Advanced Software Engineering (LAB)

Stefano Forti

name.surname@di.unipi.it

Department of Computer Science, University of Pisa


Q & A

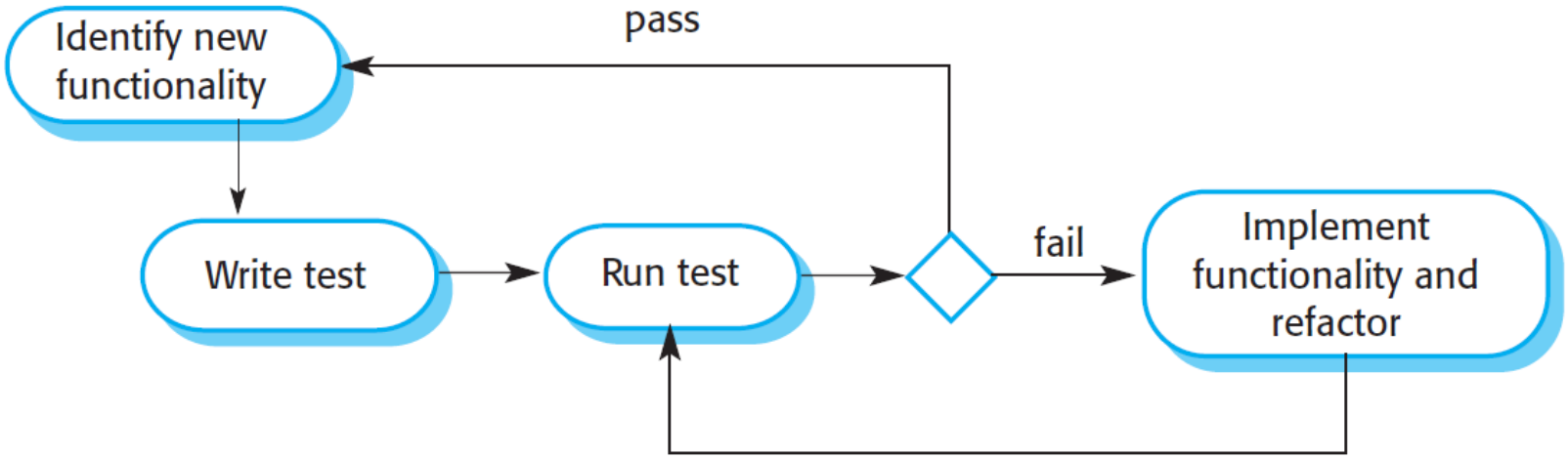
Agenda

- Overview
- Different type of tests:
  - Unit Tests
  - Component Tests (aka Functional ~~& Integration~~)
  - Performance Tests and Profiling
- Automate testing
  - Travis-CI & Coveralls

Test Driven Development

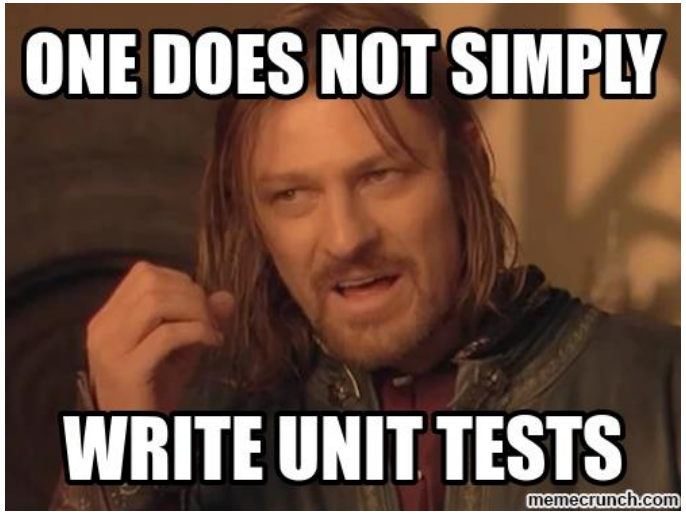
- We've seen testing, starting from Lab 1.
- TDD will not surely improve code quality, however it will make teams more agile: whenever you break a feature, you know it.

Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.



Writing tests

- It is **time-consuming** and can end up in tests that take **too long to run**.
- It is the best approach to **make a project grow at less expenses**.
- As usual :
  $$programmer(p) \wedge writesbadcode(p) \Longrightarrow writesbadtests(p)$$
- Writing tests lead to **new insights** on your project, API, code.



Testing micro-services

- **Unit tests**: Make sure a class or a function works as expected in isolation
- **Component tests**: Verify that the microservice does what it says, and behaves correctly even on bad requests (*functional*). Verify how a microservice integrates with all its network dependencies (*integration*).
- **Performance tests**: Measure the microservice performances against workload
- **System tests**: Verify that the whole system works with an end-to-end test.

Unit tests (Lab 1)

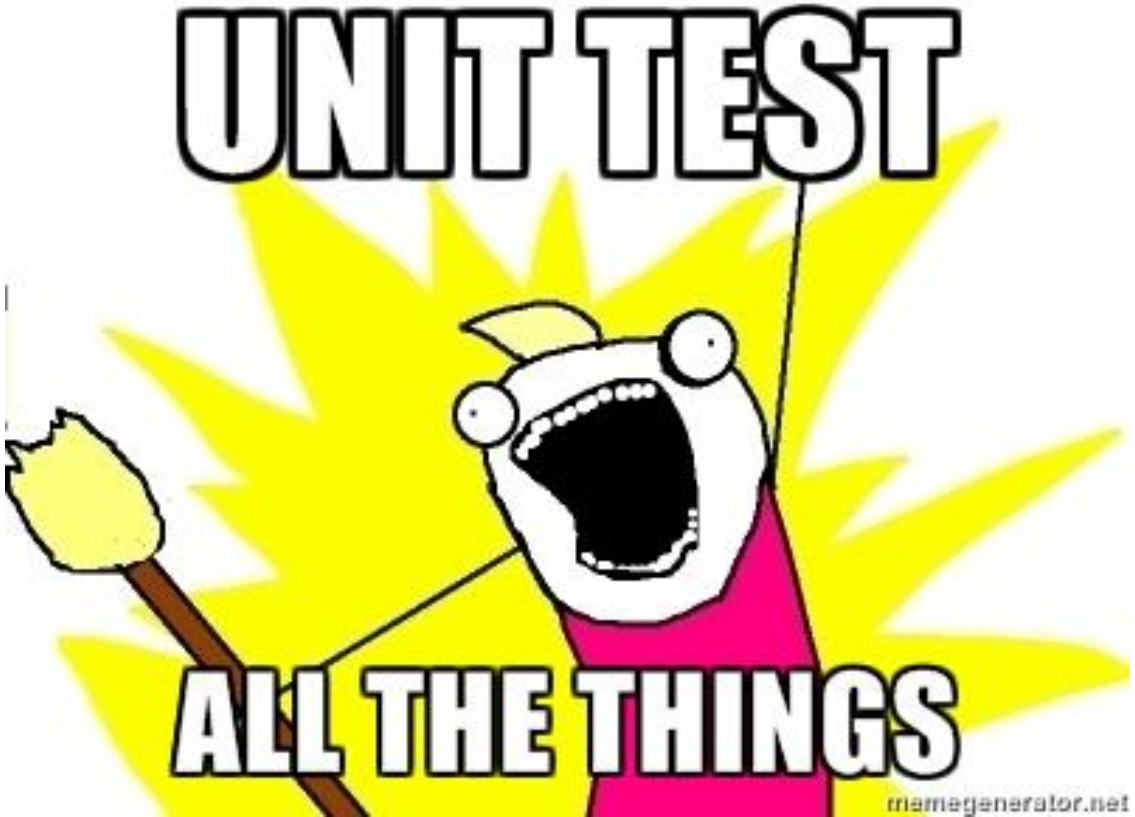
- In Flask projects, there usually are, alongside the views, some functions and classes, which can be **unit-tested in isolation**.
- In Python, calls to a class are *mocked* to achieve isolation.
- *Do you recall Lab 1?*

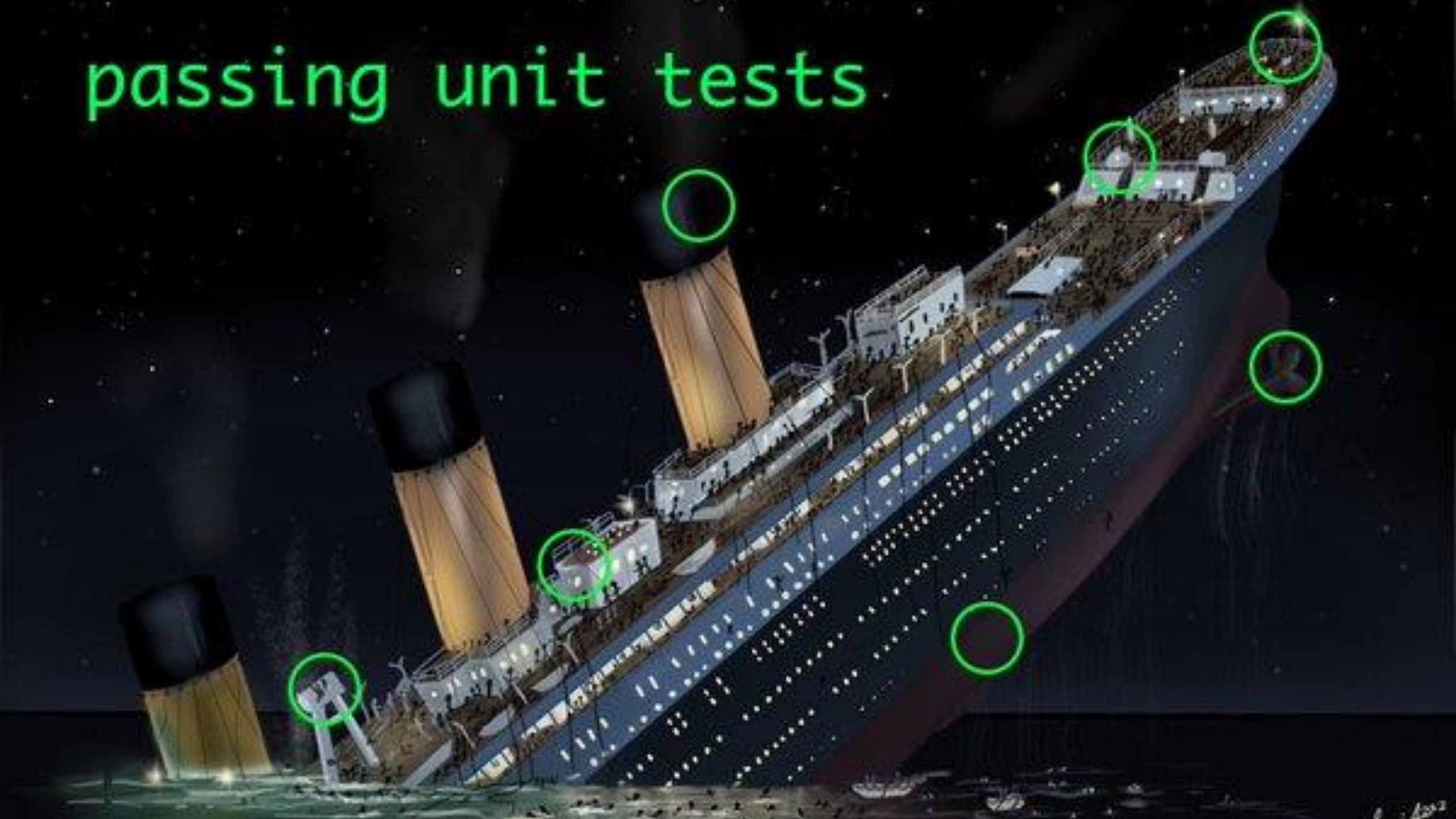
Pattern: Instantiate a class or call a function and verify that you get the expected results.

```python
import calculator as c
import unittest

class TestDivide(unittest.TestCase):

    def test_divide_integers_positive(self):
        result = c.divide(6, 3)
        self.assertEqual(result, 2)

    def test_divide_integers_positive2(self):
        result = c.divide(7, 3)
        self.assertEqual(result, 2)

    def test_divide_integers_negative(self):
        result = c.divide(-6, -2)
        self.assertEqual(result, 3)

    def test_divide_integers_negative2(self):
        result = c.divide(-7, -2)
        self.assertEqual(result, 3)

    def test_divide_integers_pos_neg(self):
        result = c.divide(6, -2)
        self.assertEqual(result, -3)

    def test_divide_integers_pos_neg2(self):
        result = c.divide(9, -2)
        self.assertEqual(result, -4)

    def test_divide_integers_neg_pos(self):
        result = c.divide(-6, 2)
        self.assertEqual(result, -3)

    def test_divide_integers_neg_pos2(self):
        result = c.divide(-7, 2)
        self.assertEqual(result, -3)

    def test_divide_zero(self):
        result = c.divide(0, 2)
        self.assertEqual(result, 0)

    def test_divide_by_zero(self):
        self.assertRaises(ZeroDivisionError, c.divide, 6, 0)


if __name__ == '__main__':
    unittest.main()
```

In-Class Work (10 mins)

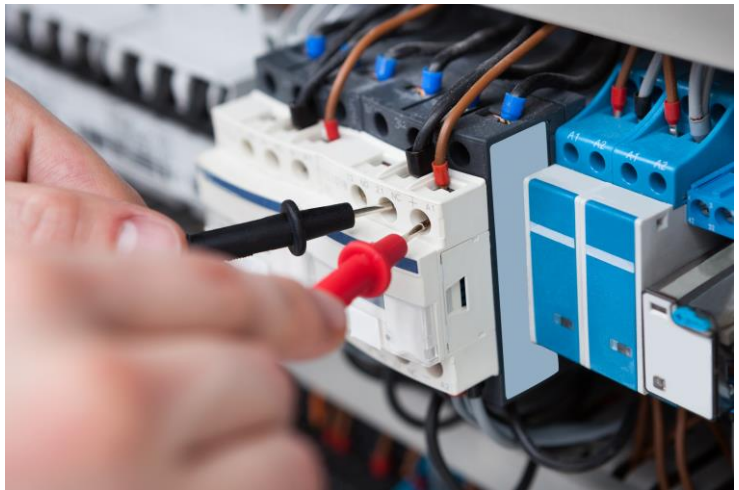
- By relying on `myservice/classes/poll.py` in **Homework 1** write 1-3 unit test cases for the methods of the class `Poll`.




passing unit tests

Component tests

- Functional tests for a microservice project are all the tests that interact with the **published API** by sending HTTP requests and asserting the HTTP responses.
- Important to test:
  - **that the application does what it is built for,**
  - **that a defect that was fixed is not happening anymore**.



Pattern: Create an instance of the component in a test class and interact with it by mock (or actual) network calls.

Where did we see this?

HW1

In-Class Work (20 mins)

- By relying on your **Homework 1** solution code (or your neighbour's code), add `test4()` to `test_doodle.py` that checks that

the _POLLNUMBER global variable never decreases

```
Why this invariant?


How to test this?
```

Load Test

- The goal of a load test is to understand your service's bottlenecks under stress.
- Understanding your system limits will help you determining how you want to deploy it and if its design is future-proof in case the load increases.
- Shoot at it!

Pattern: Create an instance of the component and stress test it by mocking different amount of workload.

Boom!

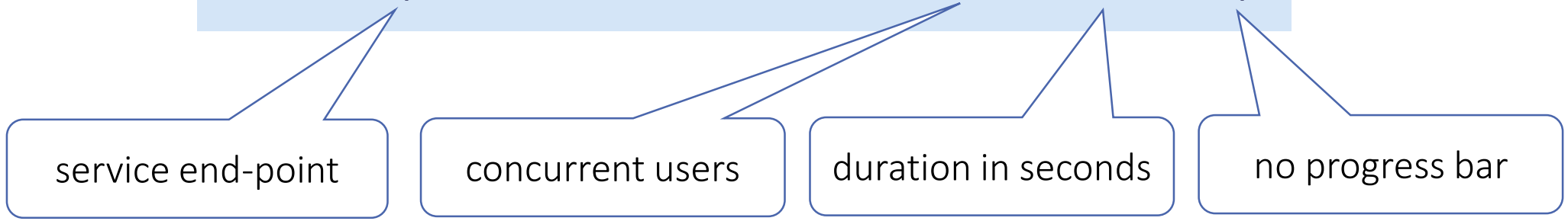
https://pypi.org/project/boom/

- Boom! is a script you can use to quickly smoke-test your web app deployment. First:

```
pip install boom
```

- Then, `flask run` the doodles microservice and boom it:

```
boom http://127.0.0.1:5000/doodles -c 100 -d 10 -q
```



Results

```
Server Software: Werkzeug/0.14.1 Python/3.6.4
Running GET http://127.0.0.1:5000/doodles
Running for 10 seconds - concurrency 100.
Starting the load..........................
..........................................
..........................................
..........................................
..........................................
..........................................
..........................................
..........................................
..........................................
..........................................
..........................................

-------- Results --------
Successful calls                2133
Total time                      10.0228 s
Average                         0.4393 s
Fastest                         0.1984 s
Slowest                         0.5655 s
Amplitude                       0.3670 s
Standard deviation              0.045126
RPS                             212
BSI                             Pretty good

-------- Status codes --------
Code 200                        2133 times.
```

- Another similar tool is Molotov.

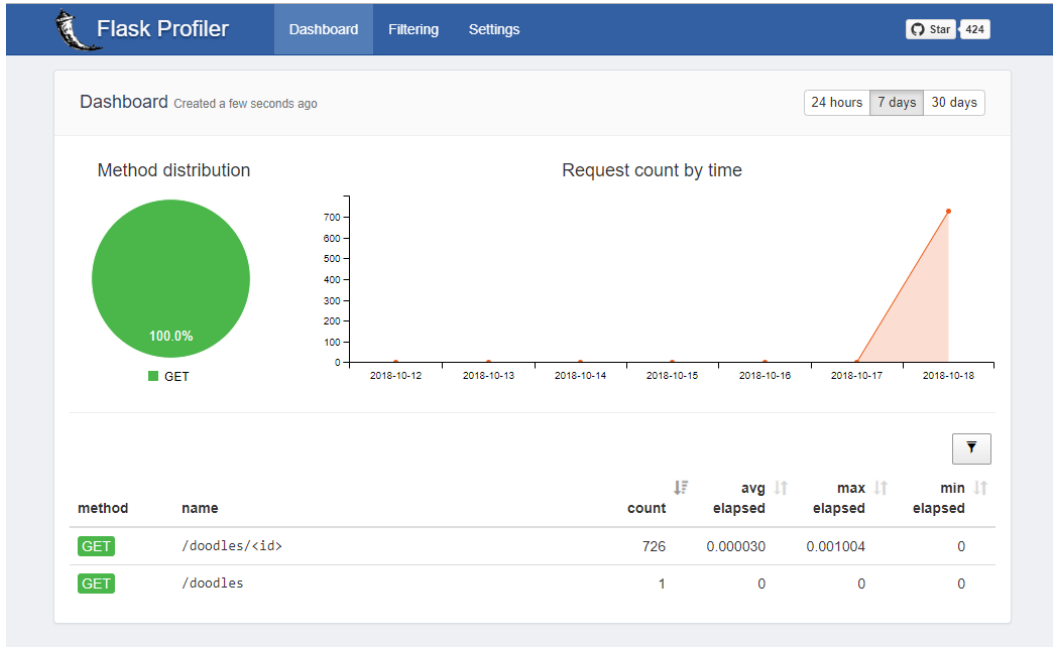
https://molotov.readthedocs.io/en/stable/

- Both tools give an idea of how a microservice behaves in case of stress, but…

flask-profiler

…it's better to add some metrics on the server side!

```
pip install flask-profiler
```



It gives answers to these questions:

- Where are the bottlenecks in my application?
- Which endpoints are the slowest in my application?
- Which are the most frequently called endpoints?
- What causes my slow endpoints?
- How much time did a specific request take?

App with Profiler

- Modify the app.py file of the Doodle project, by changing the app.py file with the new one beside (on Moodle).
- Run

```
python .\myservice\app.py
```

- Browse

  http://127.0.0.1:5000/flask-profiler

- Boom it again! :D

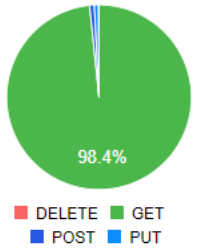
```python
#App with Profiler
import os
from flakon import create_app
from myservice.views import blueprints
from flask_profiler import Profiler

_HERE = os.path.dirname(__file__)
_SETTINGS = os.path.join(_HERE, 'settings.ini')

app = create_app(blueprints=blueprints, settings=_SETTINGS)

app.config["DEBUG"] = True

app.config["flask_profiler"] = {
    "enabled": app.config["DEBUG"],
    "storage": {
        "engine": "sqlite"
    },
    "basicAuth":{
        "enabled": True,
        "username": "admin",
        "password": "admin"
    },
    "ignore": [
        "^/static/.*"
    ]
}

profiler = Profiler(app)

if __name__ == '__main__':
    app.run(host="127.0.0.1", port=5000)
```

Let’s profile it!

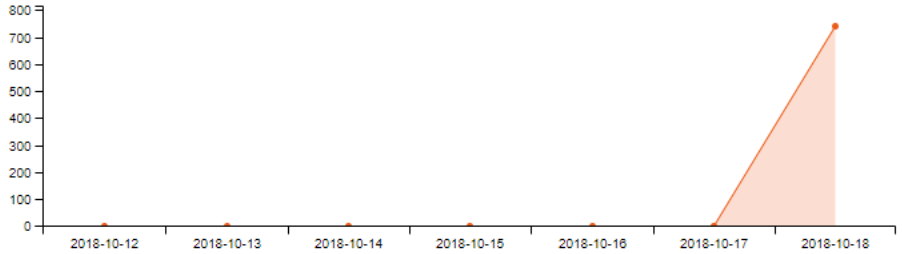
Flask Profiler | Dashboard | Filtering | Settings | Star 424

Dashboard Created a few seconds ago — 24 hours | 7 days | 30 days

Method distribution



Request count by time



method	name	count	avg elapsed	max elapsed	min elapsed
GET	/doodles/<id>	727	0.000030	0.001004	0
POST	/doodles	6	0.000332	0.000998	0
PUT	/doodles/<id>	5	0.001795	0.008975	0
DELETE	/doodles/<id>	1	0	0	0
GET	/doodles	1	0	0	0

Calculator Exercise (you can do if you have time)

1. Include the `calc` blueprint in the doodles microservice.
2. Design and implement component tests.
3. Boom and profile it!