Testing efectivo con Pytest

@hectorcanto_dev

slideshare.net/HectorCanto

PyConEs 2021

Objetivo

Nos centraremos en *testing unitario con Pytest*, pero muchas cosas también sirven para otros frameworks y lenguajes

Por qué pytest

Es un framework maduro y muy potente

Muy bien documentado y compatible con muchas herramientas

Por qué efectividad en testing

El Testing ocupa buena parte de nuestro tiempo programando

- Mejor testing, mejor Software
- Cuanto más efectivos, más tiempo para otras cosas

Testear es importante

- Garantiza que todo el código funciona cuando desarrollamos mejoras
- Documenta con ejemplos y da contexto
- Es un entorno muy útil para depuración
- **Habilita la refactorización**

Artículo: Tips and tricks for unit tests

Efectividad significa

- Tests rápidos y manejables
- Desarrollo fácil y código mantenible
- Accesible y leíble para todo el mundo

Como conseguimos esta efectividad

- Utilizando pytest en toda su potencia
- Entendiendo la teoría: mocks, fixtures, parametrización
- Aprovechamiento de librerías y plugins
- Buenas prácticas generales de código
- Estrategias y Sentido común

Fases de un test

Los test se dividen en 3 fases conocidas como la triple AAA

- Arrange: Preparación
- Act: Ejecución
- Assert: Validación

Propondremos ideas para las tres

Lanzar la Suite de tests

- Opciones de pytest
- Marcadores
- Selección de test

Solo lanzaremos lo que necesitemos

Opciones de pytest

```
pytest --help
pytest tests/folder/test_file.py::test_name
pytest -m smoke
pytest -k users
```

Podemos filtrara de muchas maneras

Suite: repetir test fallidos

```
pytest --fail-first
pytest --last-failed
pytest --failed-first
```

Daremos prioridad a los tests fallidos

```
Plugin: pytest-xdist
```

Suite: tests específicos

```
def test_users_creation():
    ...

def test_users_update():
    ...
```

```
pytest -k users
```

El *naming* es importante

Suite: marcadores

```
@pytest.mark.slow
@pytest.mark.current
def test_this():
    ...
```

```
pytest -m current -s  -v
pytest -m "slow and not integration"
pytest -m "smoke and unit"
```

Ejemplos: smoke, unit, integration, current, slow

Suite: marcadores globales

```
# Per module
pytestmark = pytest.mark.auth
pytestmark = [pytest.mark.deletion, pytest.mark.api]

# Per class
class TestClass:
    pytestmark = pytest.mark.special
```

Es fácil marcar grupos de tests

Suite: structure

```
tree tests/ --dirsfirst

tests/
├── smoke/
└── unit/
|   ├── service/
|   └── persistance/
├── integration/
|
├── fixtures/
├── factories/
├── conftest.py
├── aux.py
└── __init__.py
```

Suite: marcadores automáticos

```python
def pytest_collection_modifyitems(items):
    for item in items:
        if "/smoke/" in str(item.module):
            item.add_marker("smoke")
```

Coloca esto en `tests/conftest.py`

Recomendación: smoke tests

- Comprobar config y entorno
- Instanciación con valores por defecto
- Errores y casos simples

Los lanzaremos los primeros, local y CI

Evitaremos sustos y ahorraremos tiempo

Suite: ordenación

```
@pytest.mark.first
@pytest.mark.second
@pytest.mark.last
```

El order es importante para `smoke` y CI

Dejaremos los lentos para el final

```
Plugins: pytest-ordering
```

Suite: entorno

```
# setup.cfg
[tool:pytest]
env =
    PYTHONBREAKPOINT=ipdb.set_trace
    APP_ENVIRONMENT=test
    CACHE=memory
    DEBUG=1
    VAR=value
```

Separamos entornos local y de test

```
Plugin: pytest-env
```

Suite: entorno II

```
def test_with_different_env_vars(monkeypatch):
    monkeypatch.setenv("CACHE", "nocache")
    monkeypatch.delenv("VAR")
```

```
Fixture: monkeypatch
```

Setup

- Parametrization
- Fixtures
- Factories

También *Arrange* o Preparación

Parametrización

```python
@pytest.mark.parametrize("entry, expected", (
    (1, True),
    (2, False),
    (None, False),
    ("hello", False),
)
def test_something(entry, expected: bool):
    result = function_under_test(entry)
    assert result == expected
```

En vez de hacer un test para cada caso

Reutilizaremos un test para todas las entradas

``

Fixtures

Fixtures es el conjunto de elementos que establecemos para crear un entorno concreto.

- los datos que preparamos
- El sistema en un estado concreto
- Elementos activos con el comportamiento “trucado”

Pytest fixture

```
@pytest.fixture
def example_fixture()
    now = datetime.utcnow()
    do_setup(now)
    yield now
    do_teardown()  # clean up
```

Fixtures con setup, teardown o ambos

Ejemplo de fixture

```python
@pytest.fixture
def load_data(db_client):
    my_user = User(name="Hector", last_name="Canto")
    db_client.add(my_user)
    yield my_user  # usable as parameter
    db_client.delete(my_user)
```

Usar una fixture

```
@pytest.mark.usefixtures("load_data")
def test_with_fixtures(example_fixture):
     result, error = system_under_test(example_fixture)
     assert result
     assert not error
```

Colocad vuestras fixtures en cualquier `conftest.py`

para no tener que importarlas

Reutilizar fixtures

```
@pytest.fixture(scope="session")
def test_settings():
     yield get_settings(test=True)
```

Scopes: sesión, módulo, clase o función

Fixtures automáticas

```python
@pytest.fixture(autouse=True)
def clean_db(db_client):
    yield db_client
    for table in db_client.tables:
        db_client.truncate()
```

Se autoejecuta sola

Se apoya en otra fixture anterior

Fixtures de datos

```
Recomendación: crear datos para fixtures *programáticamente*
```

- Constantes que importamos
- Funciones para generar
- Factorías
- Evitar JSON crudos, archivos de texto …

Setup: Factory

```python
from factory import StubFactory

class ObjectFactory(StubFactory):
    name: str = "value"

my_ojb = ObjectFactory()
assert my_obj.name == "value"

other = ObjectFactory(name="another value")
assert other.name == "another value"
```

```
Librería & plugin: factoryboy, pytest-factoryboy
```

Factory & Faker

```python
from factoryboy import Faker, SelfAttribute
from factory.fuzzy import FuzzyInteger

class ObjectFactory(StubFactory):
    name: str = Faker("first_name_male")
    last_name: str = Faker("last_name")
    full_name: str = SelfAttribute(lambda self: f"{self.name} {se
    phone: str = Faker("phone_number", local="en_GB")
    money: int = FuzzyInteger(1, 1000)
```

Genera valores a discreción

```
Librería: Faker
```

Model Factory

Se puede usar con Django, SQLAlchemy y Pymongo

```
from factory import alchemy, RelatedFactory

class UserFactory(alchemy.SQLAlchemyModelFactory):
    class Meta:
        model = User
        sqlalchemy_session = Session

    id = LazyFunction(lambda: randint(1, 1_000_000))
    location = RelatedFactory(LocationFactory)
    created_at = Faker(
        "unix_time",
        start_datetime=datetime(2015, 1, 1),
        end_datetime=datetime(2019, 12, 31)
    )
```

Batches & Dict Factory

```
factory.build(dict, FACTORY_CLASS=UserFactory)
UserFactory.create_batch(5)

class UserDictFactory(DictFactory)
  ...
```

Genera *inputs* para tu API o función

Ejecución o Act

- Controlar el tiempo
- Test doubles: mocks y familia

Freezegun

```python
from datetime import datetime
from freezegun import freeze_time

@freeze_time("2012-01-14")
def test_2012():
    assert datetime.now() == datetime(2012, 1, 14)

    with freeze_time("2018-01-01"):
        assert datetime.now() > datetime(2012, 1, 4)
```

Para el tiempo a tu antojo

```
Library: freezegun
```

Freezer

```python
from datetime import datetime

def test_freezer_move(freezer):
    now = datetime.now()
    freezer.move_to('2017-05-20')
    later = datetime.now()
    assert now != later

@pytest.mark.freeze_time('2017-05-21')
def test_freeze_decorator():
    assert datetime.now() == datetime(2017, 5, 21)
```

```
Fixture: freezer
```

Otras librerías temporales

timeago - moment - pytime - arrow

```
timeago.format(timedelta(seconds = 60 * 3.4))  # 3 minutes ago
moment.date(2012, 12, 19).add(hours=1, minutes=2, seconds=3)  # n
pytime.next_month('2015-10-1')  # again, no-more-deltas
arrow.utcnow().span('hour')  # 2 datetimes in one line
```

Ejecución: test doubles

Los Test doubles substituyen a algún elemento activo con el comportamiento deseado

Tipos de test doubles

- Dummy: objeto que se pasa pero no se usa
- Fake: Implementación simplificada que funciona
  - fake server, in-memory cache
- Stubs: Respuestas prefabricadas, reemplazo total
- Mocks: Reemplazo parcial, respeta interfaces
- Spies: No intercepta, solo registra

`martinfowler.com/bliki/TestDouble.html`

Mock library

La mayoría de nuestros dobles se implementan con mock

```
Library: mock - Plugin: pytest-mock  - Fixture: mocker
```

Python mocks

```
def test_mock_patching(mocker):
    url = "https://2021.es.pycon.org/"
    mocked = mocker.patch.object(requests, "get", return_value="i
    mocker.patch.object(requests, "post", side_effect=ForbiddenEr
    response = requests.get(url)
    assert response == "intercepted"
    assert mocked.called_once()
```

Stubs

```
def test_stubbing(monkeypatch):

    def mock_exist(value):
        print(f"{value} exists")
        return True

    monkeypatch.setattr(os.path, 'exists', mock_exist)
    assert os.path.exists("/believe/me/I/exist")
```

Espía

```python
def test_with_spy(mocker):
    url = "https://2021.es.pycon.org/"
    spy = mocker.patch.object(requests, "get", wraps=requests.get
    response = requests.get(url)
    assert response.status_code -= 200
    spy.assert_called_once(), spy.mock_calls
    spy.assert_called_with(url)
```

No intercepta, solo registra llamadas

Interceptores

```python
def test_with_http_interceptor(requests_mock):
    # arranges
    url = "http://tests.com"
    requests_mock.get(url, json={"key": "value"})
    # action
    response = requests.get(url)

    assert "key" in response.json()
```

Intercepta llamadas y devuelve lo que quieras

```
Library: requests_mock
```

Validación

```
def test_one():
    expected = 5
    result = system_under_test()
    assert result
    assert result is not None
    assert result == expected
    assert result > 3
```

Validamos sentencias lógicas

Mensajes de error

```
response = requests.get(url)
assert response.json() == expected, response.text()
```

En caso de `AssertionError`, exponemos info extra

Comparaciones

```
import pytest, math

assert 2.2 == pytest.approx(2.3, 0.1)
assert math.isclose(2.2, 2.20001, rel_tol=0.01)
```

No perdáis el tiempo con errores de bulto y redondeos

Comparaciones: listas y sets

```
my_list = [1, 1, 2, 3, 4]
other_list = [4, 3, 2, 1]

list_without_duplicates = list(set(my_list))
diff = set(my_list) ^ set(other_list)
assert not diff
```

‘No os volváis locos con valores repetidos y bucles comparando listas

]# Comparaciones: dicts

Comparar diccionarios y listas es duro.

```
from deepdiff import DeepDiff

def test_dicts(parameter, expected):
    result = system_under_test(parameter)
    diff = DeepDiff(result, expected, exclude_paths=(f"root['upda
    assert not diff, diff
```

No perdáis el tiempo ordenando dicts y calculando borrando elementos difíciles c

```
Library: deepdiff
```

Comprobaciones pospuestas

```
def test_delayed_response(requests_mock):
    url = "http://tests.com"
    requests_mock.get(url, json={"key": "value"})

    response = requests.get(url)

    expect(response.status_code == 200, response.status_code)
    expect(response.json() == {}, response.text)
    assert_expectations()
```

```
Library: delayed-assert
```

Delayed assert prompt

```
    def assert_expectations():
        'raise an assert if there are any failed expectations'
        if _failed_expectations:
>           assert False, _report_failures()
E           AssertionError:
E
E           assert_expectations() called at
E           "/home/hector/Code/personal/effective_testing/tests/u
E
E           Failed Expectations : 1
E
E           1: Failed at "/home/hector/Code/personal/effective_te
E               ErrorMessage:    {"key": "value"}
E               expect(response.status_code == 200, response.stat

../../../.local/share/virtualenvs/effective_testing-OL_w0ui8/lib/
```

Validar logs

Los logs son ultra-importantes

- Monitorización y Métricas
- Flujo de programa y control de errores
- Especialmente importante en microservicios y serverless

Captura logs

```
def test_log_capture(request, caplog):
    logger = logging.getLogger(request.node.name)
    caplog.set_level("INFO")
    dtt = "2021-10-02 10:55:00"
    msg = "captured message"
    expected_message = f"{dtt} INFO module:{request.node.name} {m

    with freeze_time(dtt):
   logger.info(msg)

    with caplog.disabled():
    logger.info ("Any log here will not be captured")
    some_function_with_logs()
    breakpoint()

    with freeze_time(dtt):
```

```
Fixture: caplog, capsys, capfd
```

Bola extra: tests y debugger

- Los tests son pequeños entornos que controlamos
- Debug en máquinas reales es difícil o directamente imposible

```
pip install ipdb
PYTHONBREAKPOINT=ipdb.set_trace pytest -m current -s
```

```
Usa los puntos de ruptura del IDE
```

Ideas finales

- “”Los tests son un pérdida de tiempo””””
  - Realmente, nos ahorran mucho tiempo
- Tratar los tests como ciudadanos de primera:
  - style, docstrings, comments …
- No hayh proyecto pequeño para tener tests
- Pensad en vuestro yo del futuro

Recomendaciones

- Si algo os cuesta mucho…
  - seguro que hay una librería o receta que lo hace por vosotros
- Leed y releed la documentación
- Los frameworks son vuestros amigos

Summary

Fixtures: monkeypatch, mocker, requests_mock, caplog, parametrize, mark

Plugins: env, ordering, xdist

Librerías: factoryboy, faker, deepdict, freezegun, moment, ipdb

https://docs.pytest.org/
https://docs.pytest.org/en/latest/reference/plugin_list.html

Se ha quedado fuera

Antipatrones, AWS, Docker, DBs, frameworks, asyncio

https://github.com/spulec/moto
https://github.com/localstack/localstack
http://blog.codepipes.com/testing/software-testing-antipatterns.html
https://www.yegor256.com/2018/12/11/unit-testing-anti-patterns.html

Gracias

Espero que os gustara :)