TDD Pythonowych Mikroserwisów

Michał Bultrowicz

O mnie

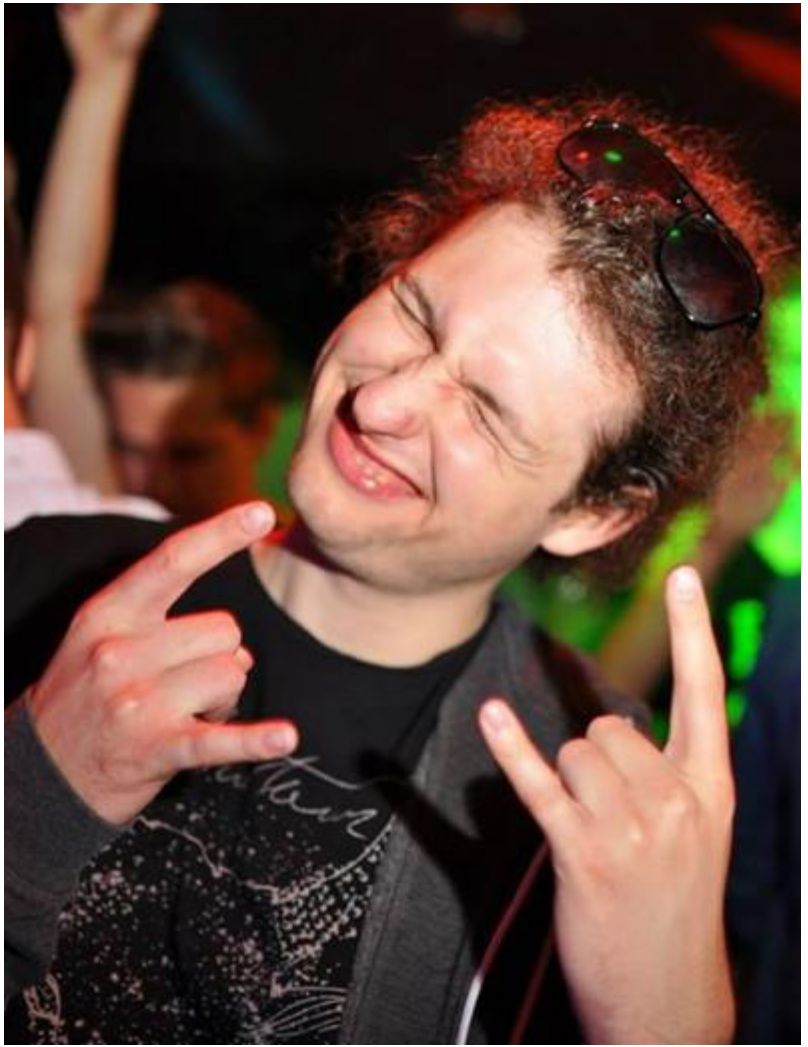
Imię: Michał

Nazwisko: Bultrowicz

Praca: Brak

Dodatkowe informacje:

- Lubię Pythona



Mikroserwisy:

- serwisy
- małe
- niezależne
- współpracujące

Twelve-Factor App (http://12factor.net/)

1. One codebase tracked in revision control, many deploys
2. Explicitly declare and isolate dependencies
3. Store config in the environment
4. Treat backing services as attached resources
5. Strictly separate build and run stages
6. Execute the app as one or more stateless processes
7. Export services via port binding
8. Scale out via the process model
9. Maximize robustness with fast startup and graceful shutdown
10. Keep development, staging, and production as similar as possible
11. Treat logs as event streams
12. Run admin/management tasks as one-off processes

Rada na przyszłość

- Zaczynajcie od monolitu.
- Wydzielanie mikroserwisów powinno być naturalne.

TESTY!

Testy

- Obecne w moim serwisie (jakieś 85% pokrycia).
- Nierzadko zagmatwane.
- Nie zapewniały, że aplikacja wstanie

Testy jednostkowe

- Obecne w moim serwisie (jakieś 85% pokrycia).
- Nierzadko zagmatwane.
- Nie zapewniały, że aplikacja wstanie

Testy całości aplikacji!

- Uruchamianie całego proces aplikacji.
- Aplikacja “nie wie”, że nie jest na produkcji.
- Odpalane lokalnie, przed commitem.
- Duża pewność poprawnego działania.
- Niezależność od czynników zewnętrznych.
- Potrzebne zewnętrzne serwisy i bazy danych.

Zewnętrzne serwisy lokalnie?

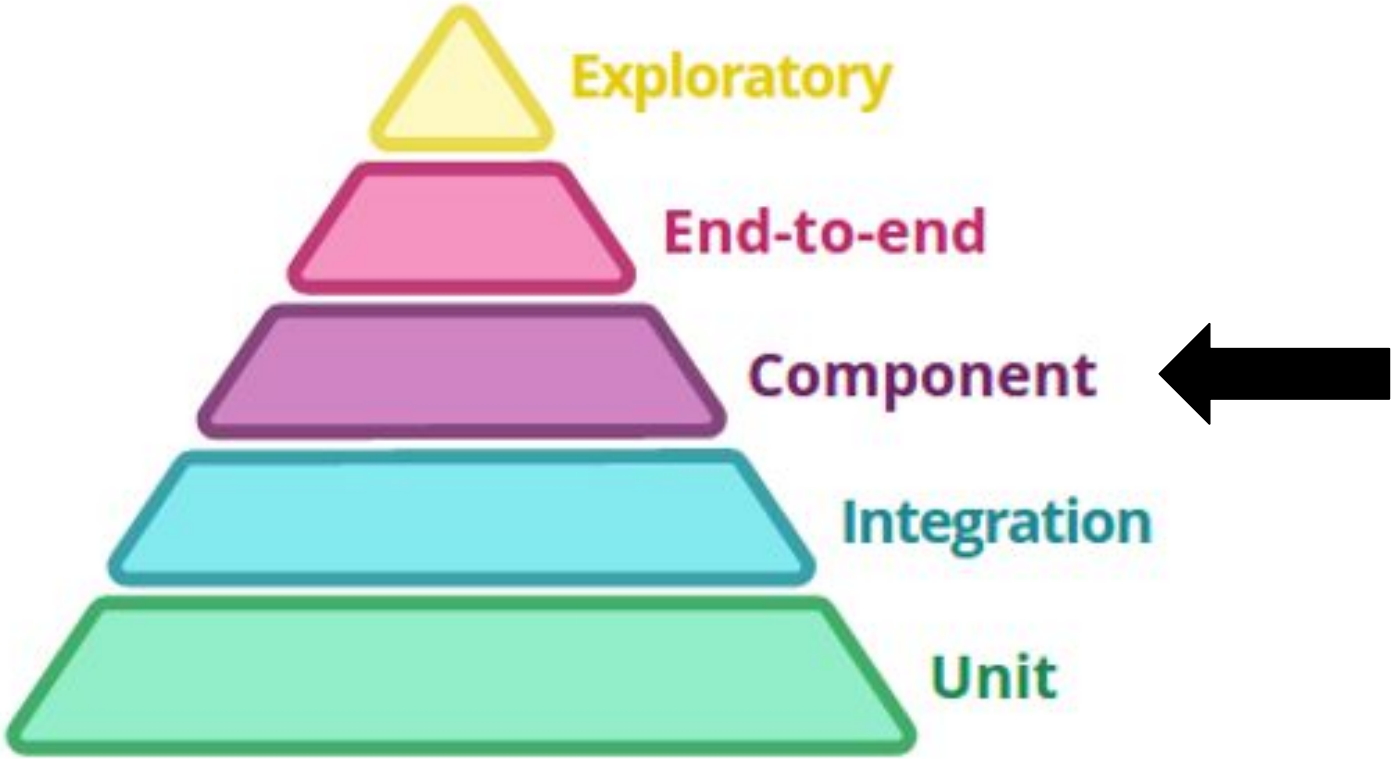
Mocki (stuby) serwisów:

- WireMock
- Pretenders (Python)
- Mountebank

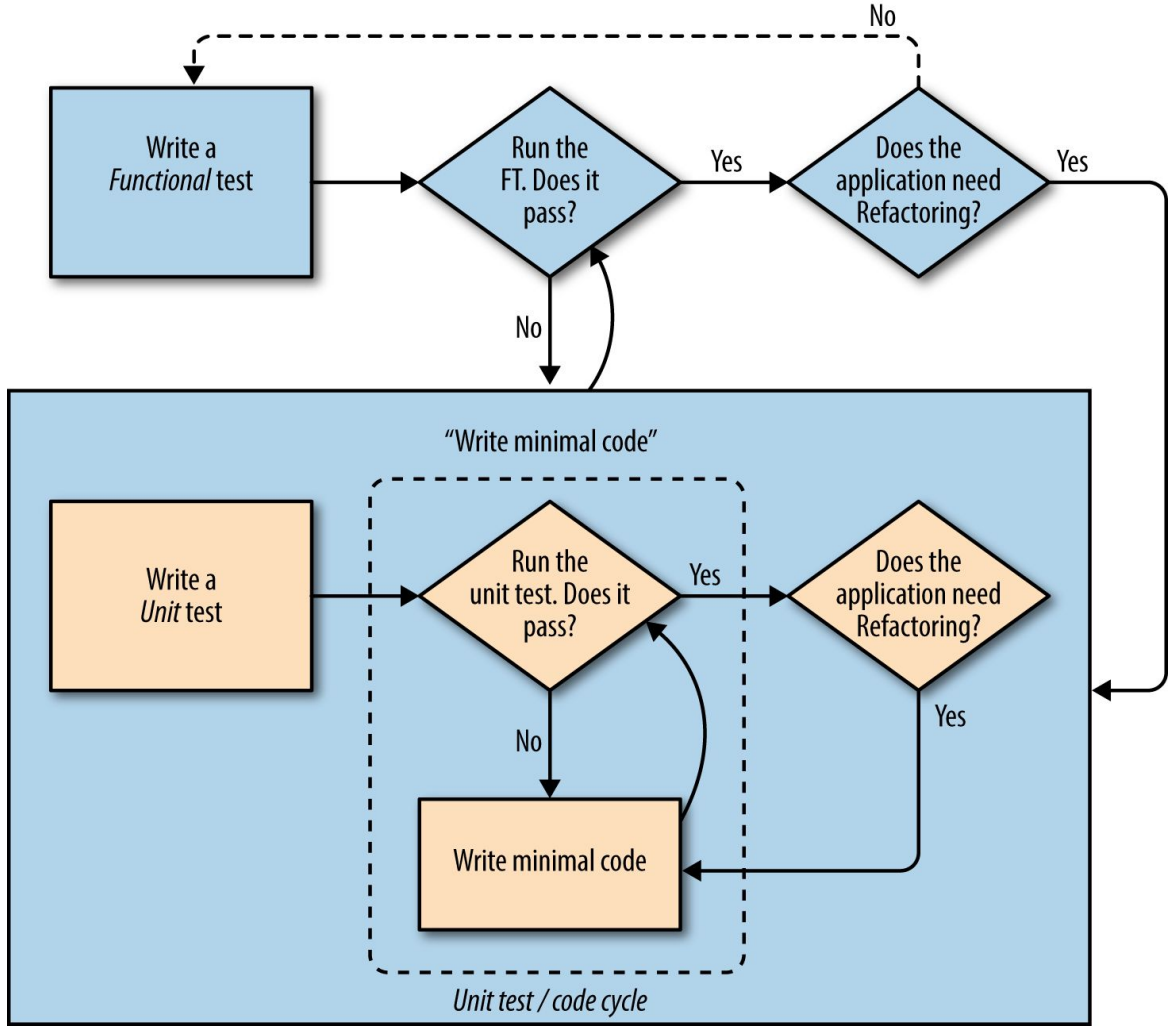
Bazy danych (i inne systemy) lokalnie?

- Tzw. verified fakes - rzadko spotykane
- Docker - po prostu stworzymy wszystko, co potrzebne

Narzędzia są.
Teraz ciut teorii.



http://martinfowler.com/articles/microservice-testing/#conclusion-test-pyramid



Harry J.W Percival, "Test Driven Development with Python"

TDD

Korzyści

- Pewność przy zmianach.
- Obrona przed złym designem.
- Automat sprawdza wszystko.

Wymagania:

- Dyscyplina
- Narzędzia

Misja: PyDAS

PyDAS

- Nowy (przepisany) serwis.
- Poligon dla moich eksperymentów.
- TDD pomoże.
- ...ostatecznie nie wyszedł idealny, ale dużo mnie nauczył

https://github.com/butla/pydas/tree/dev

Pytest

- Zwięzłość.
- Przejrzysta kompozycja “fixture’ów”.
- Kontrola nad tą kompozycją (w celu np. zmniejszenia czasu trwania testów)
- Przydatne raporty błędów.

Test serwisowy (pydas/tests/integrated/test_service.py)

```
def test_something(our_service, db):
    db.put(TEST_DB_ENTRY)
    response = requests.get(
        our_service.url + '/something',
        headers={'Authorization': TEST_AUTH_HEADER})
    assert response.status_code == 200
```

Fixture: db (pydas/tests/integrated/conftest.py)

```
import pytest, redis

@pytest.yield_fixture(scope='function')
def db(db_session):
    yield db_session
    db_session.flushdb()

@pytest.fixture(scope='session')
def db_session(redis_port):
    return redis.Redis(port=redis_port, db=0)
```

Fixture: db (pydas/tests/integrated/conftest.py)

```
import docker, pytest

@pytest.yield_fixture(scope='session')
def redis_port():
    docker_client = docker.Client(version='auto')
    download_image_if_missing(docker_client)
    container_id, redis_port = start_redis_container(docker_client)
    yield redis_port
    docker_client.remove_container(container_id, force=True)
```

Fixture: db (pydas/tests/integrated/conftest.py)

```
import port_for
def start_redis_container(docker_client):
    redis_port = port_for.select_random()
    host_config = docker_client.create_host_config(
        port_bindings={6379: redis_port})
    container_id = docker_client.create_container(
        'redis:2.8.22',
        host_config=host_config)['Id']

    docker_client.start(container_id)
    wait_for_redis(redis_port)
    return container_id, redis_port
```

Fixture: our_service (pydas/tests/integrated/conftest.py)

```
@pytest.fixture(scope='function')
def our_service(our_service_session, ext_service_impostor):
    return our_service
```

Mountepy

- Zarządza instancją Mountebanka
- Zarządza procesami serwisów
- https://github.com/butla/mountepy
- `$ pip install mountepy`

```
import mountepy

@pytest.yield_fixture(scope='session')
def our_service_session():
    service_command = [
        WAITRESS_BIN_PATH,
        '--port', '{port}',
        '--call', 'data_acquisition.app:get_app']

    service = mountepy.HttpService(
        service_command,
        env={
            'SOME_CONFIG_VALUE': 'blabla',
            'PORT': '{port}',
            'PYTHONPATH': PROJECT_ROOT_PATH})

    service.start()
    yield service
    service.stop()
```

```
@pytest.yield_fixture(scope='function')
def ext_service_impostor(mountebank):
    impostor = mountebank.add_imposter_simple(
        port=EXT_SERV_STUB_PORT,
        path=EXT_SERV_PATH,
        method='POST')
    yield impostor
    impostor.destroy()

@pytest.yield_fixture(scope='session')
def mountebank():
    mb = Mountebank()
    mb.start()
    yield mb
    mb.stop()
```

Mamy narzędzia do testu serwisu!

```
(py34) butla@B2:~/development/pydas$ py.test tests/
========================== test session starts ==========================
platform linux -- Python 3.4.3, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /home/butla/development/pydas/tests, inifile:
collected 51 items

tests/integrated/test_req_store_integrated.py ...
tests/integrated/test_service.py ........
tests/unit/test_cf_app_utils_auth.py ..............
tests/unit/test_config.py ...
tests/unit/test_falcon_bravado.py .
tests/unit/test_req_store.py ....
tests/unit/test_resources.py ................

===================== 51 passed in 2.75 seconds =====================
```

Uwagi o testach serwisowych

- Będą dawać duży log błędu.
- Zepsucie fixture’a daje pokrętny log.
- Nie uchronią przed głupimi błędami (hardcode localhost)

Zmora commitów “innych ludzi”

Oręż

- Pokrycie testowe
- Analiza statyczna
- Testy kontraktowe

.coveragerc (z pydasa)

```
[report]

fail_under = 100

[run]

source = data_acquisition

parallel = true
```

Analiza statyczna

tox.ini (duże uproszczenie)

```
[testenv]

commands =

    coverage run -m py.test tests/

    coverage report -m

    /bin/bash -c "pylint data_acquisition --rcfile=.pylintrc"
```

Testy kontraktowe: homonto na interfejs

```
swagger: '2.0'
info:
  version: "0.0.1"
  title: Jakis interfejs
paths:
  /person/{id}:
    get:
      parameters:
        -
          name: id
          in: path
          required: true
          type: string
          format: uuid
      responses:
        '200':
          description: Successful response
          schema:
            title: Person
            type: object
            properties:
              name:
                type: string
              single:
                type: boolean
```

http://swagger.io/

Bravado (https://github.com/Yelp/bravado)

- Generuje klienta serwisu ze Swaggera
- Weryfikuje
  - Parametry
  - Zwracaną wartość
  - Status HTTP
- Konfigurowalne (nie musi wszystkiego weryfikować)

Zastosowanie Bravado

- W testach serwisu: zamiast requests
- W testach jednostkowych (z klientem testowym frameworku)
- Teraz to też testy kontraktowe.

```python
from tests.falcon_bravado import FalconTestHttpClient

def test_falcon_contract(falcon_api):
    with open(SPEC_FILE_PATH) as spec_file:
        swagger_spec = yaml.load(spec_file)
    client = SwaggerClient.from_spec(
        swagger_spec,
        http_client=FalconTestHttpClient(falcon_api))
    SwaggerAcquisitionRequest = client.get_model('AcquisitionRequest')
    request_body = SwaggerAcquisitionRequest(**TEST_DOWNLOAD_REQUEST)

    resp_object = client.rest.submitAcquisitionRequest(
        body=request_body).result()

    assert resp_object
```

Nasz ogródek posprzątany

...ale to nie cały system...

Więcej o testach / mikroserwisach

“Building Microservices”, O'Reilly

“Test Driven Development with Python”

http://martinfowler.com/articles/microservice-testing/

“Fast test, slow test” (https://youtu.be/RAxiiRPHS9k)

Komentarze bardzo mile widziane

Co było dobre?

Czego brakowało?

Dzięki!