TDD Pythonowych Mikroserwisów

Michał Bultrowicz

O mnie

- Nazwisko: Michał Bultrowicz
- Poprzedni pracodawcy: Intel… i to tyle
- Poprzednie stanowisko: techniczny “team-leader” w projekcie Trusted Analytics Platform
- Aktualne stanowisko: brak
- Strona: https://bultrowicz.com

Microservices:

- 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

Drobne ostrzeżenie

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

TESTY!

Testy

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

Testy JEDNOSTKOWE

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

Testy całości aplikacji!

- Uruchamianie pełnego procesu aplikacji.
- Aplikacja “nie wie”, że nie jest na produkcji.
- Odpalane lokalnie, przed commitem.
- Duża pewność, że aplikacja wstanie na produkcji.
- ...wymagają zewnętrznych serwisów i baz danych...

Zewnętrzne serwisy lokalnie?

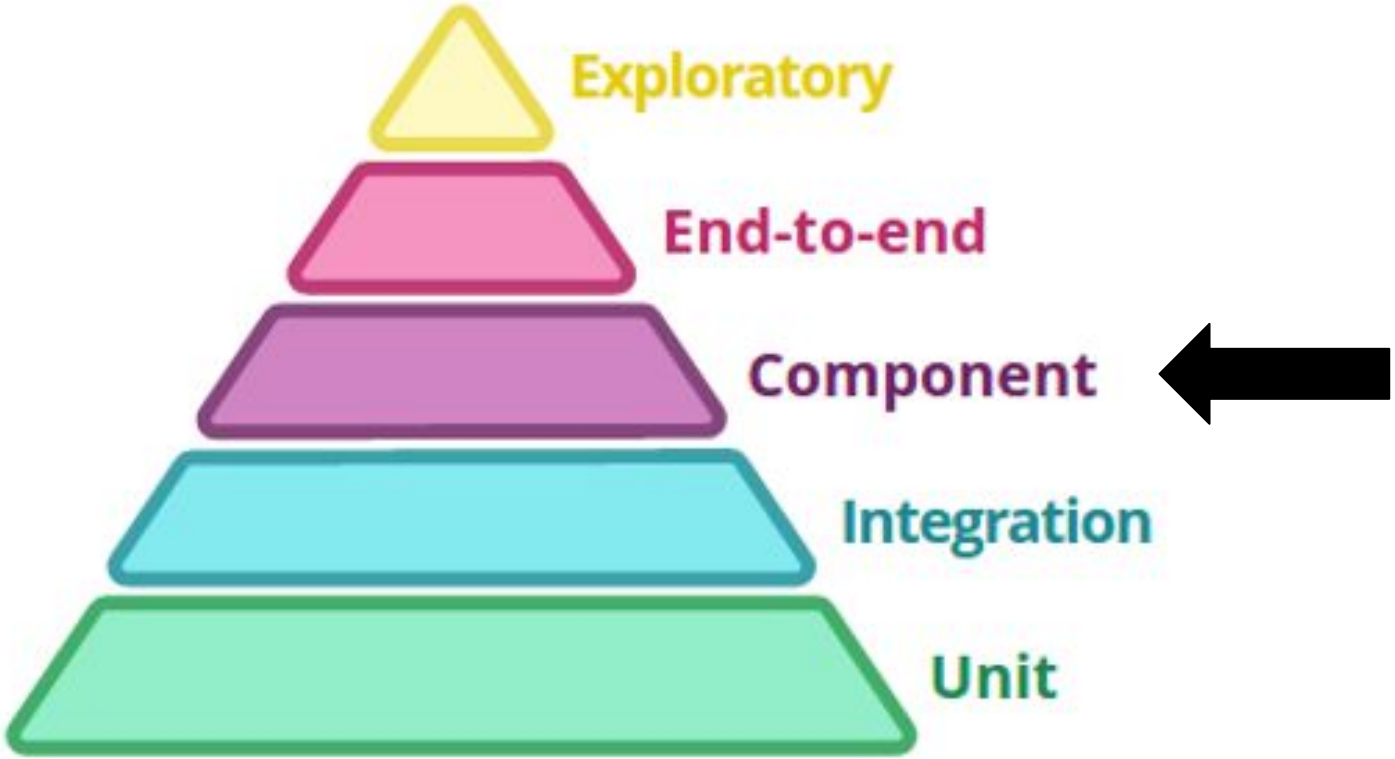
“Mockowanie” (i tworzenie “stubów”) serwisów:

- WireMock
- Pretenders (Python)
- Mountebank

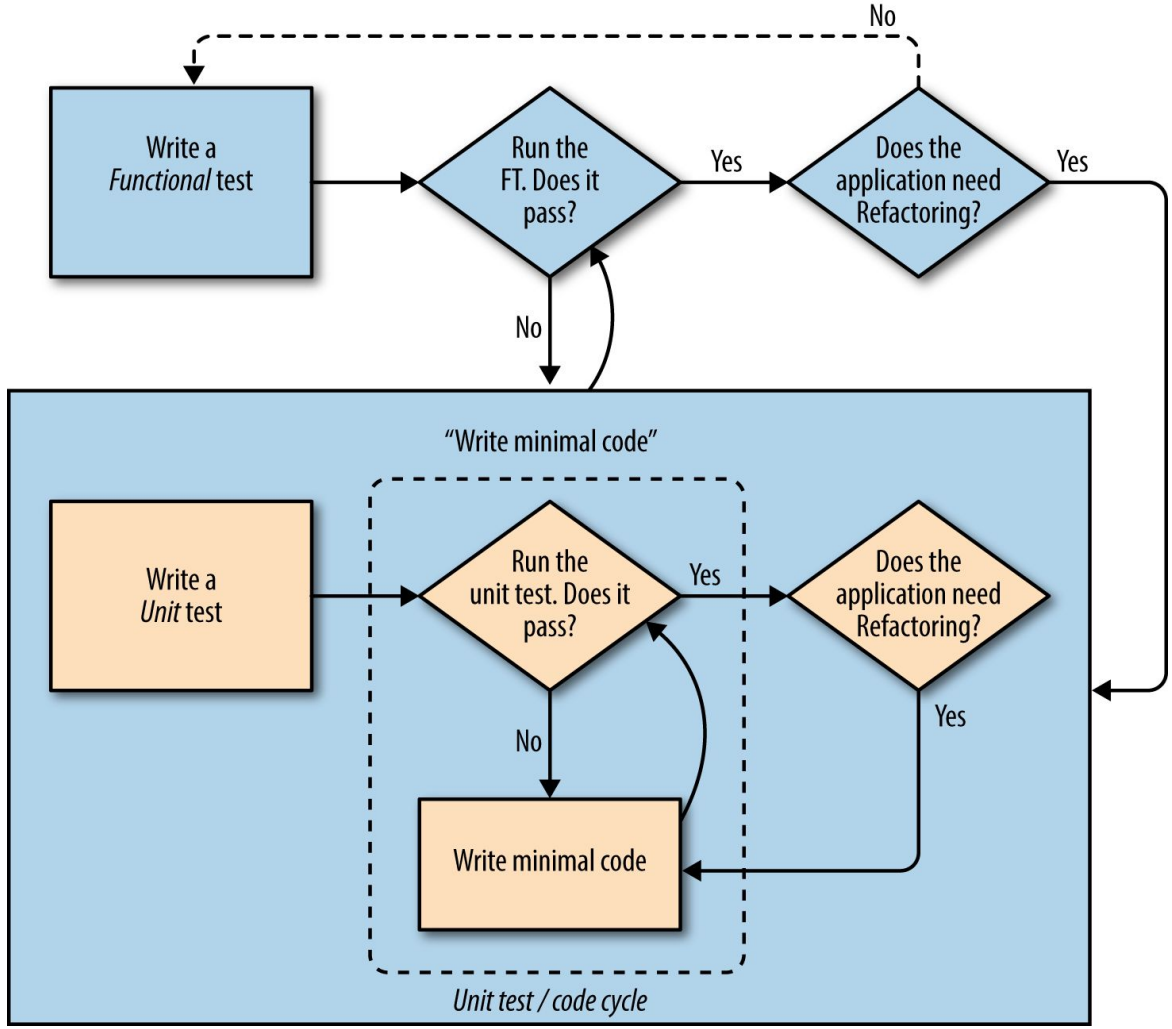
Bazy danych (i inne systemy) lokalnie?

- Normalnie - żmudna instalacja
- “Verified Fakes” - rzadko dostępne
- Docker - po prostu stwórz wszystko, czego potrzeba

Ciut teorii



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



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

TDD (tego właśnie potrzebowałem!)

Korzyści

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

Wymagania:

- Dyscyplina
- Narzędzia

Implementacja

PyDAS

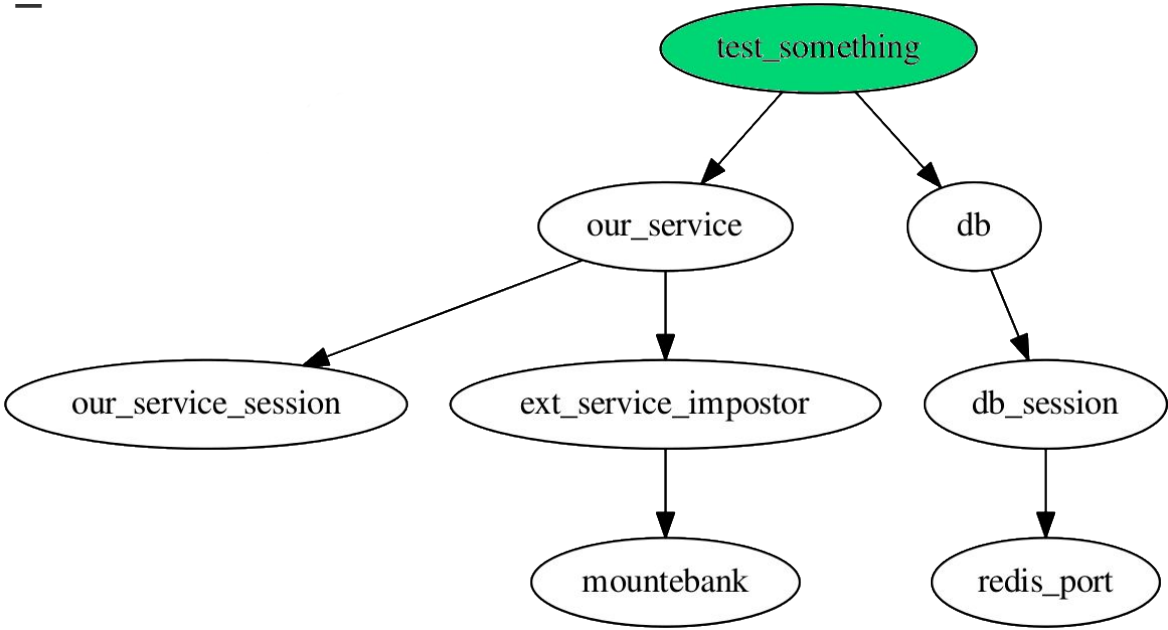
- Przepisany stary, problematyczny serwis (Java).
- TDD bardzo pomogło.
- Poligon doświadczalny.
- ...ostatecznie nie wyszedł idealny, ale dużo mnie nauczył

https://github.com/butla/pydas

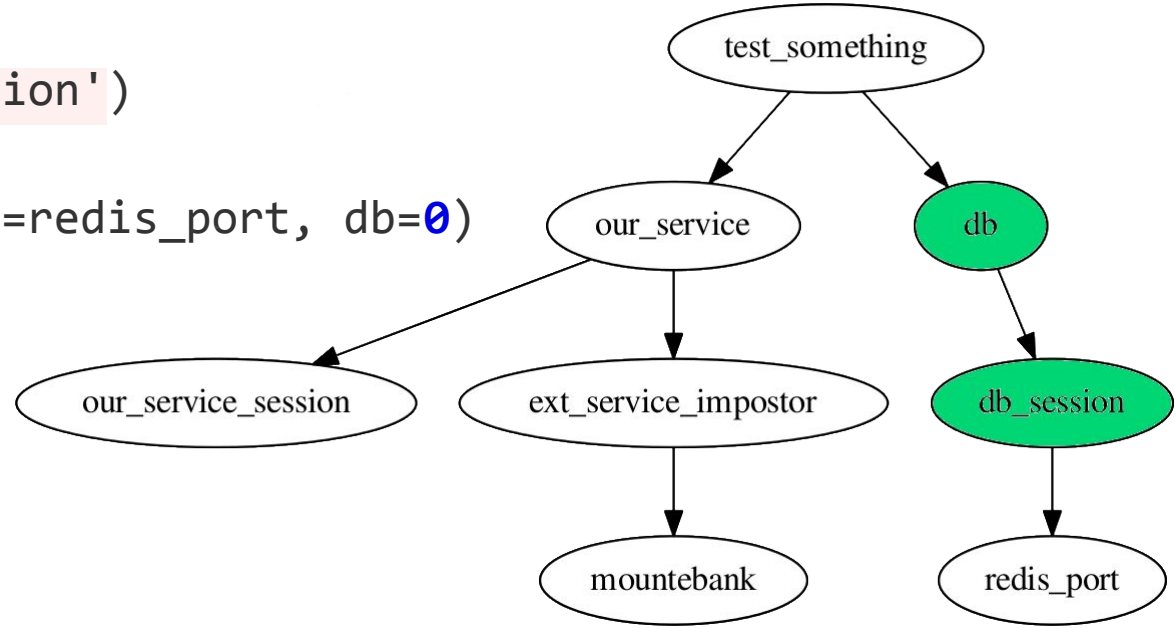
Pytest

- Zwięzły.
- Przejrzysta kompozycja “fixture’ów”.
- Kontrola tej kompozycji (np. do skrócenia testów).
- Pomocne raporty błędów.

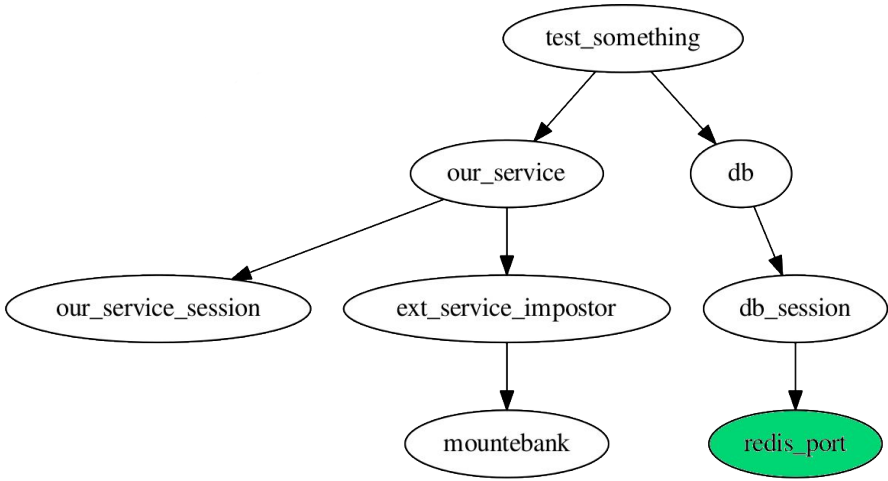
```python
def test_something(our_service, db):
    db.put(TEST_DB_ENTRY)
    response = requests.get(
        our_service.url + '/something',
        headers={'Authorization': TEST_AUTH_HEADER})
    assert response.status_code == 200
```



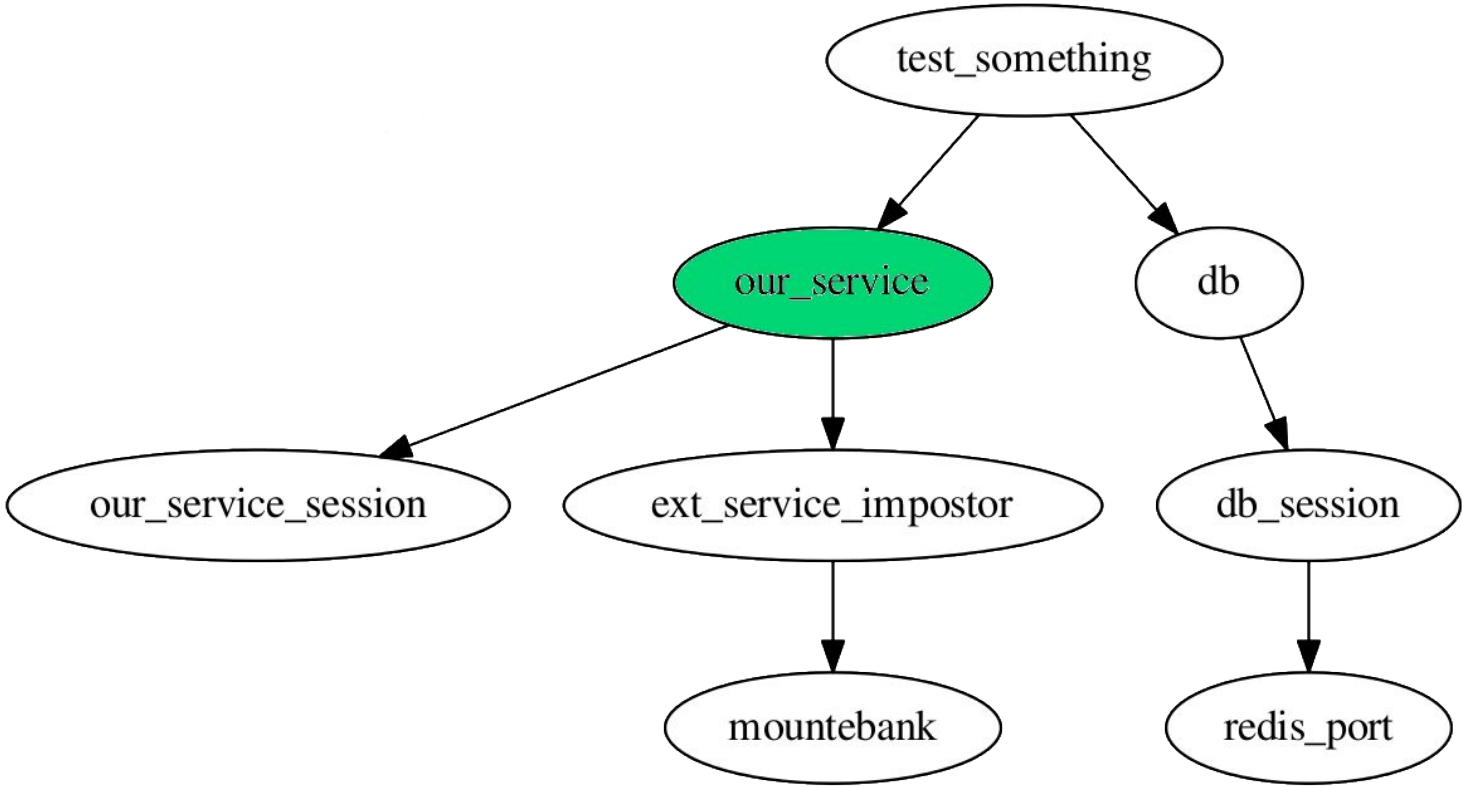
```python
import pytest, redis

@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)
```



```python
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)
```



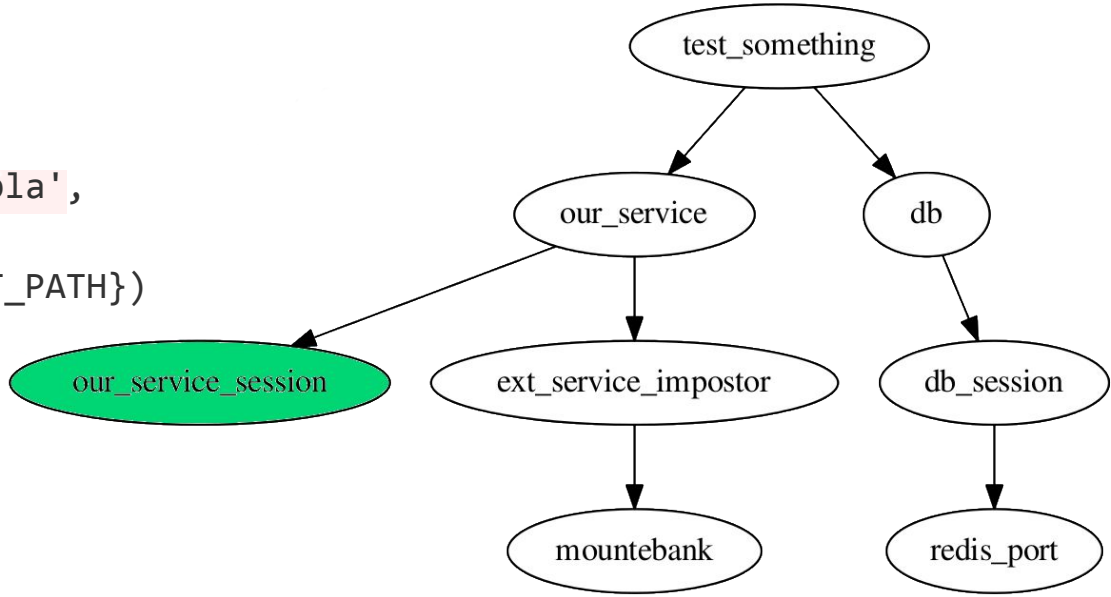
```python
@pytest.fixture(scope='function')
def our_service(our_service_session, ext_service_impostor):
    return our_service_session
```



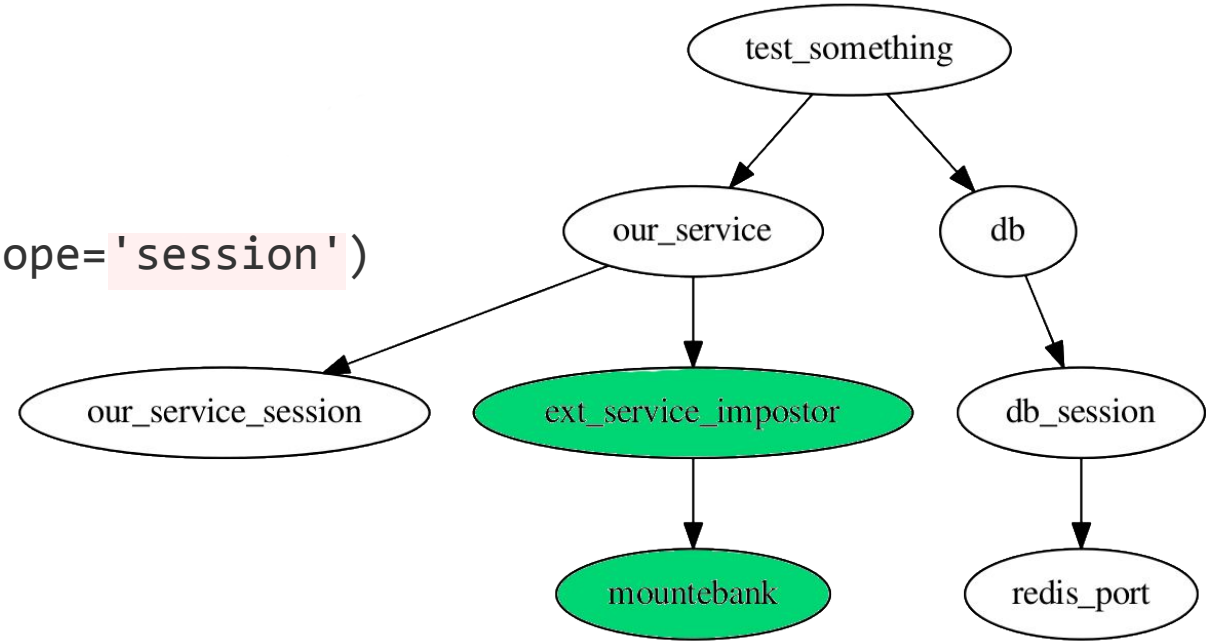
Mountepy

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

```python
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()
```



```python
@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()
```



Test(y) serwisowe gotowe!

```
(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

- Pomiar pokrycia testowego.
- Analiza statyczna (pylint, pyflakes, etc.).
- Testy kontraktowe.

.coveragerc (z PyDASa)

```
[report]

fail_under = 100

[run]

source = data_acquisition

parallel = true
```

http://coverage.readthedocs.io/en/coverage-4.0.3/subprocess.html

Analiza statyczna

tox.ini (simplified)

```
[testenv]

commands =

    coverage run -m py.test tests/

    coverage report -m

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

https://tox.readthedocs.io

Testy kontraktowe: homonto na interfejs

```
swagger: '2.0'
info:
  version: "0.0.1"
  title: Some interface
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/

Kontrakt oddzielony od kodu!

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

- Tworzy klienta serwisu ze Swaggera
- Weryfikuje
  - Parametry
  - Zwracane struktury danych
  - Zwracane kody HTTP
- Kofnigurowalne (nie musi być tak surowe)

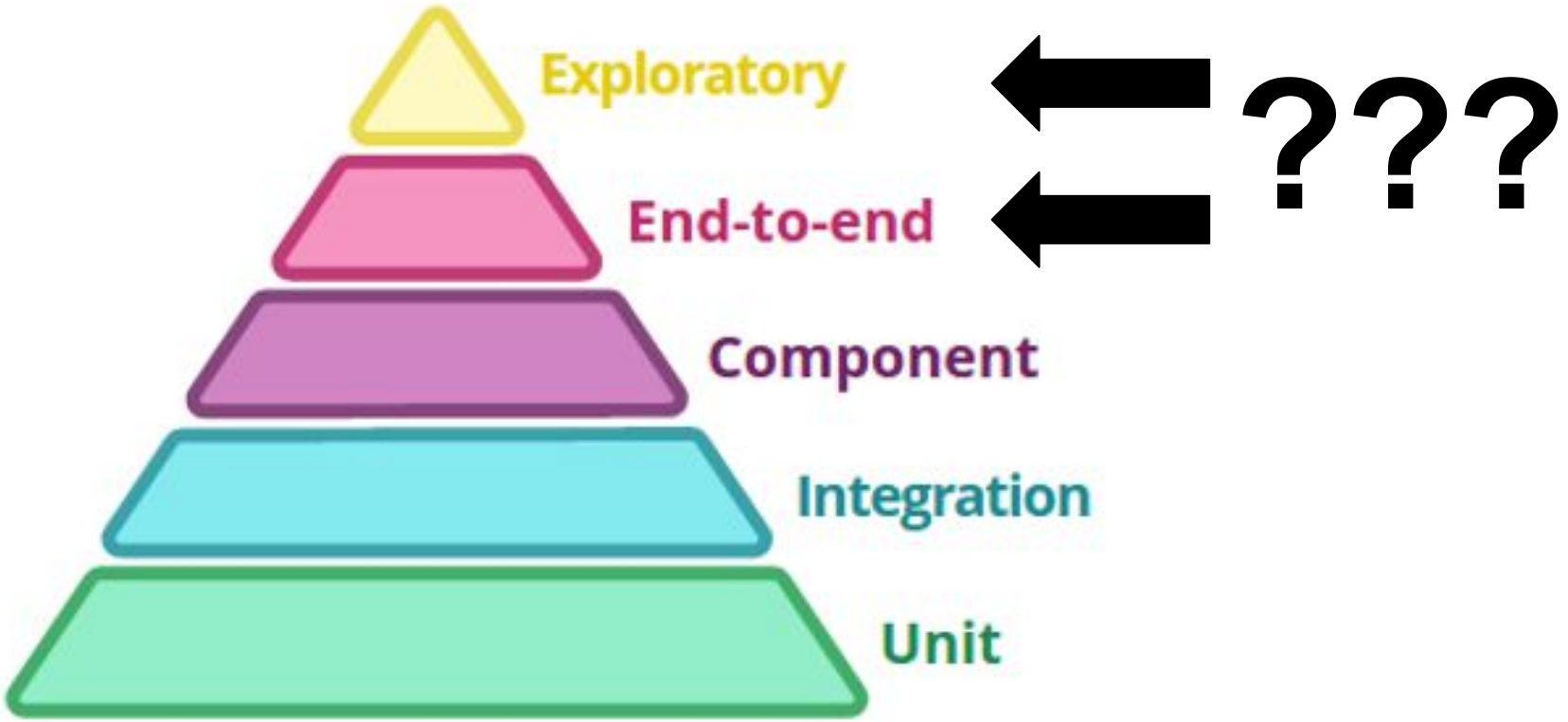
Zastosowanie Bravado

- W testach serwisowych: zamiast “requests”
- W testach jednostkowych:
  - https://github.com/butla/bravado-falcon
- Jedne i drugie zyskuję funkcję testów kontraktowych.

```python
from bravado.client import SwaggerClient
from bravado_falcon import FalconHttpClient
import yaml
import tests # our tests package

def test_contract_unit(swagger_spec):
    client = SwaggerClient.from_spec(
        swagger_spec,
        http_client=FalconHttpClient(tests.service.api))

    resp_object = client.v1.submitOperation(
        body={'name': 'make_sandwich', 'repeats': 3},
        worker='Mom').result()

    assert resp_object.status == 'whatever'

@pytest.fixture()
def swagger_spec():
    with open('api_spec.yaml') as spec_file:
        return yaml.load(spec_file)
```

```
def test_contract_service(swagger_spec, our_service):
    client = SwaggerClient.from_spec(
        swagger_spec,
        origin_url=our_service.url))

    request_options = {
        'headers': {'authorization': A_VALID_TOKEN},
    }

    resp_object = client.v1.submitOperation(
        body={'name': 'make_sandwich', 'repeats': 3},
        worker='Mom',
        _request_options=requet_options).result()

    assert resp_object.status == 'whatever'
```

Nasz ogródek ogarnięty...


Exploratory
End-to-end
Component
Integration
Unit
???

Więcej o testach, mikroserwisach itp.

“Building Microservices”, O'Reilly

“Test Driven Development with Python”

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

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

Building Service interfaces with OpenAPI / Swagger (EP2016)

System Testing with pytest and docker-py (EP2016)