Fixtures and factories

by Hector Canto

We will talk about

- **fixture** concept
- How to build fixtures with:
  - **factoryboy**: Mother pattern
  - **faker**: data generator
- Make it work in **pytest**

What is a fixture

“”””A test fixture is a device used to consistently test some item, device, or piece of software. Test fixtures are used in the testing of electronics, software and physical devices.”””

- wikipedia.com/en/Test_fixture

Why is this important

- Testing should be a big chunk of our daily work
- Testing is hard and costly
- Let’s make it easier
- Making new tests should become easier with time

Testing structure: AAA

- **Arrange**
- Act - Execute
- Assert

Arranging

- In arrange phase we prepare the test
  - data to input
  - data to be “there”
  - the system to act
  - secondary systems to interact

Data fixtures

We are going to focus on data fixtures for

- inputs
- expectancies

AAA in python unitest

```python
class TestExample(unittest.Case):

    def setUp(self):
        ...

    def test_one(self):
        dummy_user = ExampleUserFactory()
        self.db.save(dummy_user)
        ...
        result = SystemUnderTest()
        ...
        self.assertTrue(result)

    def tearDown(self):
        ...
```

In pytest

Any test dependency
usually set as parameter or decorator

```python
import pytest

@pytest.fixture(autouse=True, scope="session")
def global_fixture():
    ...

pytestmark = pytest.mark.usefixtures("module_fixture")

@pytest.mark.usefixtures("fixture_as_decorator")
def test_one(fixture_as_param):
    ...
```

AAA in pytest

```python
@pytest.fixture(scope="module", name="arranged", autouse=False)
def arrange_fixtures():
    ... # set up
    yield "value"
    ... # tear down

def test_using_fixture_explicitly(arranged):
    result = SystemUnderTest(arranged)
    assert result is True
    ...
```

Data fixtures can be

- Inputs
- Configuration
- Data present in DB, cache files,
- Params to query
- A dependency to inject

Data fixtures can be (II)

- A mock or dummy to use or inject
- Set the application’s state
- Ready the system under test
- The system under test ready for assertion
- Revert or clean-up procedures

Where to put fixtures:

- in the same place as the test
  - but it makes the IDE angry
- in the closest `conftest.py`
  - `conftest` is pytest’s `__init__.py`
  - makes fixture globally available downstream

Fixture example

```python
import random

@pytest.fixture(name="cool_fixture")
def this_name_is_just_for_the_function():
    yield random.randint()

def test_using_fixture(cool_fixture):
    system_under_test(param=cool_fixture)
```

Test name fixture

```python
def test_one(request):
    test_name = request.node.name
    result = system_under_test(test_name)
    assert result == test_name
```

Anti-patterns

- Copy-paste the same dict for each test
- Have a thousand JSON files

Recommendations

- Generate them programmatically
- In the test, highlight the difference
- Use Mother pattern and data generators

Enter factory-boy

```python
import factory

class UserMother(factory.DictFactory):
    firstname = "Hector"
    lastname = "Canto"
    address = "Praza do Rei, 1, Vigo CP 36000"
```

Enter faker

```python
class UserMother(factory.DictFactory):
    firstname = factory.Faker('first_name')
    lastname = factory.Faker('last_name')
    address = factory.Faker('address')

random_user = UserMother()
random_user == {
    'firstname': 'Rebecca',
    'lastname': 'Sloan',
    'address': '52097 Daniel Ports Apt. 689\nPort Jeffrey, NM 55289'
}
```

More faker

```
random_user = UserMother(firstname="Guido")
random_user == {
   'firstname': 'Guido',
   'lastname': 'Deleon',
    'address': '870 Victoria Mills\nWilliamville, CA 44946'
}
```

Batch generation

```
UserMother.create_batch(size=5)
```

Iterated generation

```
many = UserMother.create_batch(size=10, firstname=factory.Iterator(["One", "Two", "Three"]))
many[0].firstname == "One"
many[2].firstname == "Three"
many[3].firstname == "One"
```

FactoryBoy with ORMs

- DjangoORM
- SQLAlchemy
- Mogo and MongoEngine
- not hard to create your own

Example

```python
class UserMother(factory.orm.SQLAlchemyFactory):
    class Meta:
        model = User
        sqlalchemy_session_factory = lambda: common.TestSession()

def test_with_specific_db(test_session):
    UserMother._meta.sqlalchmey_session = test_session
    # You can also set the live DB and populate it for demos
```

Set up for SQLA

```python
import factory
from sqlalchemy import orm

TestSession = orm.scoped_session(orm.sessionmaker())
"""Global scoped session (thread-safe) for tests"""

class BaseFactory(factory.alchemy.SQLAlchemyModelFactory):
    class Meta:
        abstract = True
        sqlalchemy_session = TestSession
        sqlalchemy_session_persistence = "flush"

class UserFactory(BaseFactory):
      class Meta:
          model = User
```

Get or Create Fixture

```python
# Create user once, use everywhere
class UserFactory(BaseUserFactory):
    class Meta:
        sqlalchemy_get_or_create = ('id',)

user1 = UserFactory(id=1, firstname="Dennis", lastname="Ritchie")
user2 = UserFactory(id=1)
user1 == user2
```

Good things about factory-boy

- Highly customizable
- Related factories
- Works with ORM

Bad things

- Inner Faker is weird
- Documentation gaps (as usual)
- A bit hard to work with relationships

For the future

- polyfactory
- make your own providers
- check out Maybe, Traits, post_gen hooks
- random seed fixing
- sequence resetting
- user `random.sample` and others

Tactical tips

- Add factories to your libraries
- Specially in serverless and microservices

Real case example

- 3 libraries: common models, common APIs, common 3rd party services
- On each we have factories to create
  - DB data
  - API callback bodies
  - event message payloads

Code samples

- `tests/user_factories.py`
  - Params
  - Related
  - Fuzzy
  - Lazy
- `/tests/conftest.py`

Register fixtures

```python
# tests/conftest/py
register(AdminFactory, "admin")
register(DbUserFactory, "user1", email="user1@hello.es")
register(DbUserFactory, "user2", status=0)
register(DbProfileFactory2, "profile")

# usage
def test_with_reg_fx(admin):
    assert admin.first_name == "Admin"
```