):
8      user = create_user()
9      url = reverse('auth-url')
10     client.login(
11         username=user.username, password=test_password
12     )
13     response = client.get(url)
14     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:

```
1  import pytest
2
3
4  @pytest.fixture
5  def auto_login_user(db, client, create_user, test_password):
6      def make_auto_login(user=None):
7          if user is None:
8              user = create_user()
9              client.login(username=user.username, password=test_password)
10             return client, user
11     return make_auto_login
```

confest.py hosted with  by [GitHub](#)[view raw](#)

auto_login_user: own fixture, that takes user as parameter or creates a new one and logs 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:

```
1  import pytest
2
3  from django.urls import reverse
4
5
6  @pytest.mark.django_db
```

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:

```
1  ...
2  def test_de_language():
3      ...
4  def test_gr_language():
5      ...
6  def test_en_language():
7      ...
```

[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:

```
1  import pytest
2
3  from django.urls import reverse
4
5
6  @pytest.mark.django_db
7  @pytest.mark.parametrize([
8      ('gr', 'Yasou'),
9      ('de', 'Guten tag'),
10     ('fr', 'Bonjour')
11 ])
12 def test_languages(language_code, text, client):
13     url = reverse('say-hello-url')
14     response = client.get(
15         url, data={'language_code': language_code}
16     )
17     assert response.status_code == 200
18     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**:

```
1  import pytest
2
3  from django.urls import reverse
4
5
```

ALL CATEGORIES 

```
13     mail = mailoutbox[0]
14     assert mail.subject == f'Report to {user.email}'
15     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](#):

```
1  import pytest
2
3
4  @pytest.fixture
5  def api_client():
6      from rest_framework.test import APIClient
7      return APIClient()
```

conftest.py hosted with  by [GitHub](#)

view raw

Now we have `api_client` for our tests:

```
1  import pytest
2
3  from django.urls import reverse
4
5
6  @pytest.mark.django_db
7  def test_unauthorized_request(api_client):
8      url = reverse('need-token-url')
9      response = api_client.get(url)
10     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:

```
1  import pytest
2
3  from rest_framework.authtoken.models import Token
4
5
6  @pytest.fixture
```

ALL CATEGORIES 



get_or_create_token: inheritance **create_user**

```
1  import pytest
2
3  from django.urls import reverse
4
5
6  @pytest.mark.django_db
7  def test_unauthorized_request(api_client, get_or_create_token):
8      url = reverse('need-token-url')
9      token = get_or_create_token()
10     api_client.credentials(HTTP_AUTHORIZATION='Token ' + token.key)
11     response = api_client.get(url)
12     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:

```
1  import pytest
2
3
4  @pytest.fixture
5  def api_client_with_credentials(
6      db, create_user, api_client
7  ):
8      user = create_user()
9      api_client.force_authenticate(user=user)
10     yield api_client
11     api_client.force_authenticate(user=None)
```

confest.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.

```
1  import pytest
2
3  from django.urls import reverse
4
5
6  @pytest.mark.django_db
7  def test_authorized_request(api_client_with_credentials):
8      url = reverse('need-auth-url')
9      response = api_client_with_credentials.get(url)
10     assert response.status_code == 200
```

test_authorized_request.py hosted with  by [GitHub](#)[view raw](#)



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:

```

1  import pytest
2
3
4  @pytest.mark.django_db
5  @pytest.mark.parametrize(
6      'email, password, status_code', [
7          (None, None, 400),
8          (None, 'strong_pass', 400),
9          ('user@example.com', None, 400),
10         ('user@example.com', 'invalid_pass', 400),
11         ('user@example.com', 'strong_pass', 201),
12     ]
13 )
14 def test_login_data_validation(
15     email, password, status_code, api_client
16 ):
17     url = reverse('login-url')
18     data = {
19         'email': email,
20         'password': password
21     }
22     response = api_client.post(url, data=data)
23     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:


```

1  from rest_framework import generics
2
3  from .services import ThirdPartyService
4
5
6  class CreateEventView(generics.CreateAPIView):
7      ...
8  def perform_create(self, serializer):
9      event= serializer.save()
10     ThirdPartyService.send_new_event(event_id=event.id)

```

We store cookies on your computer. We use these cookies to collect information about how you interact with our website and remember you. We use this information to improve and customize your browsing experience and for analytics and metrics about our visitors both on this website and other media. To find out more about the cookies we use, check our [Privacy Policy](#).

```
1  import pytest
2
3  from unittest import mock
4
5
6  @pytest.fixture
7  def default_event_data():
8      return {'name': 'Wizz Marathon 2019', 'event-type': 'sport'}
9
10
11 @pytest.mark.django_db
12 @mock.patch('service.ThirdPartyService.send_new_event')
13 def test_send_new_event_service_called(
14     mock_send_new_event, default_event_data, api_client
15 ):
16     response = api_client.post(
17         'create-service', data=default_event_data
18     )
19     assert response.status_code == 201
20     assert response.data['id']
21     mock_send_new_event.assert_called_with(
22         event_id=response.data['id']
23     )
```

[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”*

```
1  import pytest
2
3
4  from django.contrib.auth.models import User
5  @pytest.fixture
6  def user_fixture(db):
7      return User.objects.create_user(
8          'john', 'lennon@thebeatles.com', 'johnpassword'
9      )
```

[confest.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:

```
1  import pytest
2
3  from django.contrib.auth.models import User, Group
```

We store cookies on your computer. We use these cookies to collect information about how you interact with our website and remember you. We use this information to improve and customize your browsing experience and for analytics and metrics about our visitors both on this website and other media. To find out more about the cookies we use, check our [Privacy Policy](#).

ALL CATEGORIES 

```
11 @pytest.fixture
12 def user_with_default_group_fixture(db, default_group_fixture):
13     user = User.objects.create_user(
14         'john', 'lennon@thebeatles.com', 'johnpassword',
15         groups=[default_group_fixture]
16     )
17     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:

```
1  [
2    {
3      "model": "auth.group",
4      "fields": {
5        "name": "default",
6        "permissions": [
7          29, 45, 46, 47, 48
8        ]
9      }
10   },
11   {
12     "model": "auth.user",
13     "pk": 1,
14     "fields": {
15       "username": "simple_user",
16       "first_name": "John",
17       "last_name": "Lennon",
18       "groups": [1],
19     }
20   },
21   // create permissions here
22 ]
```

[fixture.json](#) hosted with  by [GitHub](#)[view raw](#)

Create fixture that loads fixture data to your session:


```
1 import pytest
2
3 from django.core.management import call_command
4
5
6 @pytest.fixture(scope='session')
7 def django_db_setup(django_db_setup, django_db_blocker):
8     with django_db_blocker.unblock():
9         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:

ALL CATEGORIES 



```
1  import factory
2
3  from django.contrib.auth.models import User, Group
4
5
6  class UserFactory(factory.DjangoModelFactory):
7      class Meta:
8          model = User
9
10     username = factory.Sequence(lambda n: f'john{n}')
11     email = factory.Sequence(lambda n: f'lennon{n}@thebeatles.com')
12     password = factory.PostGenerationMethodCall(
13         'set_password', 'johnpassword'
14     )
15
16     @factory.post_generation
17     def has_default_group(self, create, extracted, **kwargs):
18         if not create:
19             return
20         if extracted:
21             default_group, _ = Group.objects.get_or_create(
22                 name='group'
23             )
24             self.groups.add(default_group)
```

factories.py hosted with  by [GitHub](#)[view raw](#)

3) Register Factory:

```
1  from pytest_factoryboy import register
2
3  from factories import UserFactory
4
5
6  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:

```
1  import pytest
2
3
4  @pytest.mark.django_db
5  def test_user_user_factory(user_factory):
6      user = user_factory(has_default_group=True)
7      assert user.username == 'john0'
8      assert user.email == 'lennon0@thebeatles.com'
9      assert user.check_password('johnpassword')
10     assert user.groups.count() == 1
```

test_user_factory.py hosted with  by [GitHub](#)[view raw](#)

You may read more about [pytest-factoryboy](#) and [factoryboy](#).


We store cookies on your computer. We use these cookies to collect information about how you interact with our website and remember you. We use this information to improve and customize your browsing experience and for analytics and metrics about our visitors both on this website and other media. To find out more about the cookies we use, check our [Privacy Policy](#).

ALL CATEGORIES 



Let’s improve parametrizing test above with some features:

```
1  import pytest
2
3
4  @pytest.mark.django_db
5  @pytest.mark.parametrize(
6      'email, password, status_code', [
7          ('user@example.com', 'invalid_pass', 400),
8          pytest.param(
9              None, None, 400,
10             marks=pytest.mark.bad_request
11         ),
12         pytest.param(
13             None, 'strong_pass', 400,
14             marks=pytest.mark.bad_request,
15             id='bad_request_with_pass'
16         ),
17         pytest.param(
18             'some@magic.email', None, 400,
19             marks=[
20                 pytest.mark.bad_request,
21                 pytest.mark.xfail
22             ],
23             id='incomprehensible_behavior'
24         ),
25         pytest.param(
26             'user@example.com', 'strong_pass', 201,
27             marks=pytest.mark.success_request
28         ),
29     ]
30 )
31 def test_login_data_validation(
32     email, password, status_code, api_client
33 ):
34     url = reverse('login-url')
35     data = {
36         'email': email,
37         'password': password
38     }
39     response = api_client.post(url, data=data)
40     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:

```
1  pytest -m bad_request
2
3  ===== test session starts =====
4  collecting ... collected 5 items / 2 deselected / 3 selected
5  .....
6  ===== 3 passed, 2 deselected in 0.01s =====
```

We store cookies on your computer. We use these cookies to collect information about how you interact with our website and remember you. We use this information to improve and customize your browsing experience and for analytics and metrics about our visitors both on this website and other media. To find out more about the cookies we use, check our [Privacy Policy](#).


- 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:

```
1 pip install pytest-mock
```

[shell](#) hosted with  by [GitHub](#)[view raw](#)


2) Re-write example above:


```
1 import pytest
2
3
4 @pytest.mark.django_db
5 def test_send_new_event_service_called(
6     mocker, default_event_data, api_client
7 ):
8     mock_send_new_event = mocker.patch(
9         'service.ThirdPartyService.send_new_event'
10    )
11     response = api_client.post(
12         'create-service', data=default_event_data
13    )
14
15     assert response.status_code == 201
16     assert response.data['id']
17     mock_send_new_event.assert_called_with(
18         event_id=response.data['id']
19    )
```

[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`

ALL CATEGORIES 



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:

```
1 pip install pytest-xdist
```

[shell](#) hosted with  by [GitHub](#)[view raw](#)

2) Running test with multiprocessing:

```
1 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:

```
1 [pytest]
2 DJANGO_SETTINGS_MODULE = yourproject.settings
3 python_files = tests.py test_*.py *_tests.py
4 addopts = -p no:warnings --strict-markers --no-migrations --reuse-db
5 norecursedirs = venv old_tests
6 markers =
7     custom_mark: some information of your mark
8     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)

We store cookies on your computer. We use these cookies to collect information about how you interact with our website and remember you. We use this information to improve and customize your browsing experience and for analytics and metrics about our visitors both on this website and other media. To find out more about the cookies we use, check our [Privacy Policy](#).

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**:

```
1  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 app you can use [pytest-cov](#) plugin

1) Install plugin:

```
1  pip install pytest-cov
```

[shell](#) hosted with  by [GitHub](#)[view raw](#)

2) Coverage of your project and example of report:

ALL CATEGORIES 

3 ----- coverage: ... -----

4 Name Stmts Miss Cover

5 -----


6 proj/__init__ 2 0 100%

7 proj/apps 257 13 94%

8 proj/proj 94 7 92%

9 -----

10 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

TAG #QA

SHARE   

We store cookies on your computer. We use these cookies to collect information about how you interact with our website and remember you. We use this information to improve and customize your browsing experience and for analytics and metrics about our visitors both on this website and other media. To find out more about the cookies we use, check our [Privacy Policy](#).

ALL CATEGORIES 



TECHCONSULT

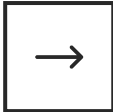
Why You Need QA and How It Can Save Your Money.

[Iryna Meshchankina](#)
[Operations Manager](#)

PYTHON/DJANGO

Debugging Python Applications with the PDB Module

[Vadym Zakovinko](#)
[Software Architect / Team Lead](#)



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



We store cookies on your computer. We use these cookies to collect information about how you interact with our website and remember you. We use this information to improve and customize your browsing experience and for analytics and metrics about our visitors both on this website and other media. To find out more about the cookies we use, check our [Privacy Policy](#).

Find
2023-11-01 15:01:01 - 07010

ALL CATEGORIES 



©2023 Django Stars, LLC. All rights reserved. 2711 Centerville Road, Suite 400, Wilmington, DE 19808, USA

We store cookies on your computer. We use these cookies to collect information about how you interact with our website and remember you. We use this information to improve and customize your browsing experience and for analytics and metrics about our visitors both on this website and other media. To find out more about the cookies we use, check our [Privacy Policy](#).