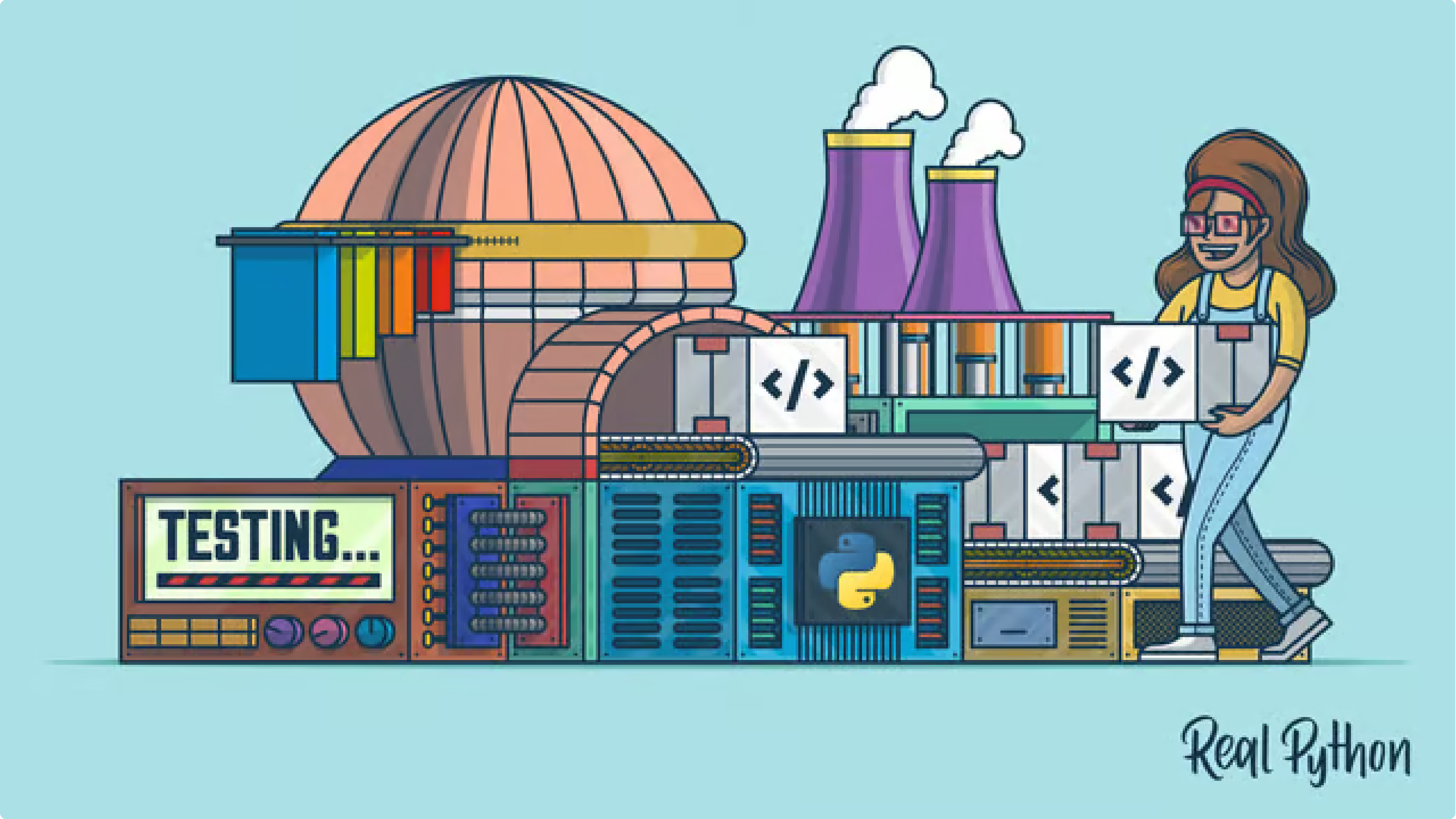


How to Provide Test Fixtures for Django Models in Pytest

by Haki Benita Apr 08, 2020 7 Comments

django intermediate testing

Mark as Completed

Tweet Share Email

Table of Contents

Help

If you're working in Django, `pytest` fixtures can help you create tests for your models that are uncomplicated to maintain. Writing good tests is a crucial step in sustaining a successful app, and **fixtures** are a key ingredient in making your test suite efficient and effective. Fixtures are little pieces of data that serve as the baseline for your tests.

As your test scenarios change, it can be a pain to add, modify, and maintain your fixtures. But don't worry. This tutorial will show you how to use the `pytest-django` plugin to make writing new test cases and fixtures a breeze.

In this tutorial, you'll learn:

- How to create and load **test fixtures** in Django
- How to create and load **`pytest` fixtures for Django models**
- How to use **factories** to create test fixtures for Django models in `pytest`
- How to create dependencies between test fixtures using the **factory as fixture** pattern

The concepts described in this tutorial are suited for any Python project using `pytest`. For convenience, the examples use the Django ORM, but the results can be reproduced in other types of ORMs and even in projects that don't use an ORM or a database.

Fixtures in Django

To get started, you're going to set up a fresh Django project. Throughout this tutorial, you'll write some tests using the built-in authentication module.

Setting Up a Python Virtual Environment

When you create a new project, it's best to also create a virtual environment for it. A virtual environment allows you to isolate the project from other projects on your computer. This way, different projects can use different versions of Python, Django, or any other package without interfering with each other.

Here's how you can create your virtual environment in a new directory:

Shell

```
$ mkdir django_fixtures
$ cd django_fixtures
django_fixtures $ python -m venv venv
```

For step-by-step instructions on how to create a virtual environment, check out Python Virtual Environments: A Primer.

Running this command will create a new directory called `venv`. This directory will store all the packages you install inside the virtual environment.

Setting Up a Django Project

Now that you have a fresh virtual environment, it's time to set up a Django project. In your terminal, activate the virtual environment and install Django:

Shell

```
$ source venv/bin/activate
$ pip install django
```

Now that you have Django installed, you can create a new Django project called `django_fixtures`:

Shell

```
$ django-admin startproject django_fixtures
```

After running this command, you'll see that Django created new files and directories. For more about how to start a new Django project, check out Starting a Django Project.

To finish setting up your Django project, apply the migrations for the built-in modules:

Shell

```
$ cd django_fixtures
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying sessions.0001_initial... OK
```

The output lists all the migrations Django applied. When starting a new project, Django applies migrations for built-in apps such as `auth`, `sessions`, and `admin`.

Now you're ready to start writing tests and fixtures!

Creating Django Fixtures

Django provides its own way of creating and loading fixtures for models from files. Django fixture files can be written in either JSON or YAML. In this tutorial, you'll work with the JSON format.

The easiest way to create a Django fixture is to use an existing object. Start a Django shell:

Shell

```
$ python manage.py shell
Python 3.8.0 (default, Oct 23 2019, 18:51:26)
[GCC 9.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
```

Inside the Django shell, create a new group called `appusers`:

Python >>>

```
>>> from django.contrib.auth.models import Group
>>> group = Group.objects.create(name="appusers")
>>> group.pk
1
```

The `Group` model is part of Django's authentication system. Groups are very useful for managing permissions in a Django project.

You created a new group called `appusers`. The **primary key** of the group you just created is `1`. To create a fixture for the group `appusers`, you are going to use [the Django management command `dumpdata`](#).

Exit the Django shell with `exit()` and execute the following command from your terminal:

Shell

```
$ python manage.py dumpdata auth.Group --pk 1 --indent 4 > group.json
```

In this example, you're using the `dumpdata` command to generate fixture files from existing model instances. Let's break it down:

- **`auth.Group`**: This describes which model to dump. The format is `<app_label>.<model_name>`.
- **`--pk 1`**: This describes which object to dump. The value is a comma-delimited list of primary keys, such as `1,2,3`.
- **`--indent 4`**: This is an optional formatting argument that tells Django how many spaces to add before each indention level in the generated file. Using indentions makes the fixture file more readable.
- **`> group.json`**: This describes where to write the output of the command. In this case, the output will be written to a file called `group.json`.

Next, inspect the contents of the fixture file `group.json`:

JSON

```
[
{
    "model": "auth.group",
    "pk": 1,
    "fields": {
        "name": "appusers",
        "permissions": []
    }
}
]
```

The fixture file contains a list of objects. In this case, you have only one object in the list. Each object includes a **header** with the name of the model and the primary key, as well as a **dictionary** with the value for each field in the model. You can see that the fixture contains the name of the group `appusers`.

You can create and edit fixture files manually, but it's usually more convenient to create the object beforehand and use Django's `dumpdata` command to create the fixture file.

Loading Django Fixtures

Now that you have a fixture file, you want to load it into the database. But before you do that, you should open a Django shell and delete the group that you already created:

Python >>>

```
>>> from django.contrib.auth.models import Group
>>> Group.objects.filter(pk=1).delete()
(1, {'auth.Group_permissions': 0, 'auth.User_groups': 0, 'auth.Group': 1})
```

Now that the group is deleted, load the fixture using the [`loaddata` command](#):

Shell

```
$ python manage.py loaddata group.json
Installed 1 object(s) from 1 fixture(s)
```

To make sure the new group was loaded, open a Django shell and fetch it:

Python >>>

```
>>> from django.contrib.auth.models import Group
>>> group = Group.objects.get(pk=1)
>>> vars(group)
{'_state': <django.db.models.base.ModelState at 0x7f3a012d08b0>,
 'id': 1,
 'name': 'appusers'}
```

Great! The group was loaded. You just created and loaded your first Django fixture.

Loading Django Fixtures in Tests

So far you've created and loaded a fixture file from the command line. Now how can you use it for testing? To see how fixtures are used in Django tests, create a new file called `test.py`, and add the following test:

Python

```
from django.test import TestCase
from django.contrib.auth.models import Group

class MyTest(TestCase):
    def test_should_create_group(self):
        group = Group.objects.get(pk=1)
        self.assertEqual(group.name, "appusers")
```

The test is fetching the group with the primary key `1` and testing that its name is `appusers`.

Run the test from your terminal:

Shell

```
$ python manage.py test test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
E
======================================================================
ERROR: test_should_create_group (test.MyTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/django_fixtures/django_fixtures/test.py", line 9, in test_should_create_group
    group = Group.objects.get(pk=1)
  File "/django_fixtures/venv/lib/python3.8/site-packages/django/db/models/manager.py", line 82, in
    return getattr(self.get_queryset(), name)(*args, **kwargs)
  File "/django_fixtures/venv/lib/python3.8/site-packages/django/db/models/query.py", line 415, in g
    raise self.model.DoesNotExist(
django.contrib.auth.models.Group.DoesNotExist: Group matching query does not exist.

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
Destroying test database for alias 'default'...
```

The test failed because a group with primary key `1` does not exist.

To load the fixture in the test, you can use a special attribute of the class `TestCase` called `fixtures`:

Python

```python
from django.test import TestCase
from django.contrib.auth.models import Group

class MyTest(TestCase):
    fixtures = ["group.json"]

    def test_should_create_group(self):
        group = Group.objects.get(pk=1)
        self.assertEqual(group.name, "appusers")
```

Adding this attribute to a `TestCase` tells Django to load the fixtures before executing each test. Notice that `fixtures` accepts an array, so you can provide multiple fixture files to load before each test.

Running the test now produces the following output:

Shell

```
$ python manage.py test test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.005s

OK
Destroying test database for alias 'default'...
```

Amazing! The group was loaded and the test passed. You can now use the group `appusers` in your tests.

Referencing Related Objects in Django Fixtures

So far you've used just one file with a single object. Most of the time, however, you will have many models in your app, and you'll need more than one model in a test.

To see how dependencies between objects look in Django fixtures, create a new user instance and then add it to the `appusers` group you created before:

Python

```python
>>> from django.contrib.auth.models import User, Group
>>> appusers = Group.objects.get(name="appusers")
>>> haki = User.objects.create_user("haki")
>>> haki.pk
1
>>> haki.groups.add(appusers)
```

The user `haki` is now a member of the `appusers` group. To see what a fixture with a foreign key looks like, generate a fixture for user `1`:

Shell

```
$ python manage.py dumpdata auth.User --pk 1 --indent 4
[
{
    "model": "auth.user",
    "pk": 1,
    "fields": {
        "password": "!M4dygH3ZWfd0214U59OR9nlwsRJ94HUZtvQciG8y",
        "last_login": null,
        "is_superuser": false,
        "username": "haki",
        "first_name": "",
        "last_name": "",
        "email": "",
        "is_staff": false,
        "is_active": true,
        "date_joined": "2019-12-07T09:32:50.998Z",
        "groups": [
            1
        ],
        "user_permissions": []
    }
}
]
```

The structure of the fixture is similar to the one you saw earlier.

A user can be associated with multiple groups, so the field `group` contains the IDs for all the groups to which the user belongs. In this case, the user belongs the group with primary key `1`, which is your `appusers` group.

Using primary keys to reference objects in fixtures is not always a good idea. The primary key of a group is an arbitrary identifier that the database assigns to the group when it is created. In another environment, or on another computer, the `appusers` group can have a different ID and it wouldn't make any difference on the object.

To avoid using arbitrary identifiers, Django defines the concept of natural keys. A natural key is a unique identifier of an object that is not necessarily the primary key. In the case of groups, two groups can't have the same name, so a natural key for the group can be its name.

To use natural keys instead of primary keys to reference related objects in a Django fixture, add the `--natural-foreign` flag to the `dumpdata` command:

Shell

```
$ python manage.py dumpdata auth.User --pk 1 --indent 4 --natural-foreign
[
{
    "model": "auth.user",
    "pk": 1,
    "fields": {
        "password": "!f4dygH3ZWfd0214X59OR9ndwsRJ94HUZ6vQciG8y",
        "last_login": null,
        "is_superuser": false,
        "username": "haki",
        "first_name": "",
        "last_name": "",
        "email": "benita",
        "is_staff": false,
        "is_active": true,
        "date_joined": "2019-12-07T09:32:50.998Z",
        "groups": [
            [
                `appusers`
            ]
        ],
        "user_permissions": []
    }
}
]
```

Django generated the fixture for the user, but instead of using the primary key of the `appusers` group, it used the group's name.

You can also add the `--natural-primary` flag to exclude an object's primary key from the fixture. When `pk` is null, the primary key will be set at runtime, usually by the database.

Maintaining Django Fixtures

Django fixtures are great, but they also pose some challenges:

- **Keeping fixtures updated**: Django fixtures must contain all the required fields of the model. If you add a new field that is not nullable, you must update the fixtures. Otherwise, they will fail to load. Keeping Django fixtures updated can become a burden when you have lots of them.

- **Maintaining dependencies between fixtures**: Django fixtures that depend on other fixtures must be loaded together and in a particular order. Keeping up with fixtures as new test cases are added and old test cases are modified can be challenging.

For these reasons, Django fixtures are not an ideal choice for models that change often. For example, it would be very difficult to maintain Django fixtures for models that are used to represent core objects in the app such as sales, orders, transactions, or reservations.

On the other hand, Django fixtures are a great option for the following use cases:

- **Constant data**: This applies to models that rarely change, such as country codes and zip codes.

- **Initial data**: This applies to models that store your app's lookup data, such as product categories, user groups, and user types.

`pytest` Fixtures in Django

In the previous section, you used the built-in tools provided by Django to create and load fixtures. The fixtures provided by Django are great for some use cases, but not ideal for others.

In this section, you're going to experiment with a very different type of fixture: the `pytest` fixture. `pytest` provides a very extensive fixture system that you can use to create a reliable and maintainable test suite.

Setting Up `pytest` for a Django Project

To get started with `pytest`, you first need to install `pytest` and the [Django plugin for `pytest`]. Execute the following commands in your terminal while the virtual environment is activated:

Shell

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

The `pytest-django` plugin is maintained by the `pytest` development team. It provides useful tools for writing tests for Django projects using `pytest`.

Next, you need to let `pytest` know where it can locate your Django project settings. Create a new file in the project's root directory called `pytest.ini`, and add the following lines to it:

Config File

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

This is the minimum amount of configuration needed to make `pytest` work with your Django project. There are many more configuration options, but this is enough to get started.

Finally, to test your setup, replace the contents of `test.py` with this dummy test:

Python

```python
def test_foo():
    assert True
```

To run the dummy test, use the `pytest` command from your terminal:

Shell

```shell
$ pytest test.py
============================== test session starts ======================
platform linux -- Python 3.7.4, pytest-5.2.0, py-1.8.0, pluggy-0.13.0
Django settings: django_fixtures.settings (from ini file)
rootdir: /django_fixtures, inifile: pytest.ini
plugins: django-3.5.1

test.py .
                                [100%]
============================= 1 passed in 0.05s =========================
```

You just completed setting up a new Django project with `pytest`! Now you're ready to dig deeper.

For more about how to set up `pytest` and write tests, check out Test-Driven Development With `pytest`.

Accessing the Database From Tests

In this section, you're going to write tests using the built-in authentication module `django.contrib.auth`. The most familiar models in this module are `User` and `Group`.

To get started with both Django and `pytest`, write a test to check if the function `create_user()` provided by Django is setting the username correctly:

Python

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

def test_should_create_user_with_username() -> None:
    user = User.objects.create_user("Haki")
    assert user.username == "Haki"
```

Now, try to execute the test from your command like:

Shell

```
$ pytest test.py
================================== test session starts ===============
platform linux -- Python 3.7.4, pytest-5.2.0, py-1.8.0, pluggy-0.13.0
Django settings: django_fixtures.settings (from ini file)
rootdir: /django-django_fixtures/django_fixtures, inifile: pytest.ini
plugins: django-3.5.1
collected 1 item

test.py F

=============================== FAILURES =============================
____________________test_should_create_user_with_username ____________

    def test_should_create_user_with_username() -> None:
>       user = User.objects.create_user("Haki")

self = <mydbengine.base.DatabaseWrapper object at 0x7fef66ed57d0>, name = None

    def _cursor(self, name=None):
>       self.ensure_connection()

E   Failed: Database access not allowed, use the "django_db" mark, or the "db"
        or "transactional_db" fixtures to enable it.
```

The command failed, and the test did not execute. The error message gives you some useful information: To access the database in a test you need to inject a special fixture called `db`. The `db` fixture is part of the `django-pytest` plugin you installed earlier, and it's required to access the database in tests.

Inject the `db` fixture into the test:

Python

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

def test_should_create_user_with_username(db) -> None:
    user = User.objects.create_user("Haki")
    assert user.username == "Haki"
```

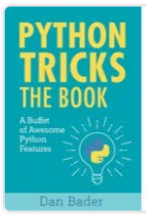
Run the test again:

Shell

```
$ pytest test.py
================================== test session starts ===============
platform linux -- Python 3.7.4, pytest-5.2.0, py-1.8.0, pluggy-0.13.0
Django settings: django_fixtures.settings (from ini file)
rootdir: /django_fixtures, inifile: pytest.ini
plugins: django-3.5.1
collected 1 item

test.py .
```

Great! The command completed successfully and your test passed. You now know how to access the database in tests. You also injected a fixture into a test case along the way.



Creating Fixtures for Django Models

Now that you're familiar with Django and `pytest`, write a test to check that a password set with `set_password()` is validated as expected. Replace the contents of `test.py` with this test:

Python

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

def test_should_check_password(db) -> None:
    user = User.objects.create_user("A")
    user.set_password("secret")
    assert user.check_password("secret") is True

def test_should_not_check_unusable_password(db) -> None:
    user = User.objects.create_user("A")
    user.set_password("secret")
    user.set_unusable_password()
    assert user.check_password("secret") is False
```

The first test checks that a user with a usable password is being validated by Django. The second test checks an edge case in which the user's password is unusable and should not be validated by Django.

There's an important distinction to be made here: The test cases above don't test `create_user()`. They test `set_password()`. That means a change to `create_user()` should not affect these test cases.

Also, notice that the `User` instance is created twice, once for each test case. A large project can have many tests that require a `User` instance. If every test case will create its own user, you might have trouble in the future if the `User` model changes.

To reuse an object in many test cases, you can create a test fixture:

Python

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

@pytest.fixture
def user_A(db) -> User:
    return User.objects.create_user("A")

def test_should_check_password(db, user_A: User) -> None:
    user_A.set_password("secret")
    assert user_A.check_password("secret") is True

def test_should_not_check_unusable_password(db, user_A: User) -> None:
    user_A.set_password("secret")
    user_A.set_unusable_password()
    assert user_A.check_password("secret") is False
```

In the above code, you created a function called `user_A()` that creates and returns a new `User` instance. To mark the function as a fixture, you decorated it with the `pytest.fixture` decorator. Once a function is marked as a fixture, it can be injected into test cases. In this case, you injected the fixture `user_A` into two test cases.

Maintaining Fixtures When Requirements Change

Let's say you've added a new requirement to your application, and now every user must belong to a special `"app_user"` group. Users in that group can view and update their own personal details. To test your app, you need your test users to belong to the `"app_user"` group as well:

Python

```python
import pytest
from django.contrib.auth.models import User, Group, Permission

@pytest.fixture
def user_A(db) -> Group:
    group = Group.objects.create(name="app_user")
    change_user_permissions = Permission.objects.filter(
        codename__in=["change_user", "view_user"],
    )
    group.permissions.add(*change_user_permissions)
    user = User.objects.create_user("A")
    user.groups.add(group)
    return user

def test_should_create_user(user_A: User) -> None:
    assert user_A.username == "A"

def test_user_is_in_app_user_group(user_A: User) -> None:
    assert user_A.groups.filter(name="app_user").exists()
```

Inside the fixture you created the group `"app_user"` and added the relevant `change_user` and `view_user` permissions to it. You then created the test user and added them to the `"app_user"` group.

Previously, you needed to go over every test case that created a user and add it to the group. Using fixtures, you were able to make the change just once. Once you changed the fixture, the same change appeared in every test case you injected `user_A` into. **Using fixtures, you can avoid repetition and make your tests more maintainable.**

Injecting Fixtures Into Other Fixtures

Large applications usually have more than just one user, and it's often necessary to test them with multiple users. In this situation, you can add another fixture to create the test `user_B`:

Python

```python
import pytest
from django.contrib.auth.models import User, Group, Permission

@pytest.fixture
def user_A(db) -> User:
    group = Group.objects.create(name="app_user")
    change_user_permissions = Permission.objects.filter(
        codename__in=["change_user", "view_user"],
    )
    group.permissions.add(*change_user_permissions)
    user = User.objects.create_user("A")
    user.groups.add(group)
    return user

@pytest.fixture
def user_B(db) -> User:
    group = Group.objects.create(name="app_user")
    change_user_permissions = Permission.objects.filter(
        codename__in=["change_user", "view_user"],
    )
    group.permissions.add(*change_user_permissions)
    user = User.objects.create_user("B")
    user.groups.add(group)
    return user

def test_should_create_two_users(user_A: User, user_B: User) -> None:
    assert user_A.pk != user_B.pk
```

In your terminal, try running the test:

Shell

```
$ pytest test.py
==================== test session starts =================================
platform linux -- Python 3.7.4, pytest-5.2.0, py-1.8.0, pluggy-0.13.0
Django settings: django_fixtures.settings (from ini file)
rootdir: /django_fixtures, inifile: pytest.ini
plugins: django-3.5.1
collected 1 item

test.py E
                              [100%]
============================= ERRORS ======================================
_____________ ERROR at setup of test_should_create_two_users ______________

self = <django.db.backends.utils.CursorWrapper object at 0x7fc6ad1df210>,
sql ='INSERT INTO "auth_group" ("name") VALUES (%s) RETURNING "auth_group"."id"'
,params = ('app_user',)

    def _execute(self, sql, params, *ignored_wrapper_args):
        self.db.validate_no_broken_transaction()
        with self.db.wrap_database_errors:
            if params is None:
                # params default might be backend specific.
                return self.cursor.execute(sql)
            else:
>               return self.cursor.execute(sql, params)
E               psycopg2.IntegrityError: duplicate key value violates
                unique constraint "auth_group_name_key"
E               DETAIL:  Key (name)=(app_user) already exists.

======================== 1 error in 4.14s ================================
```

The new test throws an `IntegrityError`. The error message originates from the database, so it might look a bit different depending on the database you are using. According to the error message, the test violates the unique constraint on the group's name. When you look at your fixtures, it makes sense. The `"app_user"` group is created twice, once in the fixture `user_A` and once again in the fixture `user_B`.

An interesting observation we've overlooked to this point is that the fixture `user_A` is using the fixture `db`. This means that **fixtures can be injected into other fixtures**. You can use this feature to address the `IntegrityError` above. Create the `"app_user"` group just once in a fixture, and inject it into both the `user_A` and `user_B` fixtures.

To do so, refactor your test and add an `"app user"` group fixture:

Python

```python
import pytest
from django.contrib.auth.models import User, Group, Permission

@pytest.fixture
def app_user_group(db) -> Group:
    group = Group.objects.create(name="app_user")
    change_user_permissions = Permission.objects.filter(
        codename__in=["change_user", "view_user"],
    )
    group.permissions.add(*change_user_permissions)
    return group

@pytest.fixture
def user_A(db, app_user_group: Group) -> User:
    user = User.objects.create_user("A")
    user.groups.add(app_user_group)
    return user

@pytest.fixture
def user_B(db, app_user_group: Group) -> User:
    user = User.objects.create_user("B")
    user.groups.add(app_user_group)
    return user

def test_should_create_two_users(user_A: User, user_B: User) -> None:
    assert user_A.pk != user_B.pk
```

In your terminal, run your tests:

Shell

```
$ pytest test.py
================================== test session starts ===============
platform linux -- Python 3.7.4, pytest-5.2.0, py-1.8.0, pluggy-0.13.0
Django settings: django_fixtures.settings (from ini file)
rootdir: /django_fixtures, inifile: pytest.ini
plugins: django-3.5.1
collected 1 item

test.py .
```

Amazing! Your tests pass. The group fixture encapsulates the logic related to the `"app_user"` group, such as setting permissions. You then injected the group into two separate user fixtures. By constructing your fixtures this way, you've made your tests less complicated to read and maintain.

Using a Factory

So far, you've created objects with very few arguments. However, some objects may be more complicated, featuring many arguments with many possible values. For such objects, you might want to create several test fixtures.

For example, if you provide all arguments to `create_user()`, this is what the fixture would look like:

Python

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

@pytest.fixture
def user_A(db, app_user_group: Group) -> User
    user = User.objects.create_user(
        username="A",
        password="secret",
        first_name="haki",
        last_name="benita",
        email="me@hakibenita.com",
        is_staff=False,
        is_superuser=False,
        is_active=True,
    )
    user.groups.add(app_user_group)
    return user
```

Your fixture just got a lot more complicated! A user instance can now have many different variations, such as superuser, staff user, inactive staff user, and inactive regular user.

In previous sections, you learned that it can be hard to maintain complicated setup logic in each test fixture. So, to avoid having to repeat all the values every time you create a user, add a function that uses `create_user()` to create a user according to your app's specific needs:

Python

```python
from typing import List, Optional
from django.contrib.auth.models import User, Group

def create_app_user(
    username: str,
    password: Optional[str] = None,
    first_name: Optional[str] = "first name",
    last_name: Optional[str] = "last name",
    email: Optional[str] = "foo@bar.com",
    is_staff: str = False,
    is_superuser: str = False,
    is_active: str = True,
    groups: List[Group] = [],
) -> User:
    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,
    )
    user.groups.add(*groups)
    return user
```

The function creates an app user. Each argument is set with a sensible default according to your app's specific requirements. For example, your app might require that every user has an email address, but Django's built-in function does not enforce such a restriction. You can enforce that requirement in your function instead.

Functions and classes that create objects are often referred to as **factories**. Why? It's because these functions act as factories that produce instances of a specific class. For more about factories in Python, check out The Factory Method Pattern and Its Implementation in Python.

The function above is a straightforward implementation of a factory. It holds no state and it's not implementing any complicated logic. You can refactor your tests so that they use the factory function to create user instances in your fixtures:

Python

```python
@pytest.fixture
def user_A(db, app_user_group: Group) -> User:
    return create_user(username="A", groups=[app_user_group])

@pytest.fixture
def user_B(db, app_user_group: Group) -> User:
    return create_user(username="B", groups=[app_user_group])

def test_should_create_user(user_A: User, app_user_group: Group) -> None:
    assert user_A.username == "A"
    assert user_A.email == "foo@bar.com"
    assert user_A.groups.filter(pk=app_user_group.pk).exists()

def test_should_create_two_users(user_A: User, user_B: User) -> None:
    assert user_A.pk != user_B.pk
```

Your fixtures got shorter, and your tests are now more resilient to change. For example, if you used a custom user model and you just added a new field to the model, you would only need to change `create_user()` for your tests to work as expected.

Using Factories as Fixtures

Complicated setup logic makes it harder to write and maintain tests, making the entire suite fragile and less resilient to change. So far, you've addressed this issue by creating fixtures, creating dependencies between fixtures, and using a factory to abstract as much of the setup logic as possible.

But there is still some setup logic left in your test fixtures:

Python

```python
@pytest.fixture
def user_A(db, app_user_group: Group) -> User:
    return create_user(username="A", groups=[app_user_group])

@pytest.fixture
def user_B(db, app_user_group: Group) -> User:
    return create_user(username="B", groups=[app_user_group])
```

Both fixtures are injected with `app_user_group`. This is currently necessary because the factory function `create_user()` does not have access to the `app_user_group` fixture. Having this setup logic in each test makes it harder to make changes, and it's more likely to be overlooked in future tests. Instead, you want to encapsulate the *entire* process of creating a user and abstract it from the tests. This way, you can focus on the scenario at hand rather than setting up unique test data.

To provide the user factory with access to the `app_user_group` fixture, you can use a pattern called factory as fixture:

Python

```python
from typing import List, Optional

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

@pytest.fixture
def app_user_group(db) -> Group:
    group = Group.objects.create(name="app_user")
    change_user_permissions = Permission.objects.filter(
        codename__in=["change_user", "view_user"],
    )
    group.permissions.add(*change_user_permissions)
    return group

@pytest.fixture
def app_user_factory(db, app_user_group: Group):
    # Closure
    def create_app_user(
        username: str,
        password: Optional[str] = None,
        first_name: Optional[str] = "first name",
        last_name: Optional[str] = "last name",
        email: Optional[str] = "foo@bar.com",
        is_staff: str = False,
        is_superuser: str = False,
        is_active: str = True,
        groups: List[Group] = [],
    ) -> User:
        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,
        )
        user.groups.add(app_user_group)
        # Add additional groups, if provided.
        user.groups.add(*groups)
        return user
    return create_app_user
```

This is not far from what you've already done, so let's break it down:

- The `app_user_group` fixture remains the same. It creates the special `"app user"` group with all the necessary permissions.
- A new fixture called `app_user_factory` is added, and it is injected with the `app_user_group` fixture.
- The fixture `app_user_factory` creates a closure and returns an inner function called `create_app_user()`.
- `create_app_user()` is similar to the function you previously implemented, but now it has access to the fixture `app_user_group`. With access to the group, you can now add users to `app_user_group` in the factory function.

To use the `app_user_factory` fixture, inject it into another fixture and use it to create a user instance:

Python

```python
@pytest.fixture
def user_A(db, app_user_factory) -> User:
    return app_user_factory("A")

@pytest.fixture
def user_B(db, app_user_factory) -> User:
    return app_user_factory("B")

def test_should_create_user_in_app_user_group(
    user_A: User,
    app_user_group: Group,
) -> None:
    assert user_A.groups.filter(pk=app_user_group.pk).exists()

def test_should_create_two_users(user_A: User, user_B: User) -> None:
    assert user_A.pk != user_B.pk
```

Notice that, unlike before, the fixture you created is providing a *function* rather than an *object*. This is the main concept behind the factory as fixture pattern: **The factory fixture creates a closure, which provides the inner function with access to fixtures.**

For more about closures in Python, check out Python Inner Functions — What Are They Good For?

Now that you have your factories and fixtures, this is the complete code for your test:

Python

```python
from typing import List, Optional

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

@pytest.fixture
def app_user_group(db) -> Group:
    group = Group.objects.create(name="app_user")
    change_user_permissions = Permission.objects.filter(
        codename__in=["change_user", "view_user"],
    )
    group.permissions.add(*change_user_permissions)
    return group

@pytest.fixture
def app_user_factory(db, app_user_group: Group):
    # Closure
    def create_app_user(
        username: str,
        password: Optional[str] = None,
        first_name: Optional[str] = "first name",
        last_name: Optional[str] = "last name",
        email: Optional[str] = "foo@bar.com",
        is_staff: str = False,
        is_superuser: str = False,
        is_active: str = True,
        groups: List[Group] = [],
    ) -> User:
        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,
        )
        user.groups.add(app_user_group)
        # Add additional groups, if provided.
        user.groups.add(*groups)
        return user
    return create_app_user

@pytest.fixture
def user_A(db, app_user_factory) -> User:
    return app_user_factory("A")

@pytest.fixture
def user_B(db, app_user_factory) -> User:
    return app_user_factory("B")

def test_should_create_user_in_app_user_group(
    user_A: User,
    app_user_group: Group,
) -> None:
    assert user_A.groups.filter(pk=app_user_group.pk).exists()

def test_should_create_two_users(user_A: User, user_B: User) -> None:
    assert user_A.pk != user_B.pk
```

Open the terminal and run the test:

Shell

```
$ pytest test.py
======================== test session starts ========================
platform linux -- Python 3.8.1, pytest-5.3.3, py-1.8.1, pluggy-0.13.1
django: settings: django_fixtures.settings (from ini)
rootdir: /django_fixtures/django_fixtures, inifile: pytest.ini
plugins: django-3.8.0
collected 2 items

test.py ..                                                    [100%]

======================== 2 passed in 0.17s ==========================
```

Great job! You've successfully implemented the factory as fixture pattern in your tests.

Factories as Fixtures in Practice

The factory as fixture pattern is very useful. So useful, in fact, that you can find it in the fixtures provided by `pytest` itself. For example, the `tmp_path` fixture provided by `pytest` is created by the fixture factory `tmp_path_factory`. Likewise, the `tmpdir` fixture is created by the fixture factory `tmpdir_factory`.

Mastering the factory as fixture pattern can eliminate many of the headaches associated with writing and maintaining tests.

Conclusion

You've successfully implemented a fixture factory that provides Django model instances. You've also maintained and implemented dependencies between fixtures in a way that takes some of the hassle out of writing and maintaining tests.

In this tutorial, you've learned:

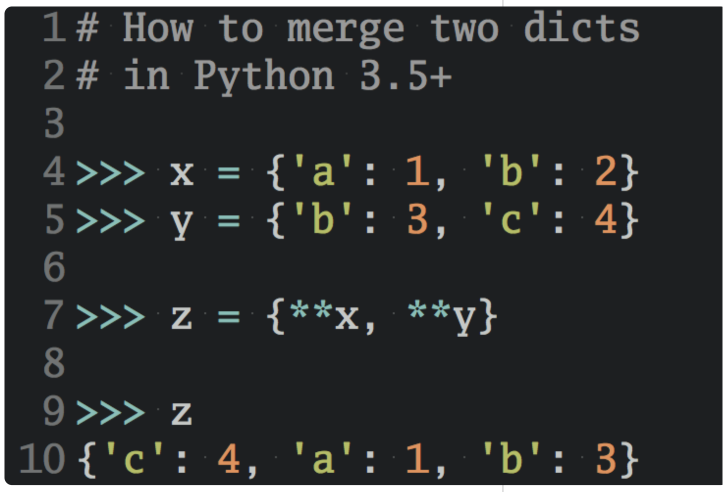
- How to create and load **fixtures** in Django
- How to provide **test fixtures for Django models** in `pytest`
- How to use **factories** to create fixtures for Django models in `pytest`
- How to implement the **factory as fixture** pattern to create dependencies between test fixtures

You're now able to implement and maintain a solid test suite that will help you produce better and more reliable code, faster!

Mark as Completed

🐍 Python Tricks 💌

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```