

**Fixture Replacement: Factory Boy and Faker**

In the previous tutorial we started learning about pytest fixtures and factories. In this tutorial we will look at a fixture replacement in *factory boy* and *faker* (*a package that generates fake data for us*).

***Factory Boy***:



The idea here is that *factories can be considered quite hard to manage as our application scales*, so with this package factories are defined in a clean way even for complex objects.

Since factory boy is a class based approach, it is easier to use sub factories. For example *if we had a factory that we wanted to connect with another complex factory then by using factory boy this process could be fairly simplified*.

Second if we are going to need data from our database in our test then its likely that our tables are connected to one another in some sort of relation (*ForeignKey, ManyToMany etc.*) then we can use this tool to model our data or prepare our data.

In the previous tutorial, we created this factory *using pytest fixture* to create different types of users (*admin , staff users*).

@pytest.fixture

def new\_user\_factory(db):

    def create\_app\_user(

        username: str,

        password: str = None,

        first\_name: str = "firstName",

        last\_name: str = "lastName",

        email: str = "test@test.com",

        is\_staff: str = False,

        is\_superuser: str = False,

        is\_active: str = True,

    ):

        user = User.objects.create\_user(

            username=username,

            password=password,

            first\_name=first\_name,

            last\_name=last\_name,

            email=email,

            is\_staff=is\_staff,

            is\_superuser=is\_superuser,

            is\_active=is\_active,

        )

        return user

    return create\_app\_user

Then we use this fixture in another fixtures, to create a specific user

@pytest.fixture 🡪 *for non – admin user*

def new\_user(db, new\_user\_factory):

    return new\_user\_factory("Test\_user", "password", "MyName")

@pytest.fixture 🡪 *for admin user*

def new\_user2(db, new\_user\_factory):

    return new\_user\_factory("Test\_user", "password", "MyName", is\_staff=True)

Then in the actual test,

def test\_new\_user(new\_user2):

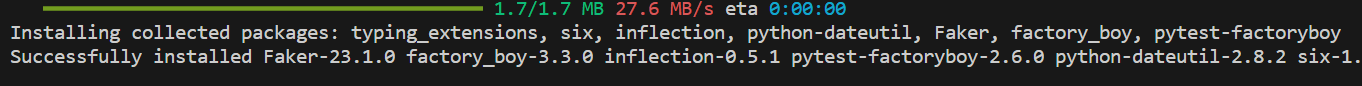
    print(new\_user2.is\_staff)

    assert new\_user2.is\_staff

We tested to see if the user is staff or not.

Now we are going to the same thing but using the *factory boy*.

pip install pytest-factoryboy



As we can see Faker is installed automatically with this as well.

Note: We can print all the project requirements in a single file with this command.

pip freeze > requirements.txt



*//requirements.txt*

asgiref==3.7.2

colorama==0.4.6

Django==5.0.1

factory-boy==3.3.0

Faker==23.1.0

inflection==0.5.1

iniconfig==2.0.0

packaging==23.2

pluggy==1.4.0

pytest==8.0.0

pytest-django==4.8.0

pytest-factoryboy==2.6.0

python-dateutil==2.8.2

six==1.16.0

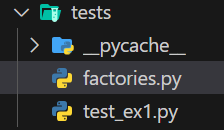
sqlparse==0.4.4

typing\_extensions==4.9.0

tzdata==2023.4

This creates a new file called requirements.txt in our project root folder.

Back to the topic, in order to migrate to factory boy, first step is to create a new file called *factories.py* in our tests directory.



In this file we import factory, in order to use factory boy.

import factory

Next we need to create classes and essentially *going to connect this class to an existing model*, so if our model changes then so does this factory class (*provides flexibility*).

class UserFactory(factory.django.DjangoModelFactory):

we create a UserFactory class which inherits from *factory.django.DjangoModelFactory*.

now we just define our model attribute in Meta class.

import factory

from django.contrib.auth.models import User

class UserFactory(factory.django.DjangoModelFactory):

    class Meta:

        model = User

We can also define what fields want to utilize from this User table.

class UserFactory(factory.django.DjangoModelFactory):

    class Meta:

        model = User

    username = "name" *🡪 some default parameters we can set in the table*

    is\_staff = True

***How to use faker here***?

*Faker* can help us here by generating fake data for our table. So let’s import Faker class from faker module.

from faker import Faker

We get the instance of this class and then can use in our UserFactory like this.

fake = Faker() 🡪 *instantiating Faker class*

class UserFactory(factory.django.DjangoModelFactory):

    class Meta:

        model = User

    username = fake.name() 🡪 *using fake object as username*

    is\_staff = True

In order *to utilize this UserFactory in our test, we need to register it first in our conftest.py file*. So let’s do all the necessary imports first.

*#conftest.py*

import pytest

from pytest\_factoryboy import register 🡪 *function for registering factories*

from tests.factories import UserFactory

Next we simply pass UserFactory to *register* function.

*#conftest.py*

import pytest

from pytest\_factoryboy import register

from tests.factories import UserFactory

register(UserFactory) 🡪 *here*

Note: Our UserFactory class will be accessed as *user\_factory* (*lowercase separated by underscores*) in our test case.

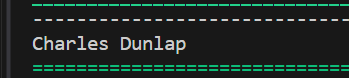
In our test file,

def test\_new\_user(user\_factory): 🡪*we used user\_factory as parameter to our test*

    print(user\_factory.username)

    assert True

O/P:



As we can see a user is generated from the factory and it will be different every time we run our test.

So factory boy can provide us two built in strategies.

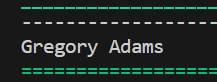
🡪 We can use factory to just create an object using *build* method.

def test\_new\_user(user\_factory):

    user = user\_factory.build()

    print(user.username)

    assert True



🡪 We can use factory to actually save data in test database using *create* method.

import pytest

from django.contrib.auth.models import User

@pytest.mark.django\_db 🡪 *use django\_db in order to access database*

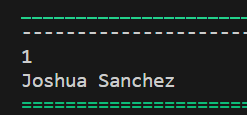
def test\_new\_user(user\_factory):

    user = user\_factory.create()

    print(User.objects.all().count())

    print(user.username)

    assert True



If we change create to build in above code,

@pytest.mark.django\_db

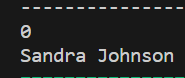
def test\_new\_user(user\_factory):

    user = user\_factory.build()

    print(User.objects.all().count())

    print(user.username)

    assert True

🡨zero objects stored in our test database.

In order to have this part of creating or building (*create() / build()* ) user separated from our test file, we can write a fixture in our conftest.py file for the same.

import pytest

from pytest\_factoryboy import register

from tests.factories import UserFactory

register(UserFactory)

@pytest.fixture

def new\_user1(db, user\_factory):

    user = user\_factory.build()

    return user

Then we can use it in our test file.

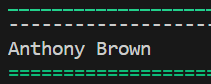
import pytest

@pytest.mark.django\_db

def test\_new\_user(new\_user1):

    print(new\_user1.username)

    assert True



Our test file is smaller now and more concise.