Advanced Software Engineering (LAB)

Stefano Forti

name.surname@di.unipi.it

Department of Computer Science, University of Pisa

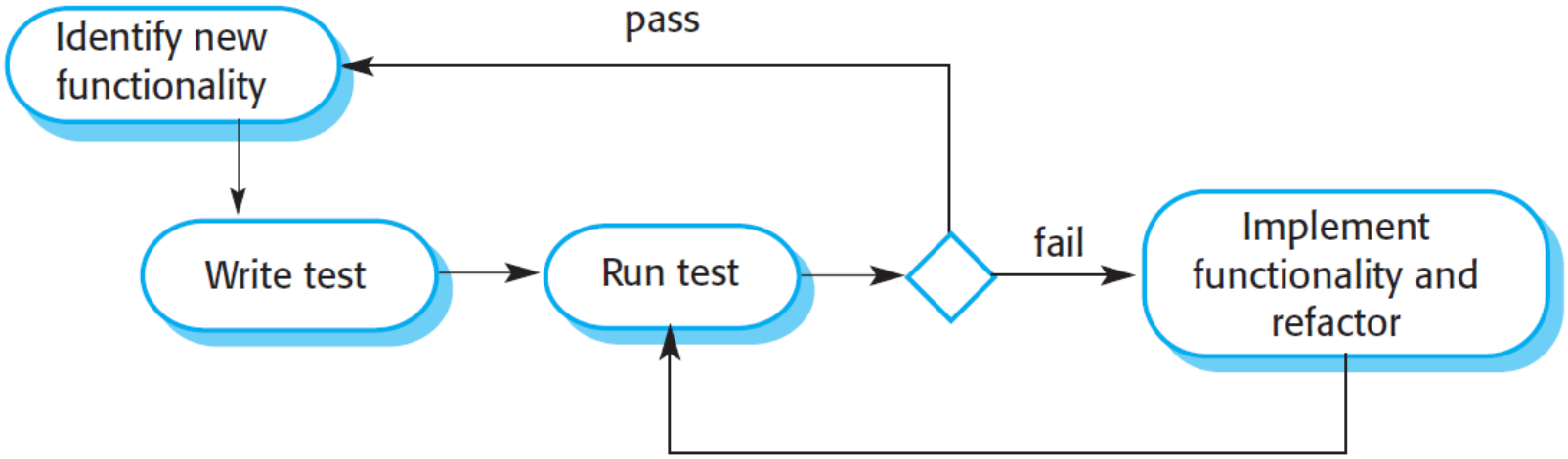
Agenda

- Overview
- Different type of tests:
  - Functional tests (`pytest`)
  - Performance Tests (`locust`)

Test Driven Development

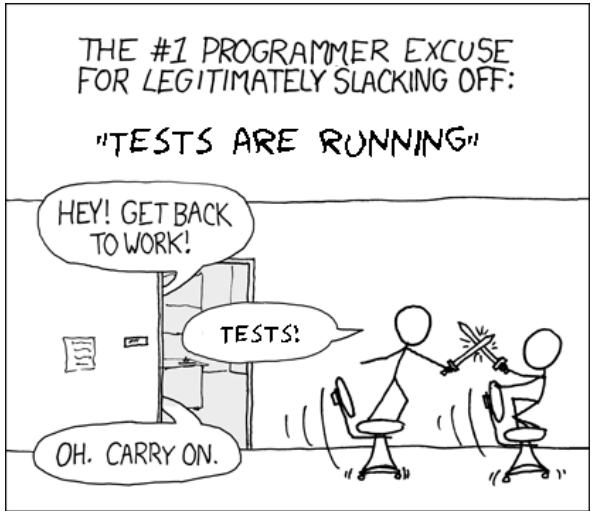
- TDD will not surely improve code quality, however it will make teams more agile: whenever you break a feature, you know it.

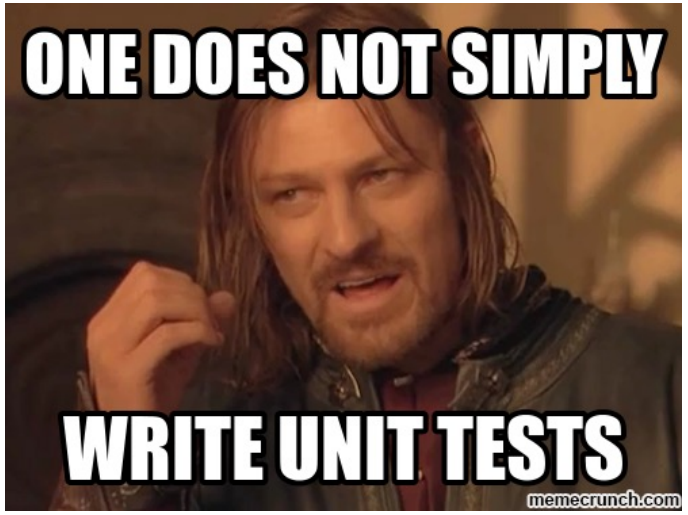
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.



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

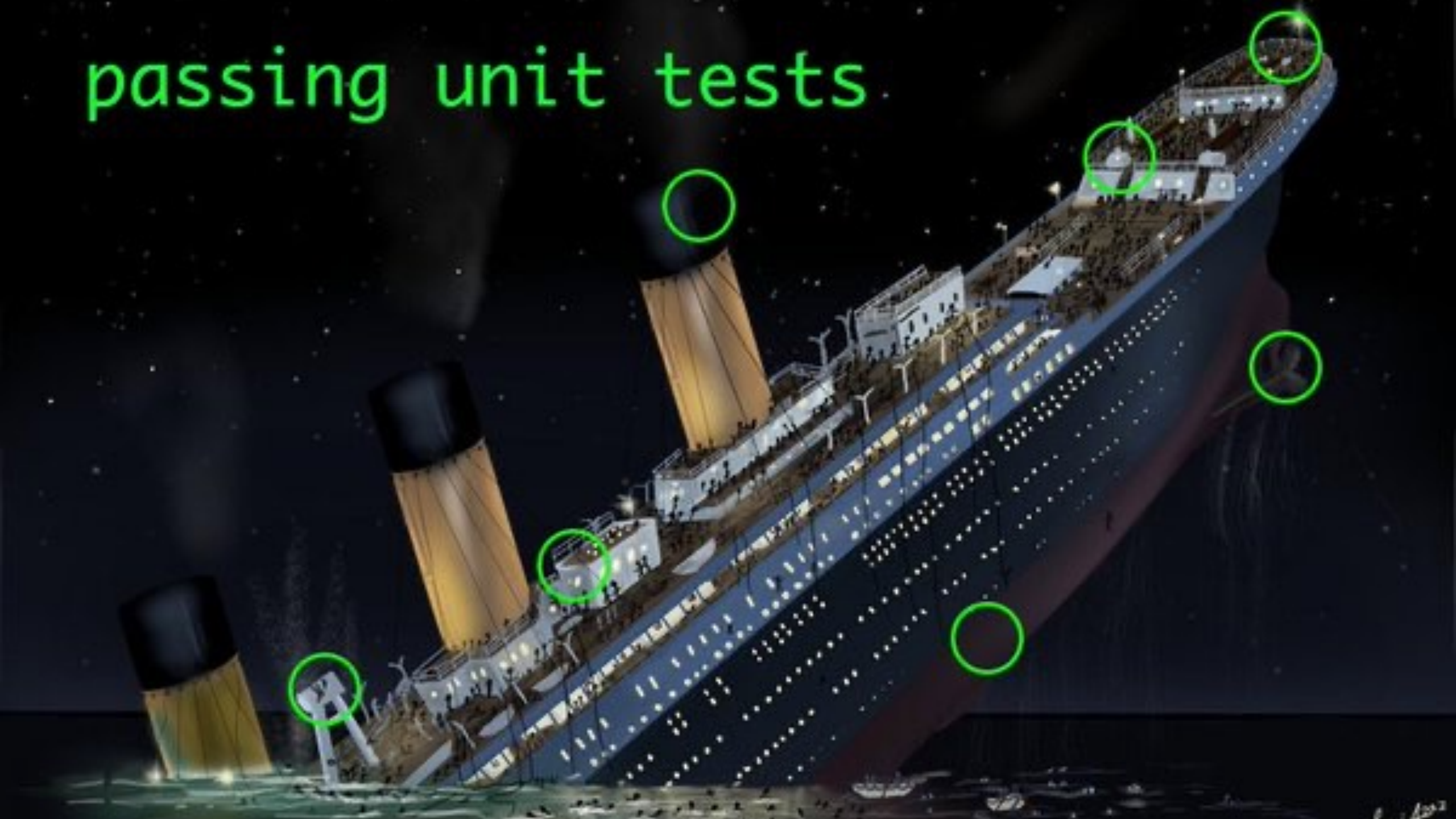
- **Functional testing**: Test the functionality of the whole system (unit -> feature -> system -> release testing)
- **User testing**: Test usability by end-users.
- **Performance testing**: Measure the microservice performances against varying workload

- **Security testing**: Still remember bandit and OWASP Zap?

Unit tests 101

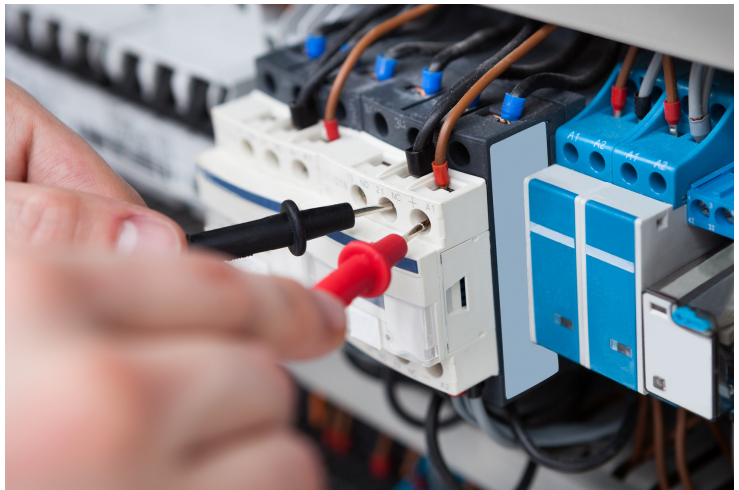
- In Flask projects, there usually are, alongside the views, some functions and classes, which can be **unit-tested in isolation**.
- In Python, calls to a class are *mocked* to achieve isolation.
- Examples of unit tests from the Jenkins lab: https://github.com/teto1992/tic-tac-toe/blob/master/test_game.py

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


passing unit tests

Unit tests (cont'd)

- 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.

Unit testing with `pytest`

```
pip install pytest
```

1. Download the **`test-lab.zip`** primer
2. Add files like **`a_test.py`** to unit test the **`math_py`** microservice.
3. Use tests to spot and fix bugs.

Note:

- **`conftest.py`** configures the mock microservice
- **`app.py`** has been modified to inhibit **`update_stats(service, op)`** in testing mode.

```python
def update_stats(service, op):
    if (not app.config['TESTING']): # ADD THIS LINE for TESTING PURPOSES
        # make post to stats service
        x = requests.post(f"http://stats-service:5000/stats/{service}/{op}")
        print(x, file=sys.stderr)
```

pytest 101

- `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=math_py
```

```
---------- 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 `microase`:

```
docker compose build
docker compose up
```

- Open a new terminal and issue:

```
locust
```

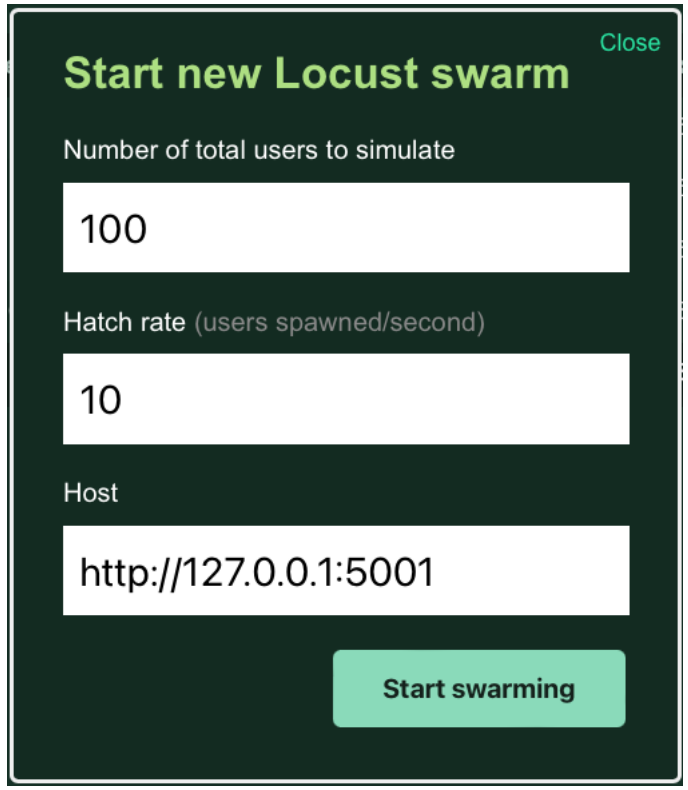
- Browse to http://localhost:8089
- **Set up and run your tests!**

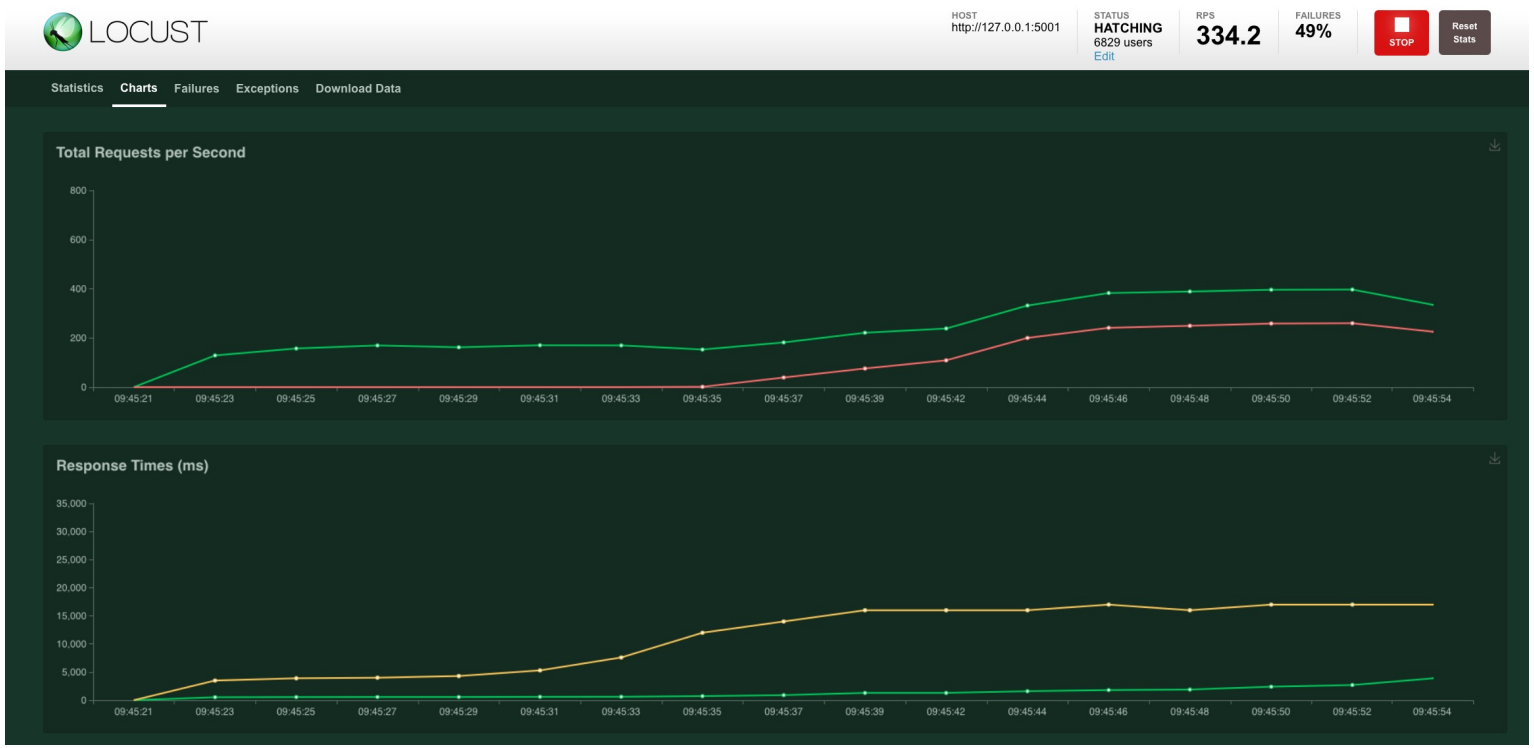
locustfile.py

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


class QuickstartUser(HttpUser):
    wait_time = between(1, 2)

    @task
    def index_page(self):
        self.client.get("/")
        self.client.get("/")

    @task(3) # 3 times more likely to be chosen than other tasks
    def upper(self):
        for _ in range(10):
            self.client.get(f"/str/upper?a=aaaa", name="/str/upper")
            time.sleep(1)

    @task
    def mul(self):
        for a in range(10):
            self.client.get(f"/math/mul?a={a}00&b=99999", name="/math/mul")
            time.sleep(1)

    @task
    def stats(self):
        for _ in range(10):
            self.client.get(f"/stats", name="/stats")
            time.sleep(1)
```

Stress your microservice!

http://localhost:8089





Exercise

- Extend the `locustfile.py` with more tasks.
- Spot the bottleneck service, if any.
- Assuming it is `string-service`, you can scale it by means of:

```
docker-compose up -d --scale string-service=6 --no-recreate
```