Advanced Software Engineering (LAB)

Stefano Forti

name.surname@di.unipi.it

Department of Computer Science, University of Pisa


Q & A

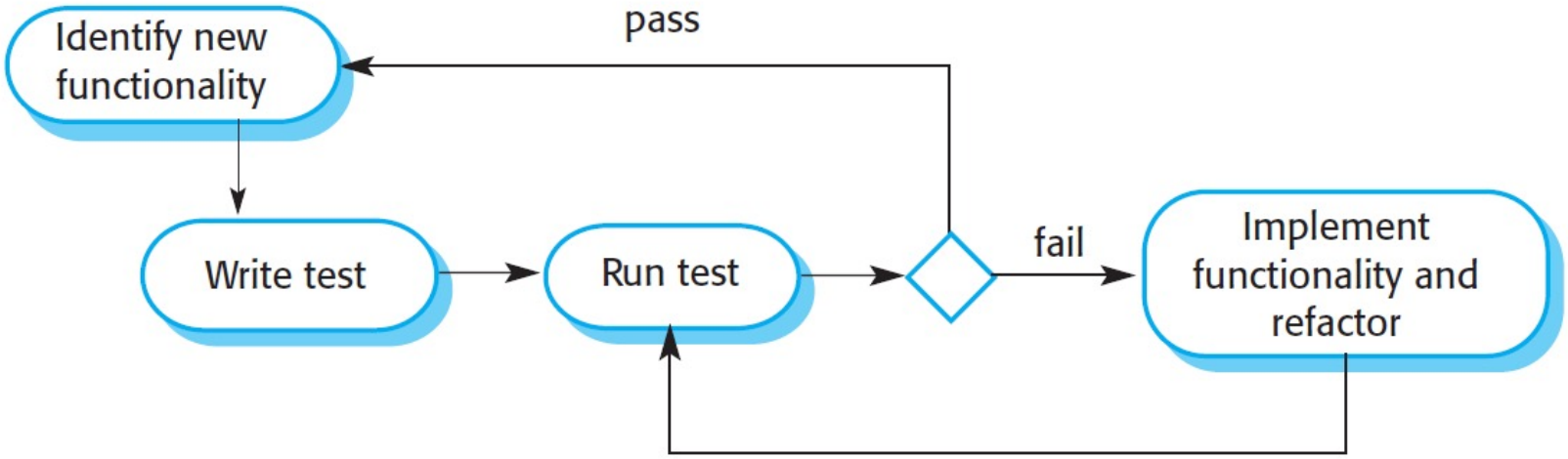
Agenda

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

Test Driven Development

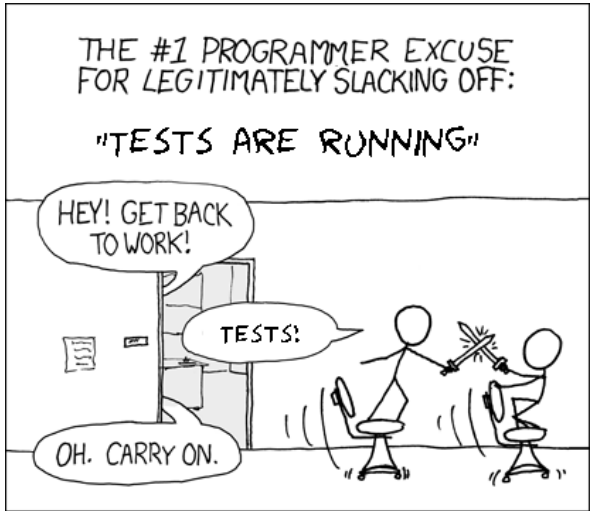
- We've seen testing, starting from Lab 1.
- TDD will not surely improve code quality, however it will make teams more agile: whenever you break a feature, you know it.

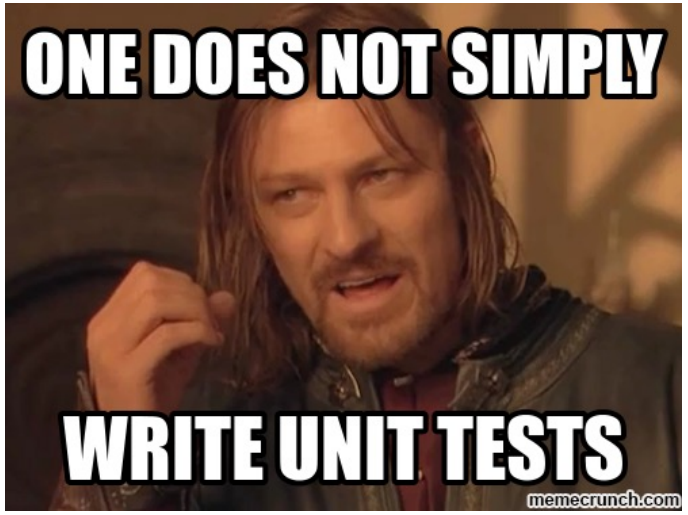
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.



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)

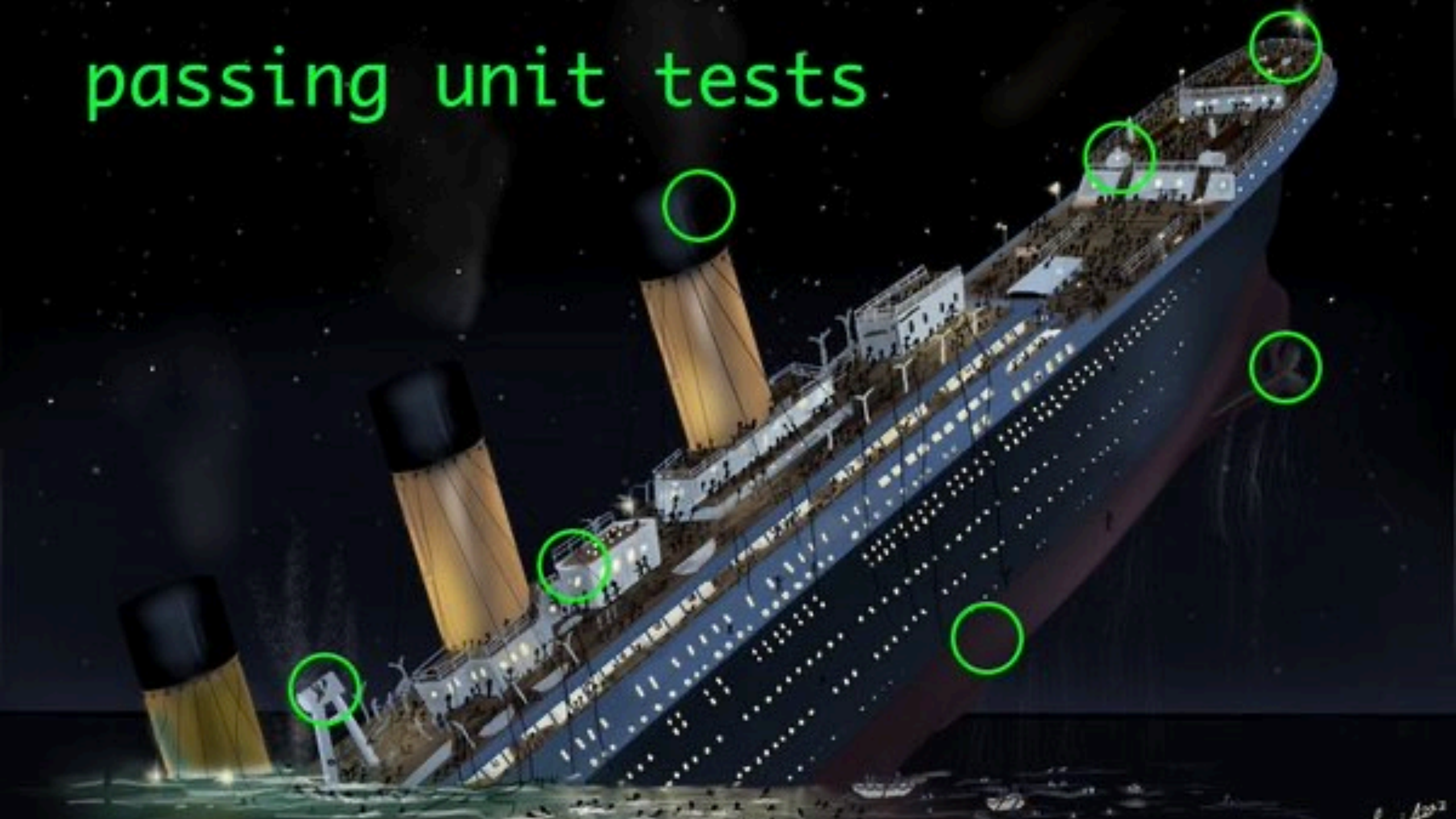
- In Flask projects, there usually are, alongside the views, some functions and classes, which can be **unit-tested in isolation**.
- In Python, calls to a class are *mocked* to achieve isolation.

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()
```

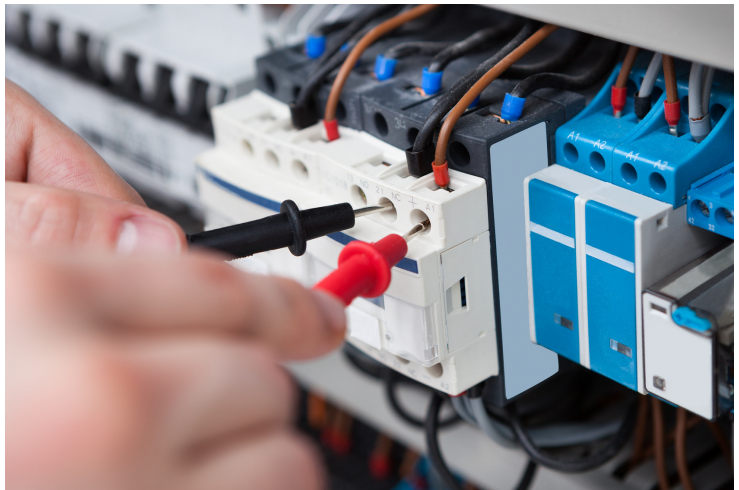
Where did we see this?

LAB 1


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.

Calculator SCALE-UP [40 mins]

1. Download the `calc` microservice from the Moodle. In the tests folder you find:
   - Unit tests for the calculator module
     - test_calculator.py
   - Component tests for the home and calc view:
     - test_home.py
     - test_calc.py

2. Extend unit and component tests, trying to increase coverage by 5% at least (**and to spot bugs!**).

3. Try Locust.

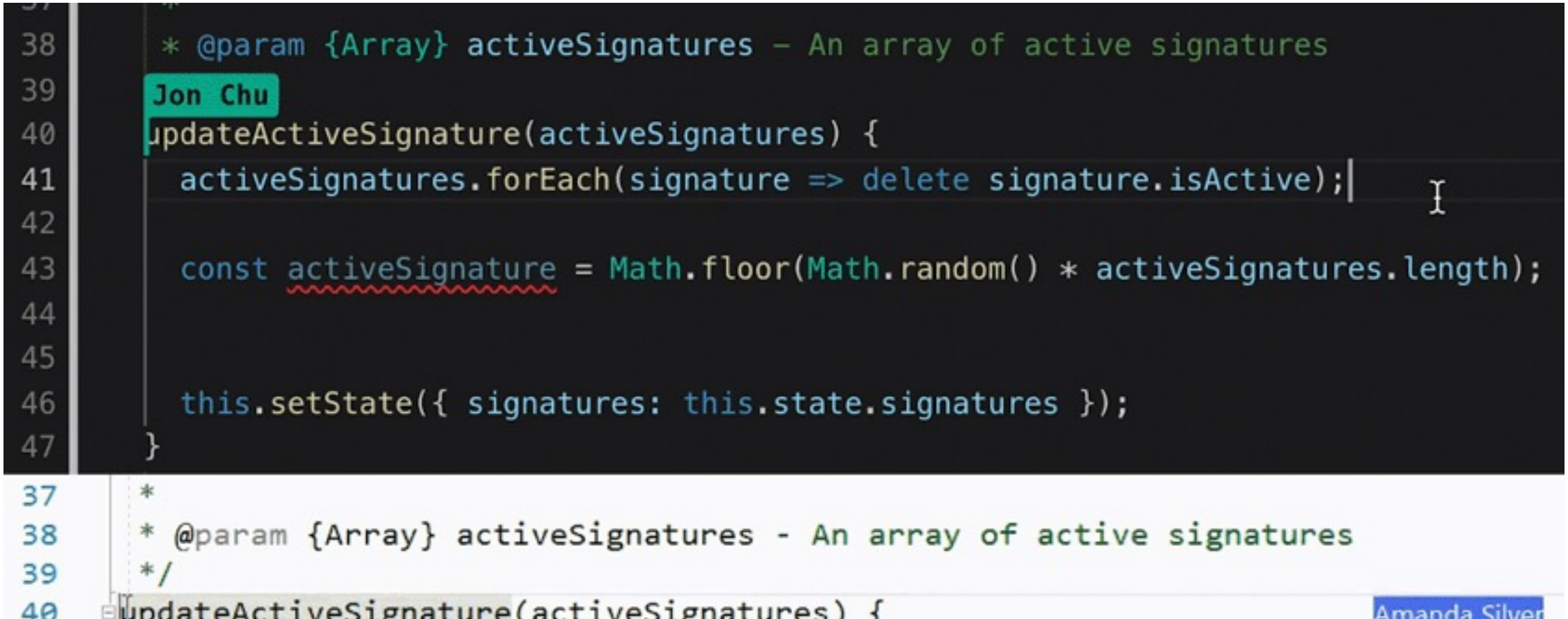
4. Build a pipeline for TravisCI with coveralls.

Submit the GitHub link to your repo by the end of the class! ☺

VS Code LiveShare

https://visualstudio.microsoft.com/it/services/live-share/

- To ease cooperation within the group, you might try this tool.



pytest

- As we've seen with the homework, **`pytest`** launches all **`test*`** files inside the **`tests`** folder. It performs test discovery.
- For our "skeleton", we might need to use: `python -m pytest`
- A useful extension to evaluate test coverage is:

```
pip install pytest-cov
```

```
pytest --cov=calc tests/
```

```
---------- coverage: platform win32, python 3.7.0-final-0 -----------
Name                                Stmts   Miss  Cover
-------------------------------------------------------
myservice\__init__.py                   1      0   100%
myservice\app.py                       12      1    92%
myservice\classes\__init__.py           0      0   100%
myservice\classes\poll.py              47      2    96%
myservice\tests\__init__.py             0      0   100%
myservice\tests\_test_int.py           32     32     0%
myservice\tests\test_home.py          112      0   100%
myservice\tests\test_poll.py           12      1    92%
myservice\views\__init__.py             2      0   100%
myservice\views\doodles.py             59      0   100%
-------------------------------------------------------
TOTAL                                 277     36    87%
```

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.

What are these?

- An open source load testing tool use by Big Companies.
- 6 steps:

```
pip install locust
```

- Create `locustfile.py` in your root project folder to define users behaviours. (See next slide)

- Start your microservice.
- Open a new terminal and issue:

```
locust
```

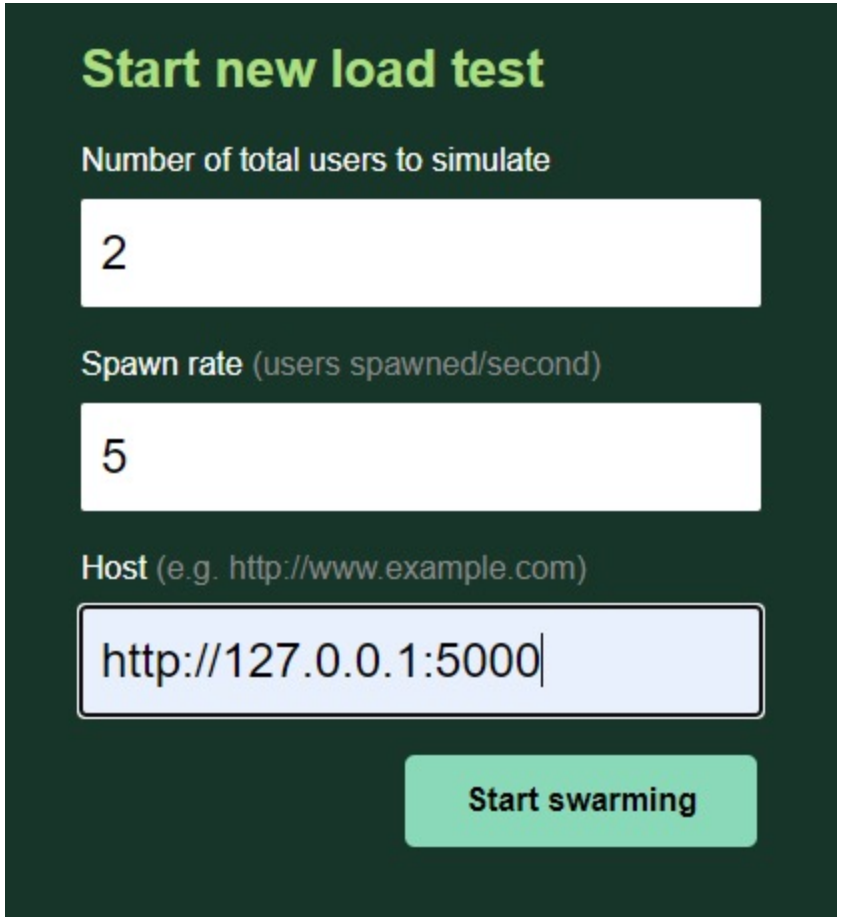
- Browse to http://localhost:8089
- **Set up and run your tests!**

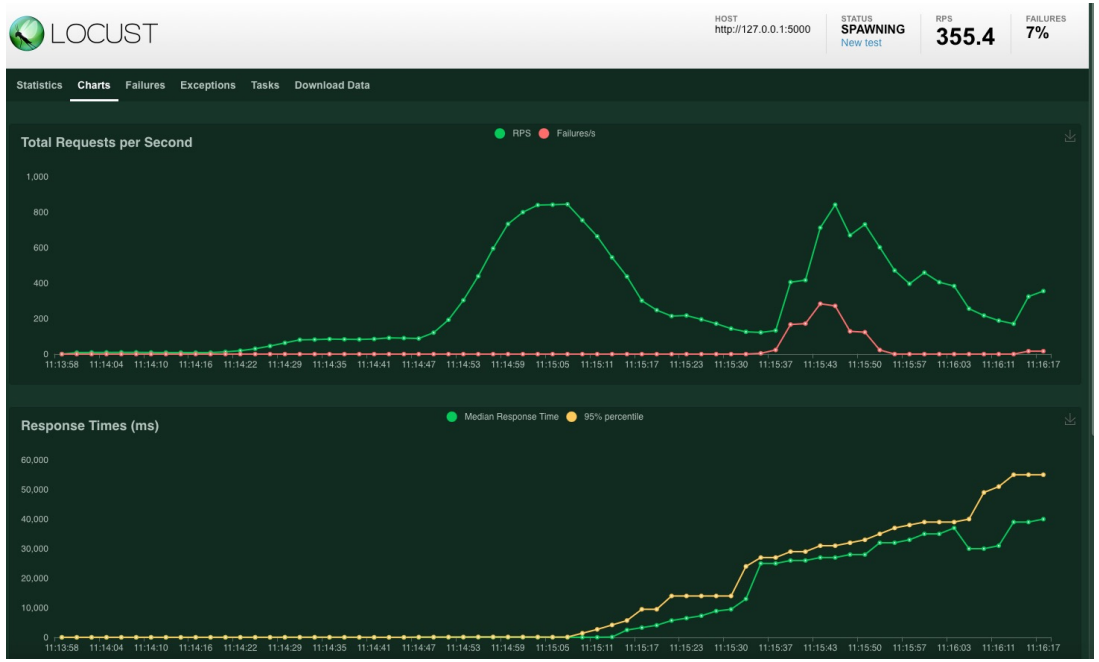
locustfile.py

```python
import time
from locust import HttpUser, task, between

class QuickstartUser(HttpUser):
    wait_time = between(1, 2) # 1-2 seconds between simulated events

    @task # defines a user task, asssociated to a microthread by locust
    def index_page(self):
        self.client.get("/") # get home page

    @task(3) # this task is 3 times likelier than the previous!
    def view_item(self):
        for item_id in range(10):
            self.client.get(f"/calc/sum?m={item_id}&n={42}", name="/calc/sum")
            time.sleep(1)

    def on_start(self): # init for each virtual user
         pass
```

Stress your microservice!

http://localhost:8089





tox

```
pip install tox
```

- Tox is used to run tests on different versions of Python in isolated environment.
- It requires a **`tox.ini`** file in the root folder of the project, the requirements for the test environment and the commands to be run.
- It runs with **`tox`**.

```
[tox]
envlist = py39,flake8
skipsdist=True

[testenv]
passenv = TRAVIS TRAVIS_JOB_ID TRAVIS_BRANCH
deps = pytest
       pytest-cov
       coveralls
       -r requirements.txt

commands =
    pytest --cov-config .coveragerc --cov calc calc/tests
    - coveralls


[testenv:flake8]
commands = flake8 calc
deps =
    flake8
```

```
_____________________________
py39: commands succeeded
flake8: commands succeeded
congratulations :)
```

Travis-CI

- Register with your GitHub to https://travis-ci.com
- Add and commit the `.travis.yml` file in your repo.
- Add the calc repo to your travis account and trigger a build of the repo.

```
language: python
python: 3.9
env:
      - TOX_ENV=py39
install:
      - pip install tox
script:
      - tox -e $TOX_ENV
```

Coveralls

- Register with your GitHub to https://coveralls.io/
- Add the calc repo.
- Run build on travis-ci ☺

ASE Calc Microservice

build passing

coverage 80%