TDD of Python Microservices

Michał Bultrowicz

About me

- Name: Michał Bultrowicz
- Previous employers: Intel… that’s the full list
- Previous occupation: technical team-leader on Trusted Analytics Platform project
- Current Occupation: N/A
- “Website”: https://github.com/butla (...working on a blog...)

Microservices:

- services
- micro
- independent
- cooperating

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

Word of advice

- Start with a monolith.
- Separating out microservices should be natural.

TESTS!

Tests

- Present in my service (around 85% coverage).
- Sometimes convoluted.
- Didn’t ensure that the service will even get up.

UNIT tests

- Present in my service (around 85% coverage).
- Sometimes convoluted.
- Didn’t ensure that the service will even get up

Tests of the entire application!

- Run the whole app’s process.
- App “doesn’t know” it’s not in production.
- Run locally, before a commit.
- High confidence that the app will get up.
- ...require external services and data bases...

External services locally?

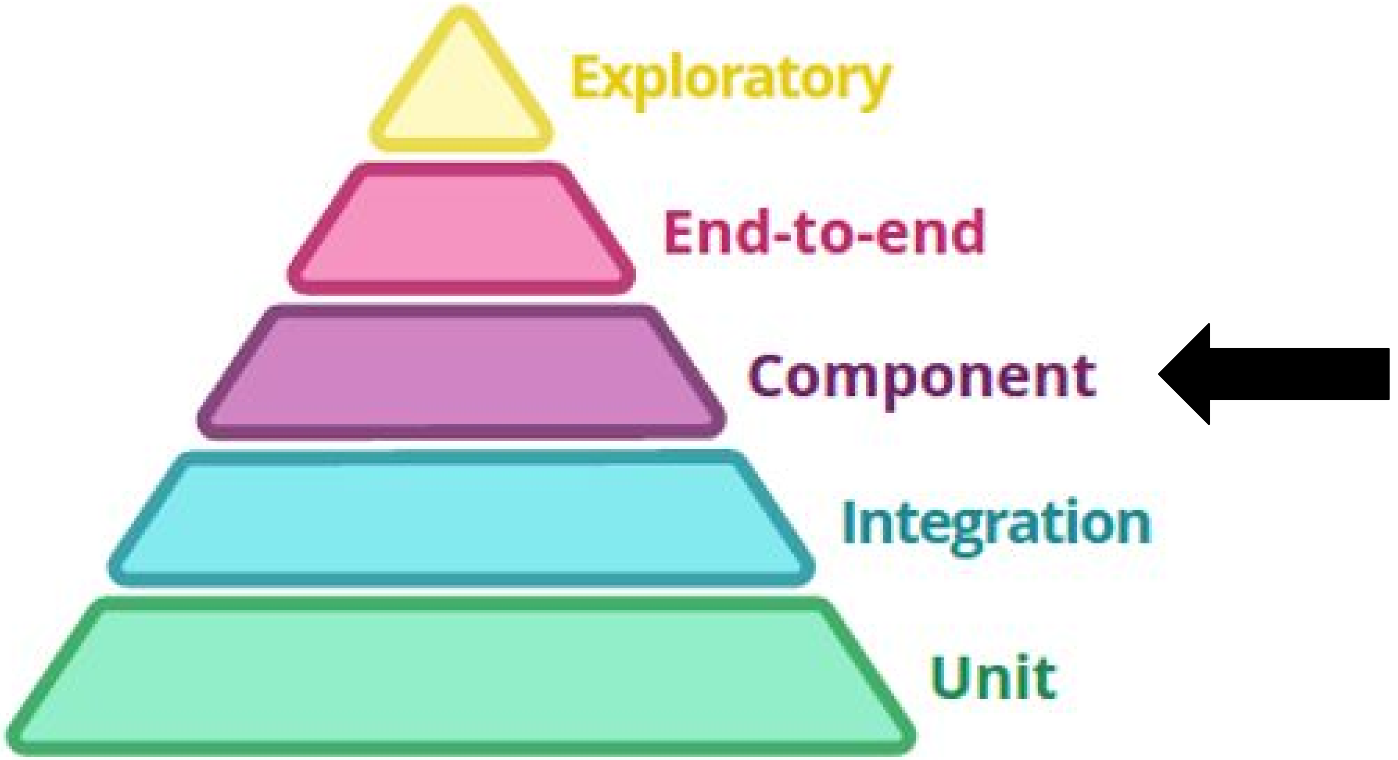
Service mocks (and stubs):

- WireMock
- Pretenders (Python)
- Mountebank

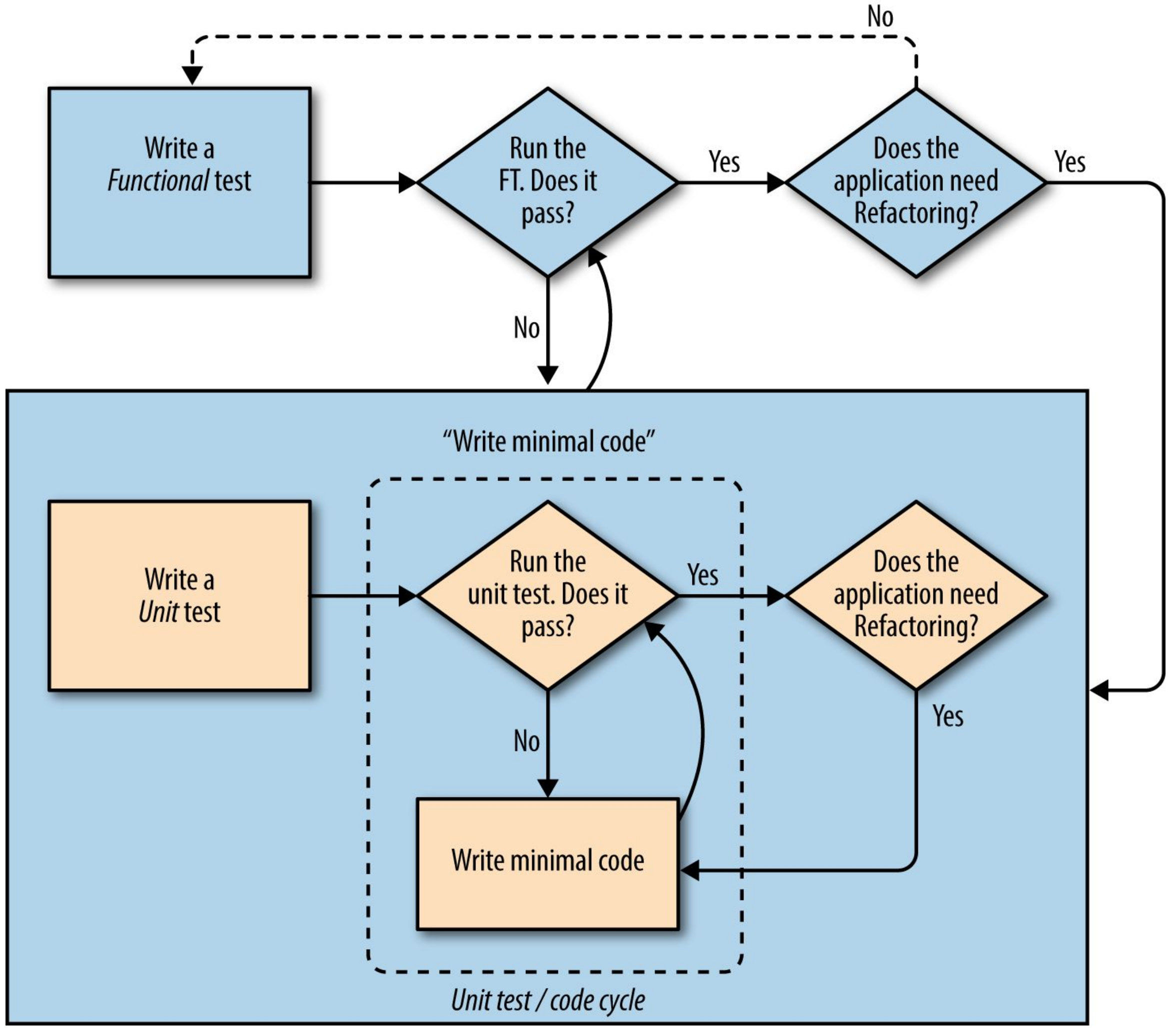
Data bases (and other systems) locally?

- Normally - a tiresome setup
- Verified Fakes - rarely seen
- Docker - just create everything you need

Now some theory



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


No
Write a *Functional* test
Run the FT. Does it pass?
Yes
Does the application need Refactoring?
Yes
No
"Write minimal code"
Write a *Unit* test
Run the unit test. Does it pass?
Yes
Does the application need Refactoring?
Yes
No
Write minimal code
Unit test / code cycle

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

TDD (the thing I needed!)

Pros:

- Confidence in face of changes.
- Automation checks everything.
- Ward off bad design.

Requirements:

- Discipline
- Tools

Implementation

PyDAS

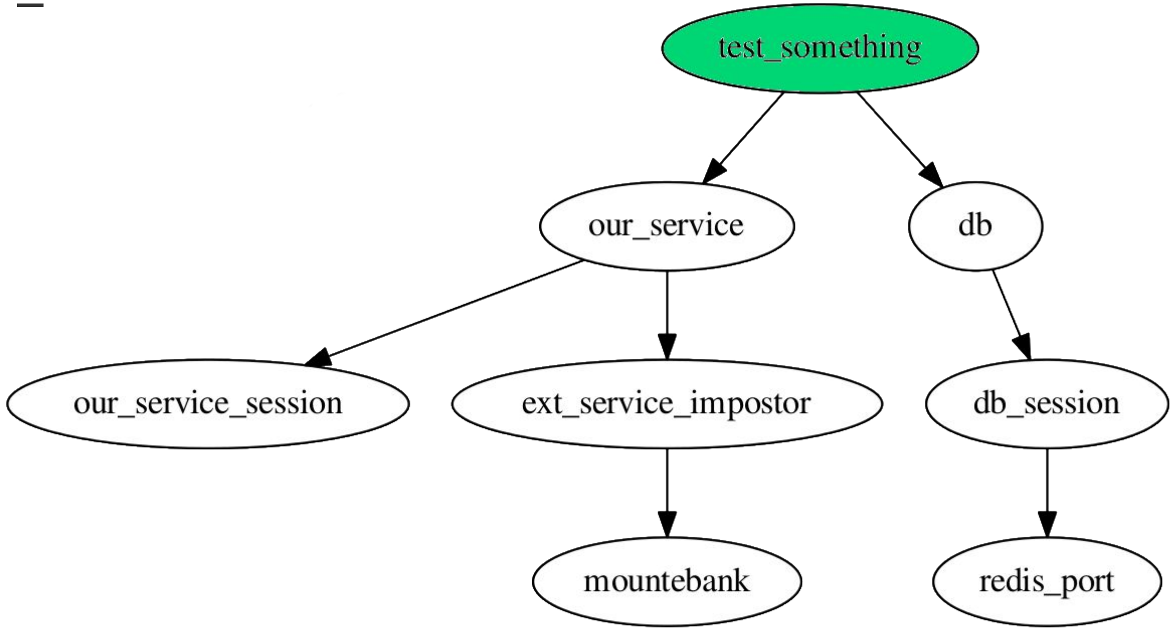
- A rewrite of an old, problematic (Java) service.
- My guinea pig.
- TDD helped a lot.
- ...not perfect, but educating

https://github.com/butla/pydas

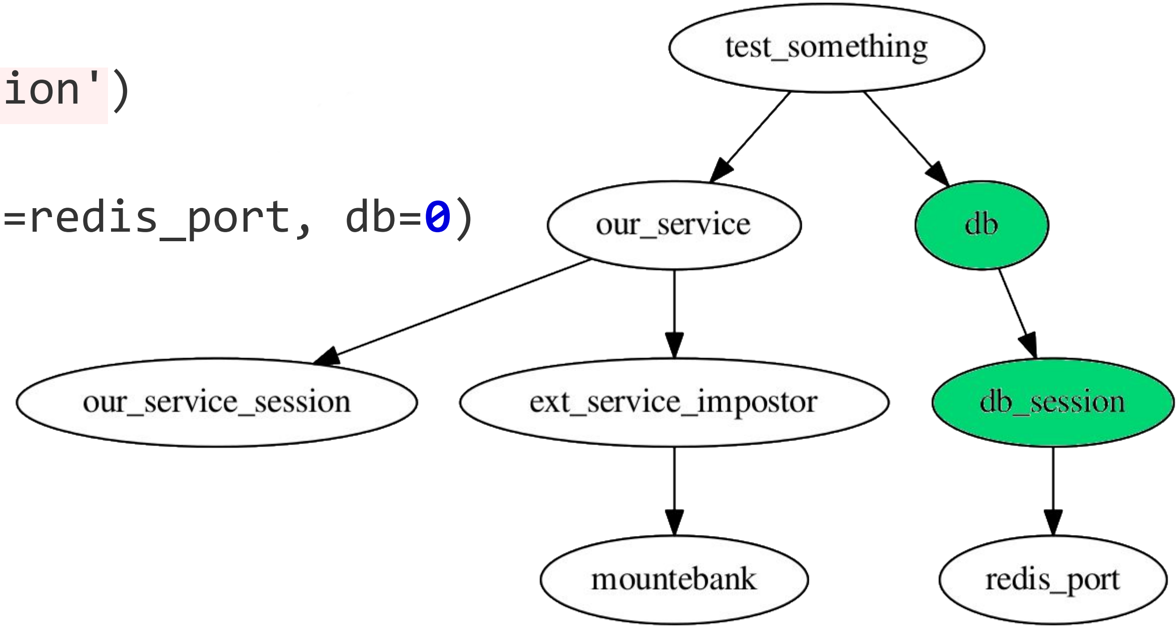
Pytest

- Concise
- Clear composition of fixtures
- Control of this composition (e.g. for reducing test duration)
- Helpful error reports

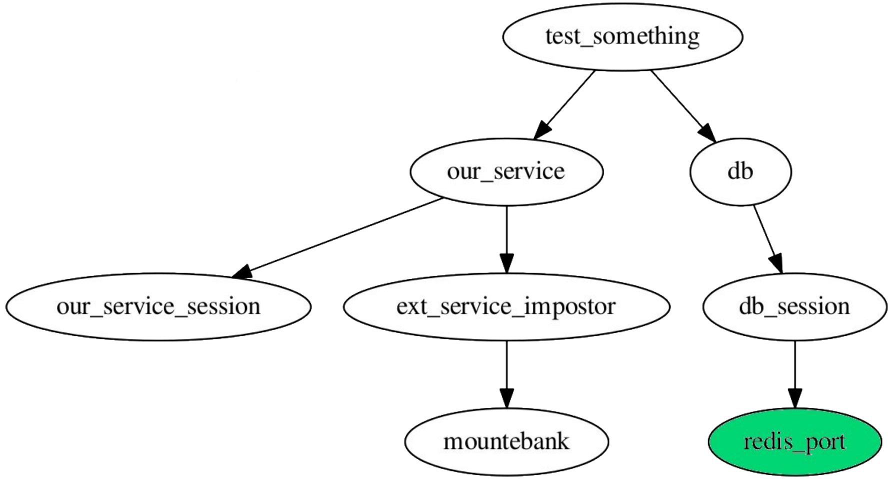
```python
def test_something(our_service, db):
    db.put(TEST_DB_ENTRY)
    response = requests.get(
        our_service.url + '/something',
        headers={'Authorization': TEST_AUTH_HEADER})
    assert response.status_code == 200
```



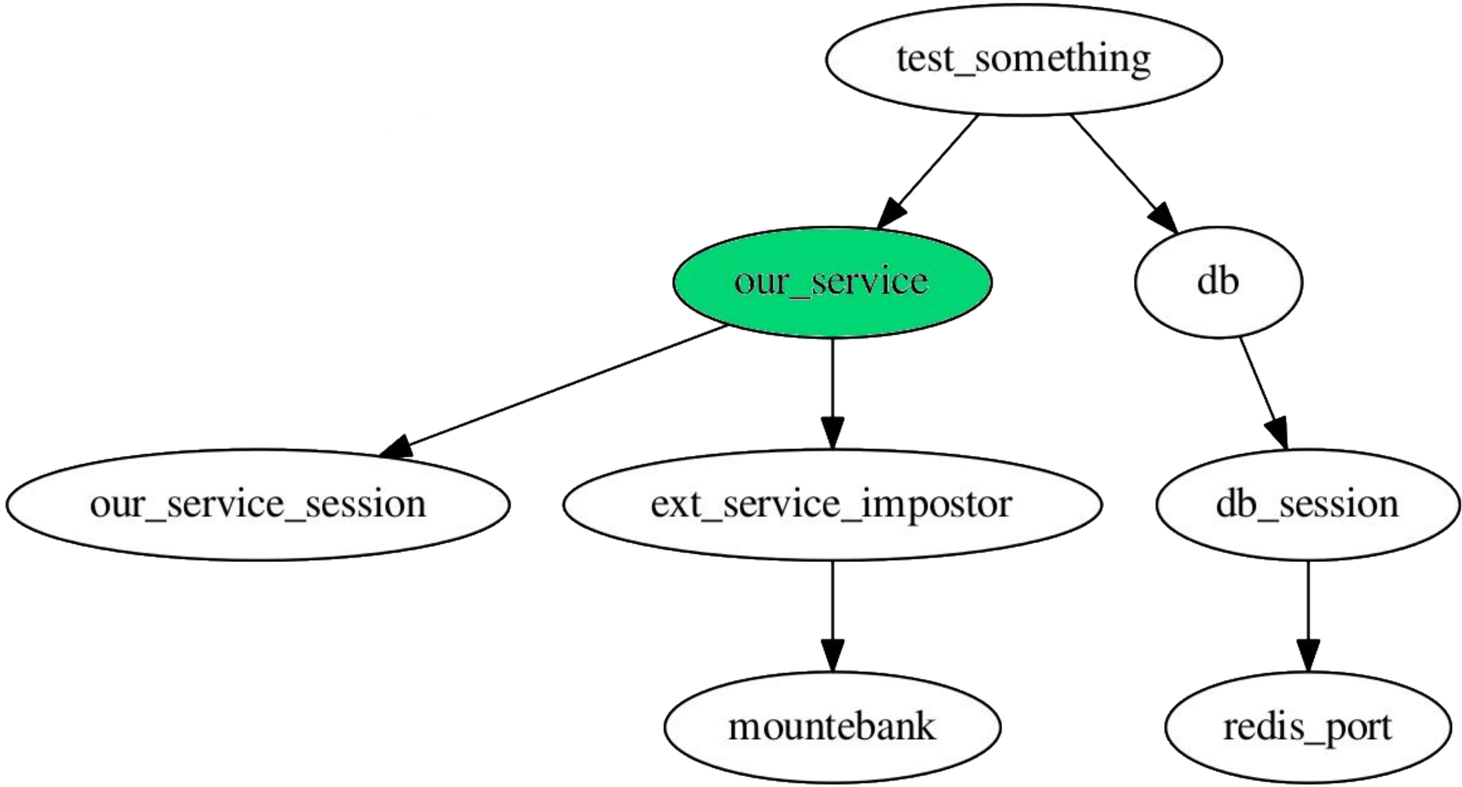
```python
import pytest, redis

@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)
```



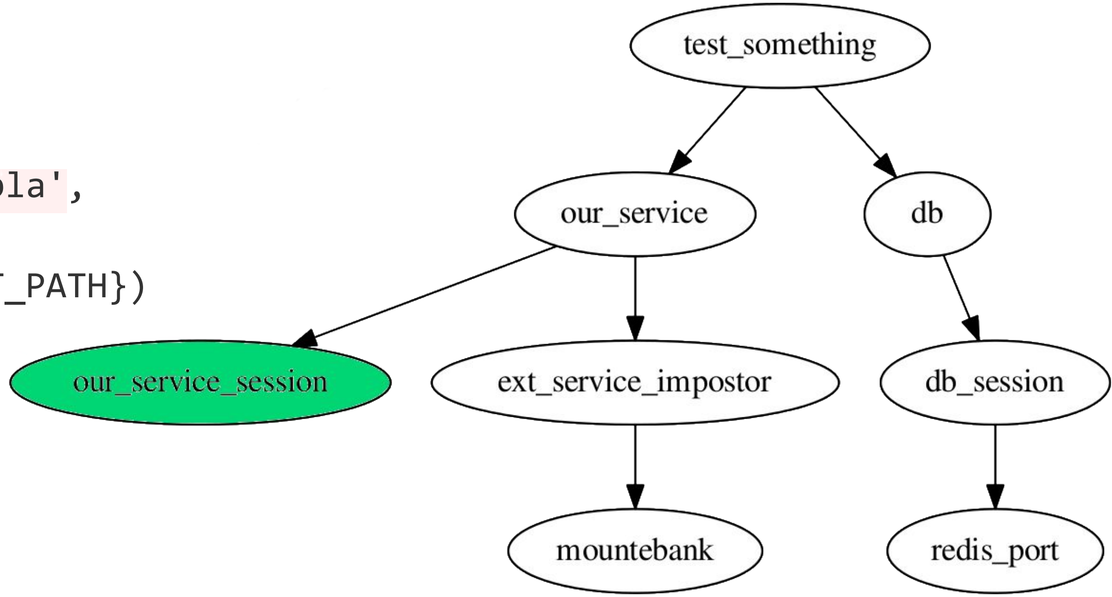
```python
@pytest.fixture(scope='function')
def our_service(our_service_session, ext_service_impostor):
    return our_service
```


$$\begin{tikzcd}
 & & \text{test\_something} \arrow[dl] \arrow[dr] & \\
 & \text{our\_service} \arrow[dl] \arrow[d] & & \text{db} \arrow[d] \\
\text{our\_service\_session} & \text{ext\_service\_impostor} \arrow[d] & & \text{db\_session} \arrow[d] \\
 & \text{mountebank} & & \text{redis\_port}
\end{tikzcd}$$

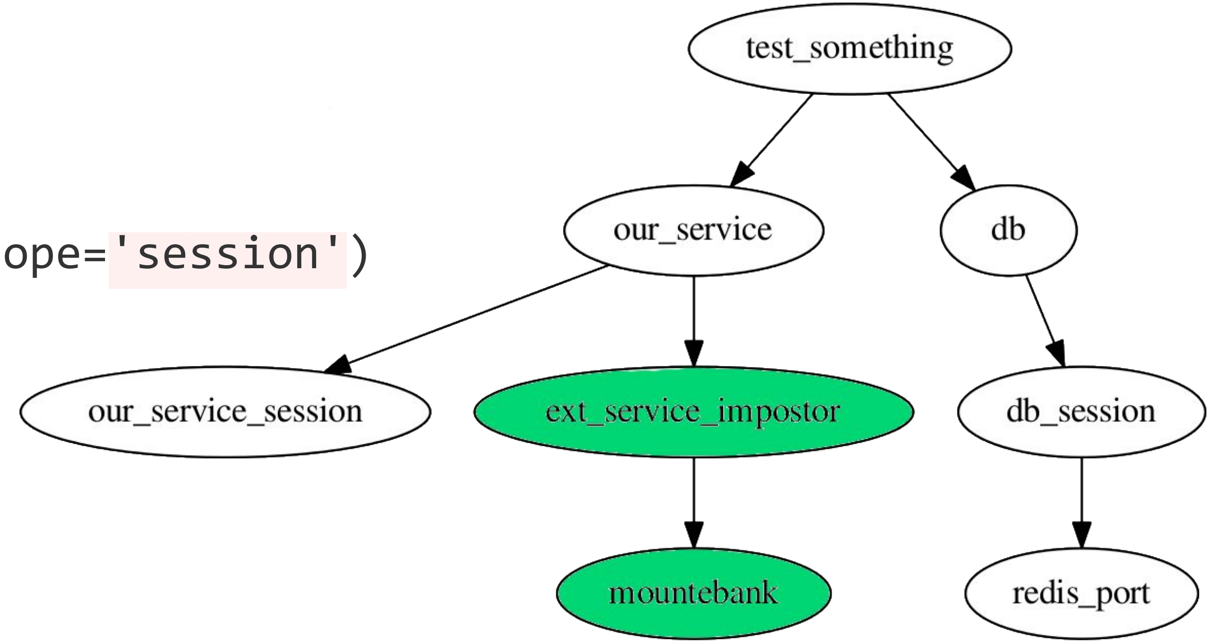
Mountepy

- Manages a Mountebank instance.
- Manages service processes.
- https://github.com/butla/mountepy
- `$ pip install mountepy`

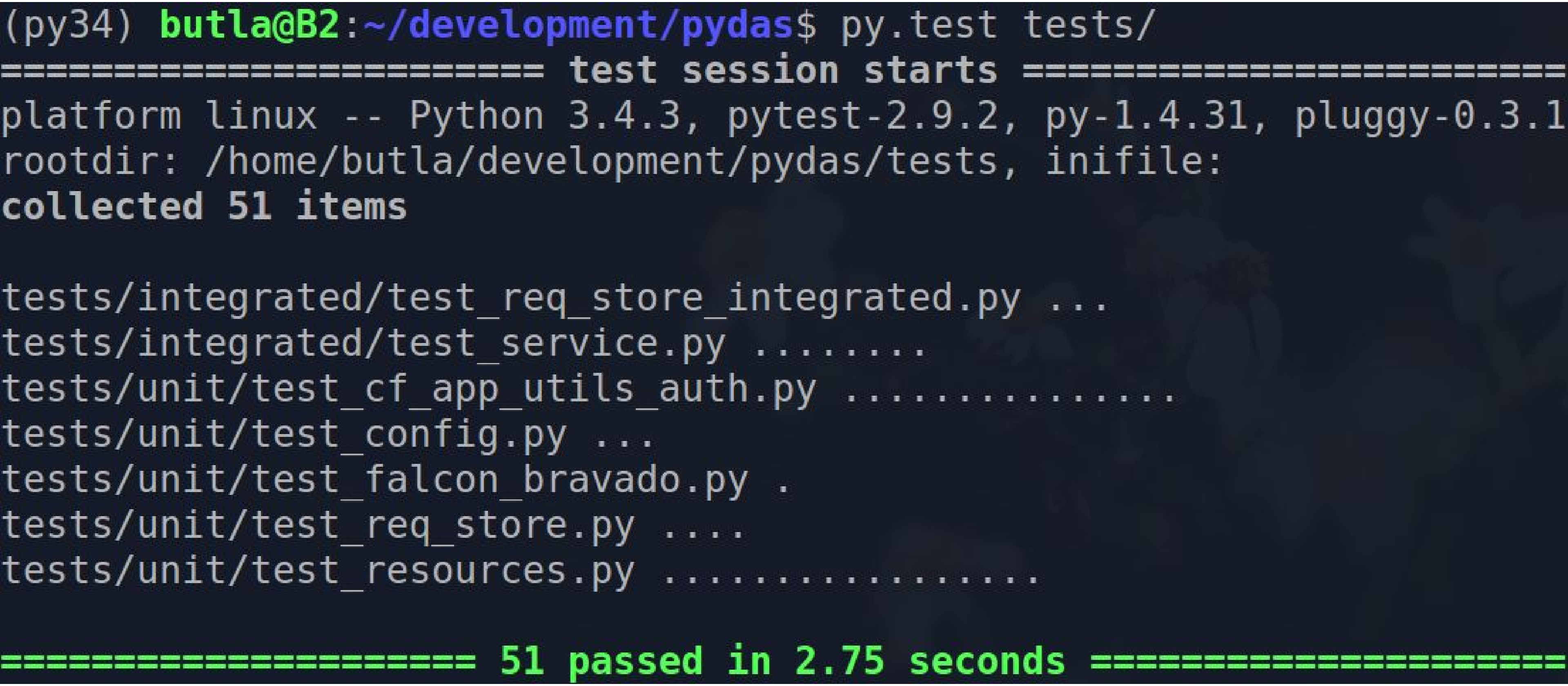
```python
import mountepy

@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()
```


$$\begin{tikzcd}
 & & \text{test\_something} \arrow[dl] \arrow[dr] & \\
 & \text{our\_service} \arrow[dl] \arrow[d] & & \text{db} \arrow[d] \\
\text{our\_service\_session} & \text{ext\_service\_impostor} \arrow[d] & & \text{db\_session} \arrow[d] \\
 & \text{mountebank} & & \text{redis\_port}
\end{tikzcd}$$

```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()
```



Service test(s) ready!



```
(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 =====================
```

Remarks about service tests

- Will yield big error logs.
- Breaking a fixture yields crazy logs.
- Won’t save us from stupidity (e.g. hardcoded localhost)

The danger of “other people’s commits”

Our weapons

- Test coverage
- Static analysis (pylint, pyflakes, etc.)
- Contract tests

.coveragerc (from PyDAS)

```
[report]

fail_under = 100

[run]

source = data_acquisition

parallel = true
```

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

Static analysis

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

Contract tests:
a leash for the interface

```yaml
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/

Contract is separate from the code!

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

- Creates service client from Swagger
- Verifies
  - Parameters
  - Returned values
  - Returned HTTP codes
- Configurable (doesn’t have to be as strict)

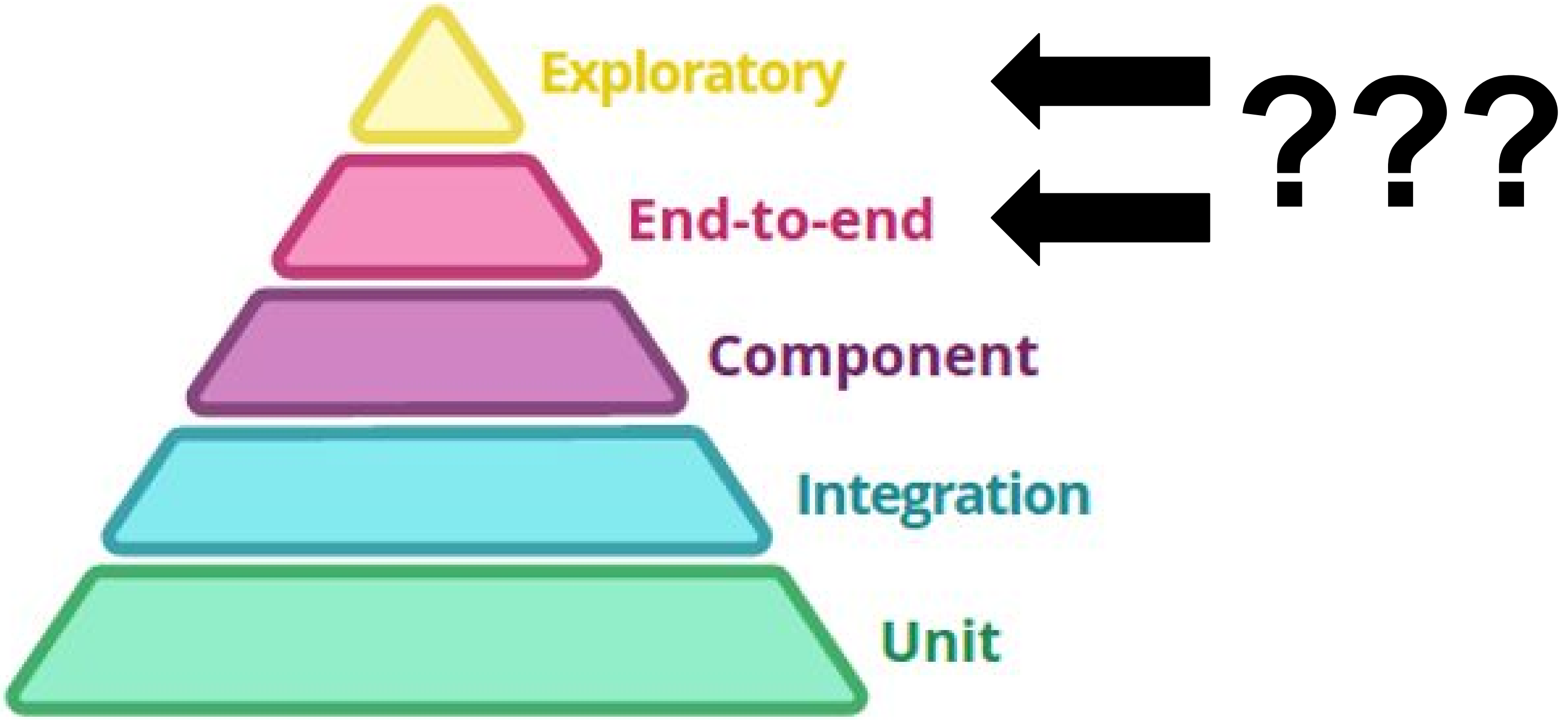
Bravado usage

- In service tests: instead of “requests”
- In unit tests:
  - https://github.com/butla/bravado-falcon
- Now they all double as contract tests

```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)
```

```python
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'
```

“Our stuff” is taken care of...


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

More about tests / microservices / stuff

“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)