TESTING

Alessandro Bocci

name.surname@unipi.it

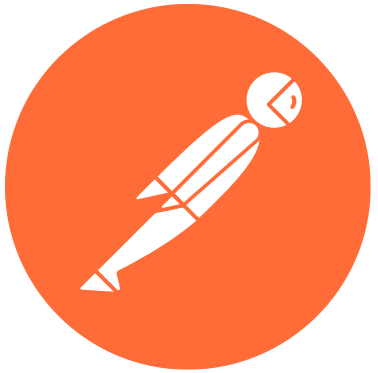
Advanced Software Engineering (Lab)

31/10/2025

What will you do?

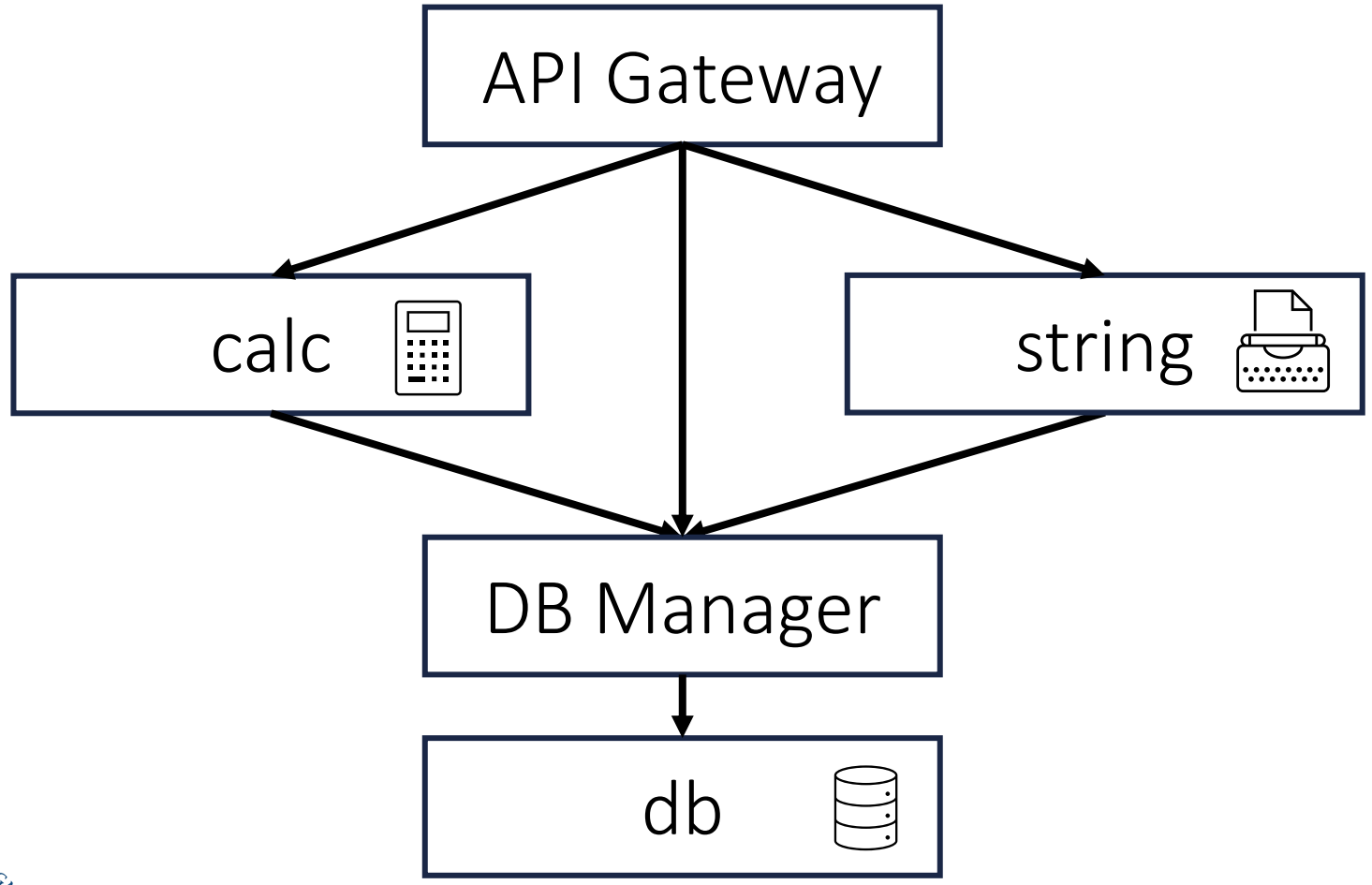
- Learn how to test microservices in isolation.
- Write unit tests to find bugs.
- Write performance tests to find bottlenecks.
- Remove bugs and bottlenecks.

Software Prerequisites



- Postman (creat an account and download the app).
- (optional) if you are familiar with npm and js, you can install Newman to run Postman from the terminal.
- Locust `pip install locust`
- Docker images for **`microase`** `(python:3.12-slim, mongo:latest)`
- **`microase`** folder (from Moodle)

microase architecture

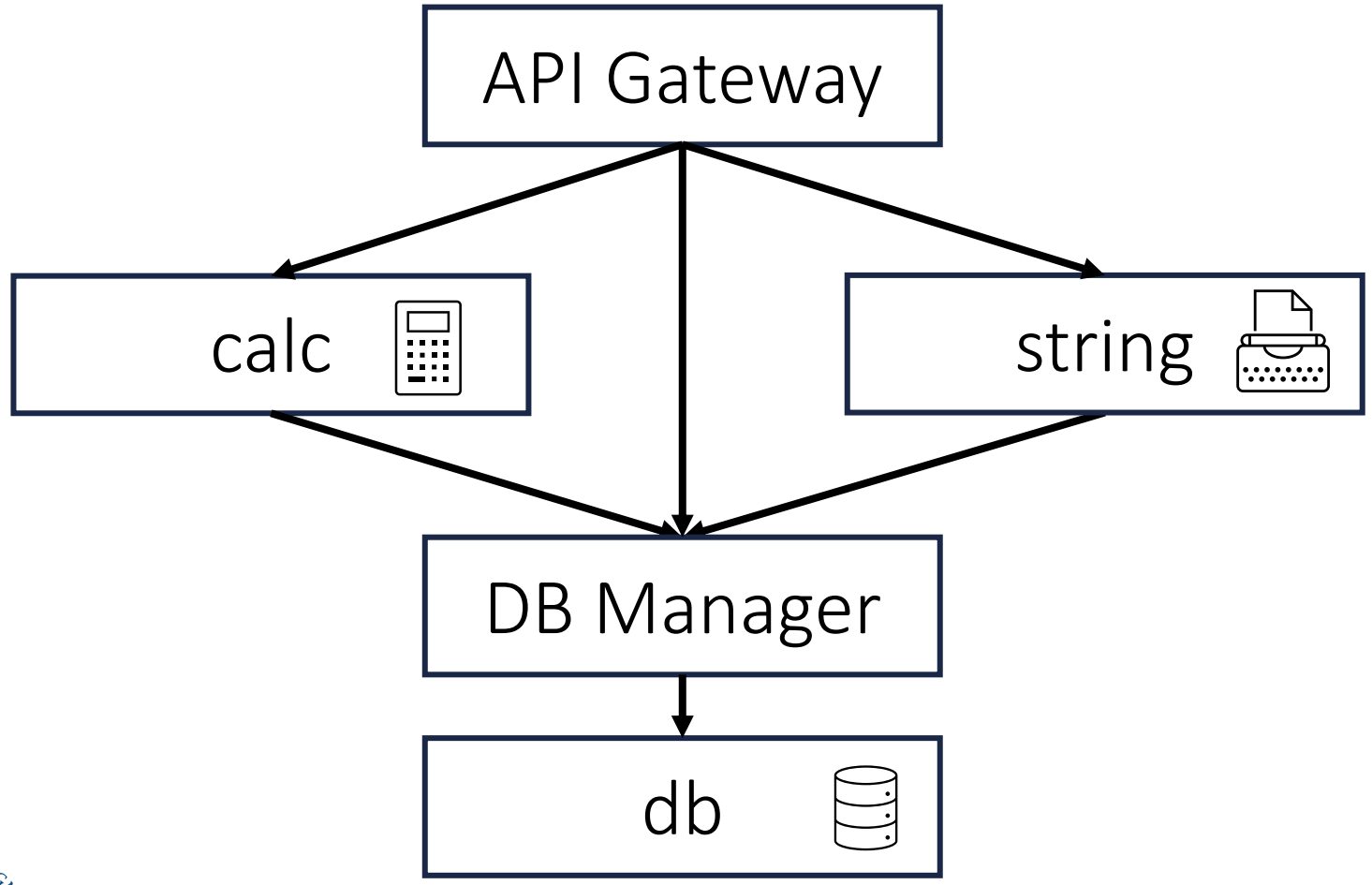


You have the OpenAPI files for all the microservices.

This is the API of the gateway:

GET	`/calc/{op}`	Perform a math operation
GET	`/str/{op}`	Perform a string operation
GET	`/getAll`	Get all records from the DB Manager

microase architecture



I added a `/getAll` endpoint to the API to retrieve all the content of the DB.

It is exposed by the API Gateway and the DB Manager.

The code I was providing in previous labs…

… contains bugs!

It has purposely bugs waiting for this class.

The code of today has new bugs.

Testing microservices

- **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**: Look for vulnerabilities or attacks (We will see something later in the course).

Unit tests basics

Unit testing is a software testing method where individual components (or "units") of a program are tested in isolation to ensure they function correctly.

A unit can be a function, a class, an endpoint, a microservice…

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

Unit test example – single function

Imagine testing a single function `div(a,b)`

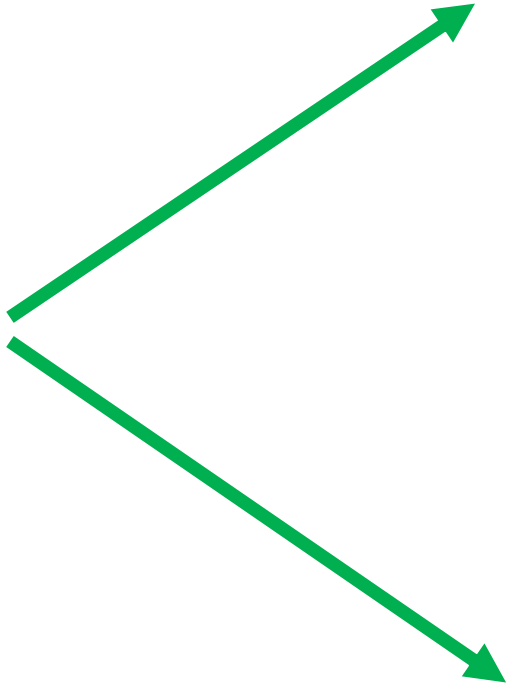
To do the unit test, you need:

- A set of input values for `a` and `b`
- The expected correct output of each test case.
- An assertion that compares the execution of the function with the expected output.

Unit test example - pytest

Function to test

Test cases



```python
import pytest

def div(a, b):
    return a / b


def test_div_positive_numbers():
    assert div(10, 2) == 5

def test_div_negative_numbers():
    assert div(-10, -2) == 5

def test_div_positive_and_negative():
    assert div(-10, 2) == -5

def test_div_by_zero():
    with pytest.raises(ZeroDivisionError):
        div(10, 0)

def test_div_floats():
    assert div(7.5, 2.5) == 3
```

This is an example where the unit is a Python function.

Unit test example - pytest

Function to test

- Function call with inputs
- Expected correct result
- Assertion of equivalence

```python
import pytest

def div(a, b):
    return a / b

def test_div_positive_numbers():
    assert div(10, 2) == 5

def test_div_negative_numbers():
    assert div(-10, -2) == 5

def test_div_positive_and_negative():
    assert div(-10, 2) == -5

def test_div_by_zero():
    with pytest.raises(ZeroDivisionError):
        div(10, 0)

def test_div_floats():
    assert div(7.5, 2.5) == 3
```

This is an example where the unit is a Python function.

Unit test example - pytest

Function to test

- Function call with inputs
- Expected correct result
- Assertion of equivalence

```python
import pytest

def div(a, b):
    return a / b

def test_div_positive_numbers():
    assert div(10, 2) == 5

def test_div_negative_numbers():
    assert div(-10, -2) == 5

def test_div_positive_and_negative():
    assert div(-10, 2) == -5

def test_div_by_zero():
    with pytest.raises(ZeroDivisionError):
        div(10, 0)

def test_div_floats():
    assert div(7.5, 2.5) == 3
```

This is an example where the unit is a Python function.

Unit test example - pytest

Function to test

- Function call with inputs
- Expected correct result
- Assertion of equivalence

```python
import pytest

def div(a, b):
    return a / b

def test_div_positive_numbers():
    assert div(10, 2) == 5

def test_div_negative_numbers():
    assert div(-10, -2) == 5

def test_div_positive_and_negative():
    assert div(-10, 2) == -5

def test_div_by_zero():
    with pytest.raises(ZeroDivisionError):
        div(10, 0)

def test_div_floats():
    assert div(7.5, 2.5) == 3
```

This is an example where the unit is a Python function.

Unit test example - pytest

Function to test

- Function call with inputs
- Expected correct result
- Assertion of equivalence

```python
import pytest

def div(a, b):
    return a / b

def test_div_positive_numbers():
    assert div(10, 2) == 5

def test_div_negative_numbers():
    assert div(-10, -2) == 5

def test_div_positive_and_negative():
    assert div(-10, 2) == -5

def test_div_by_zero():
    with pytest.raises(ZeroDivisionError):
        div(10, 0)

def test_div_floats():
    assert div(7.5, 2.5) == 3
```

This is an example where the unit is a Python function.

Unit test example - pytest

Function to test 

If the assertion is true the test is passed otherwise is failed.

This is an example where the unit is a Python function.

```python
import pytest

def div(a, b):
    return a / b


def test_div_positive_numbers():
    assert div(10, 2) == 5

def test_div_negative_numbers():
    assert div(-10, -2) == 5

def test_div_positive_and_negative():
    assert div(-10, 2) == -5

def test_div_by_zero():
    with pytest.raises(ZeroDivisionError):
        div(10, 0)

def test_div_floats():
    assert div(7.5, 2.5) == 3
```

Unit test example - pytest

Function to test

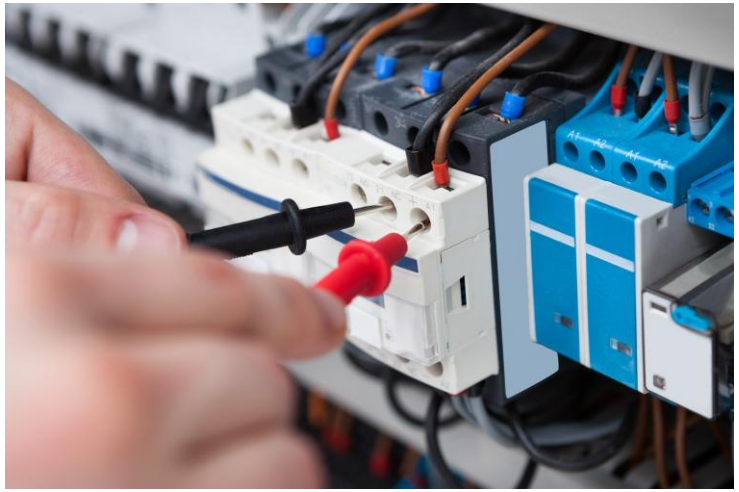
Negative testing for ‘failing’ behavior

```python
import pytest

def div(a, b):
    return a / b


def test_div_positive_numbers():
    assert div(10, 2) == 5

def test_div_negative_numbers():
    assert div(-10, -2) == 5

def test_div_positive_and_negative():
    assert div(-10, 2) == -5

def test_div_by_zero():
    with pytest.raises(ZeroDivisionError):
        div(10, 0)

def test_div_floats():
    assert div(7.5, 2.5) == 3
```

This is an example where the unit is a Python function.

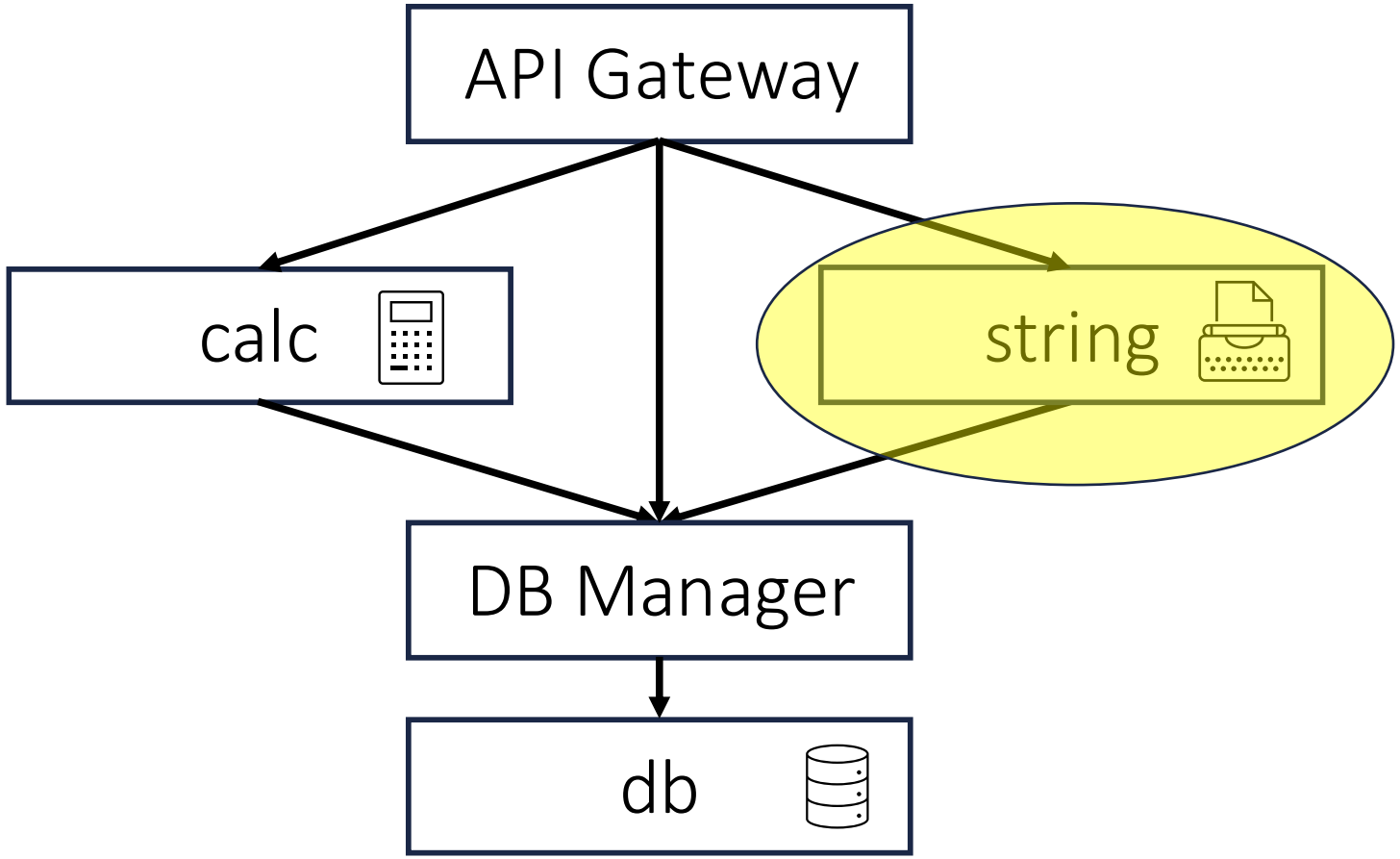
Unit tests and microservices

- Functional tests for a microservice project are all the tests that interact with the **published API (endpoints)** 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**.



Test in isolation

Testing a component separated by all the others.



Let's focus only on the string service

The string microservice

We can say that it has a dependency on the `DB Manager`, as each operation needs to be logged.

The string microservice

Test in isolation is done without other microservices.

The string microservice

Test in isolation is done without other microservices.

How can `string` work if it needs to send logs to the `DB Manager`?

Mock the dependencies

We simulate the interaction with the dependencies in a controlled way, using a *mock*.

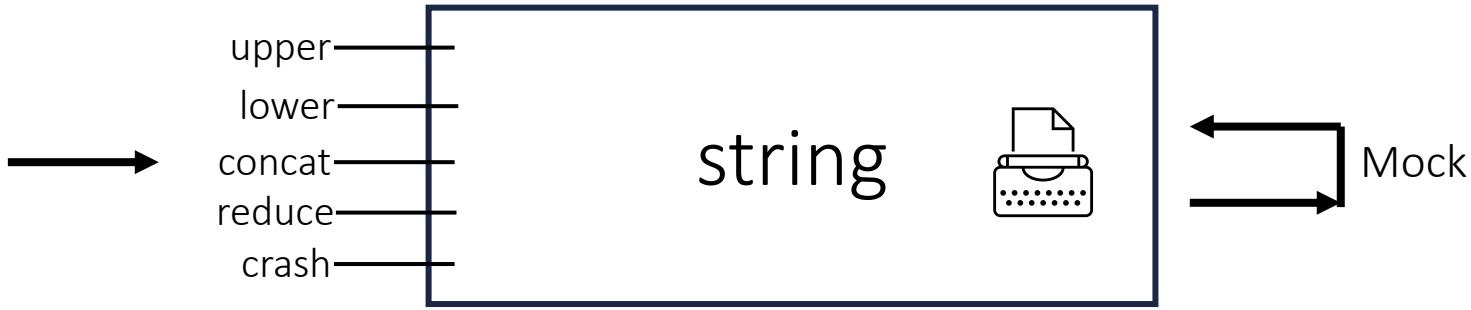
In general, we add ‘testing code’ that is not used in production but is used only during the testing phase.

Mock the dependencies

For example: here the DB is used only to write, so we can print the logged operation instead of sending the data to the DB.

In other cases, you can also use mock libraries to generate different kinds of mock values.

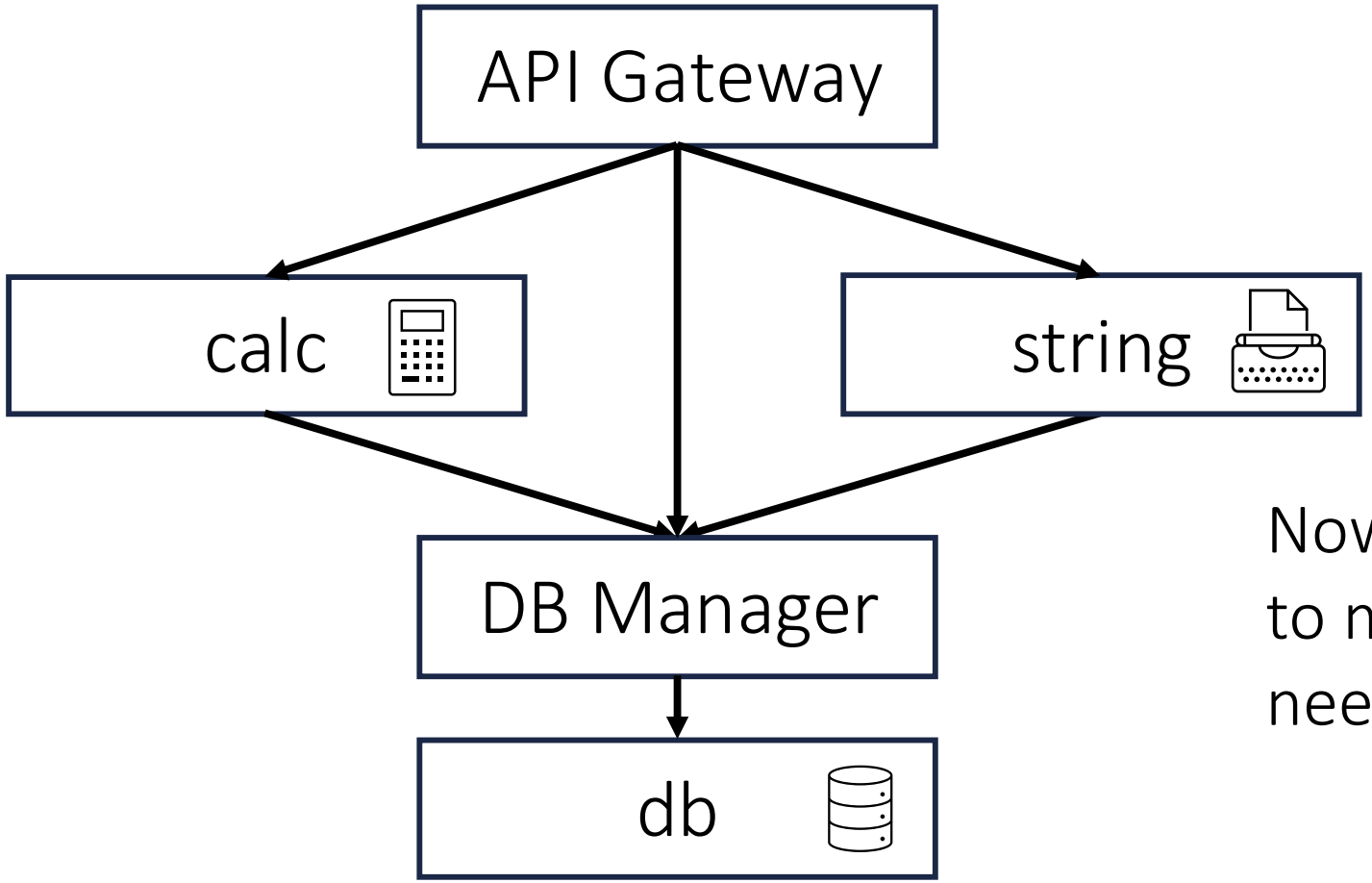
Mock the dependencies



We then test the API by invoking the endpoints and checking whether the microservice behaves as expected.

Integration test

After testing the microservices in isolation, we test the architecture through the gateway.



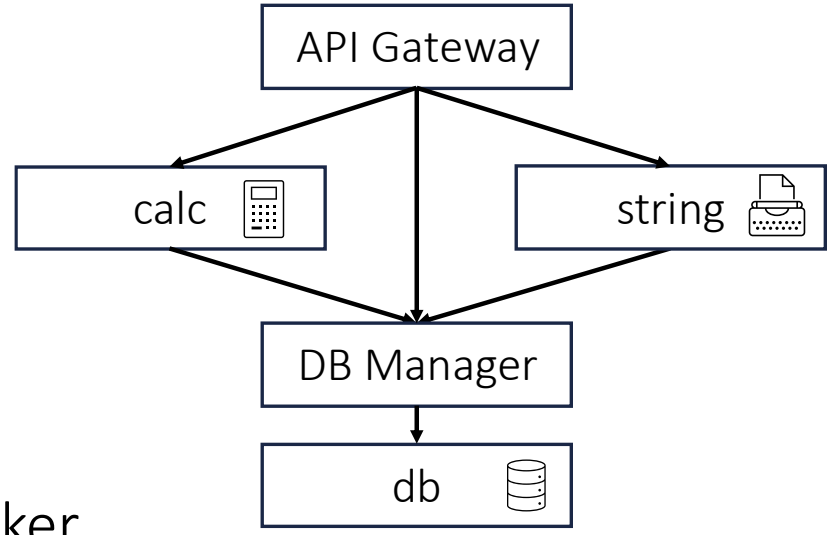
Now the ‘testing code’ used to mock dependencies is not needed

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.

Today's Lab



PART ONE – TESTS WITH POSTMAN

1. Write unit tests for:
   - `/sub`, `/mod`, `/random` for `calc`, and
   - `/lower` and `/concat` for `string`.
2. Run the tests for `calc` and `string` in isolation with Docker.
3. Perform integration testing, running `microase` with Docker compose by writing tests for the `gateway`.

Spot the bugs and resolve them if any. Check for symmetric code (e.g. lower and upper). Do not forget special cases (no input, zeros, numeric vs alphanumeric, etc.)!

PART TWO – PERFORMANCE TEST WITH LOCUST

1. Modify the `locustfile.py`, adding performance tests for the `gateway` to the operations of the previous endpoints.
2. Run the tests with `locust` for the `gateway`.
3. Spot the bottlenecks and resolve them.

Test microservices in isolation

The main steps for this part are:

1. Mock the dependencies, with as few changes as possible to the code to test.
2. Build and run the microservice alone using the mock code.
3. Test the API of the microservice.

Let’s see an example.

Test microservices in isolation

In this example for an HTTP GET. This function is called from other routes.

app.py

```python
def function(op):
    r = requests.get(f'http://other-service:5000/{op}')
    return r.json()
```

We have a dependency on **`other-service`**'s JSON.

In simple cases, we can only modify the code to have mock behaviour.

app.py

```python
mock_fun = None
def function(op):
    if mock_fun:
        return mock_fun(op)
    else:
        r = requests.get(f'http://other-service:5000/{op}')
        return r.json()
```

Mock behaviour

Normal behaviour

Test microservices in isolation

Then, we create a different file which imports the microservice code and implements the mock function.

This file will be used to execute isolation tests.

test_app.py

```
import app as main_app

flask_app = main_app.app #Flask app

def mock_fun(op): #Implementation
    if op == 'add':
        return {'s': 5}
    elif op == 'sub':
        return {'s': 3}

#Assigning the implementation in app.py
main_app.mock_fun = mock_fun
```

Those invocations will always answer with the same mock values.

Test microservices in isolation

There are several patterns to separate the test code, the previous example is the most basic one for Python apps.

Mock libraries allow to configure more sophisticated answers:

- Random choice from multiple values
- Errors for negative testing
- Complex objects
- Database interactions
- …

Test microservices in isolation

After adding the mock code, you have to run the microservice alone:

- Using a separate Dockefile for the testing, or
- Using a separate docker compose file with only a service for testing, or
- Using the ‘production’ Dockerfile but overriding CMD when running the container in order to use the mock code.

Test microservices in isolation

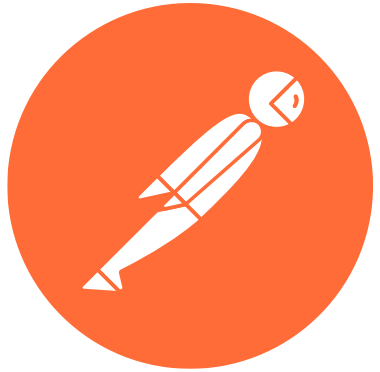
Recapping, you need:

- The code to test in a file.
- The testing code in a different file.
- A way to link them in order to switch the test on and off.

With Docker you can:

1. Have a Dockerfile for each mode (isolation, integration, production, etc.).
2. Have a single Dockerfile but replace the CMD during the run.

Unit testing with Postman



Postman is a tool to perform HTTP requests.

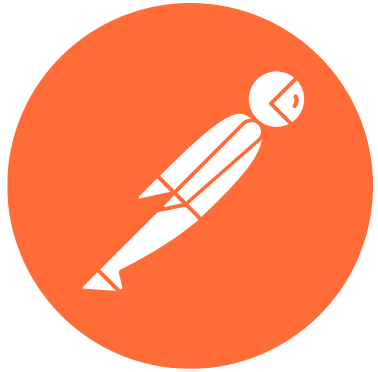
You can use the application or the VSCode extension (needs a free account).

Postman web do not support localhost requests.

Requests can be grouped in collections and are fully customisable:

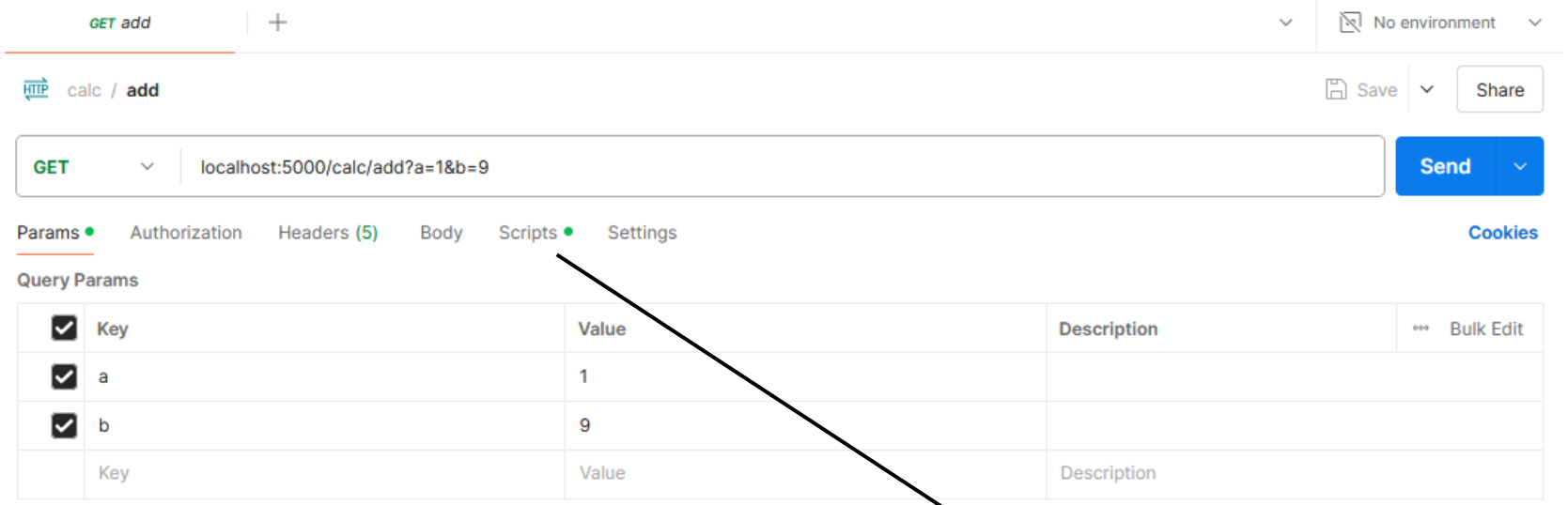
- HTTP method and URL,
- Headers,
- Authorization,
- Body, and
- Tests! (aka Scripts post response)

Unit testing with Postman



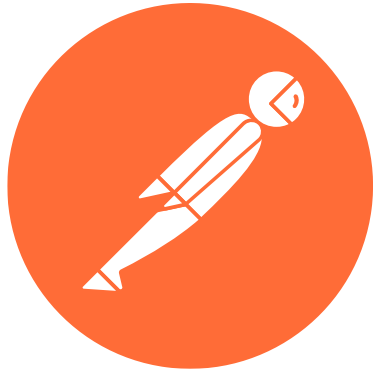
The tests are JavaScript scripts that use the Postman pm API.

For example:

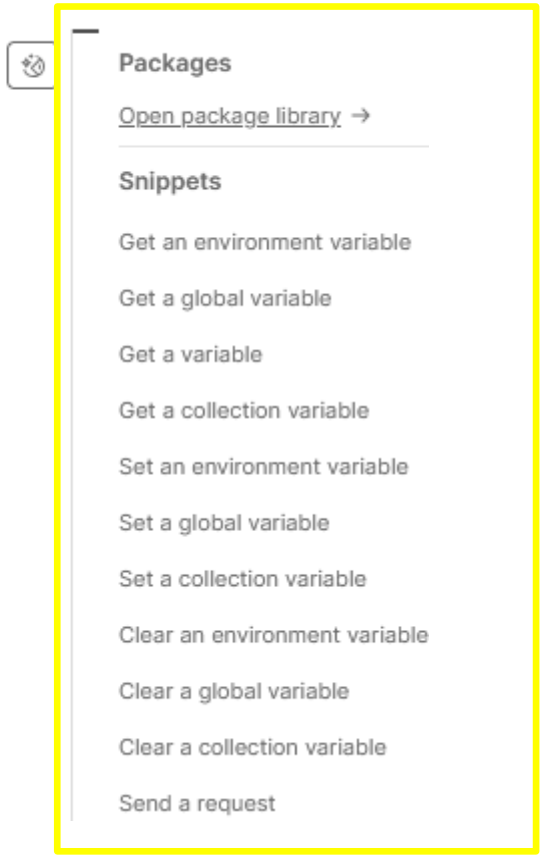



```
pm.test("Add basic test", function(){
    pm.response.to.have.status(200);
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.eql({ "s": 10 });
});
```

Unit testing with Postman

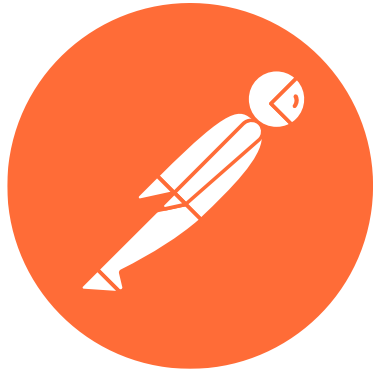


In the script sections you have plenty of code example.



Click on them to add the code example to scripts

Unit testing with Postman



Run a single request or a collection of requests to see the results of testing.

calc - Run results

Ran today at 12:04:24 · View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	1	596ms	4	26 ms

All Tests | Passed (4) | Failed (0) | Skipped (0)

Iteration 1

GET add
localhost:5000/calc/add?a=1&b=9

PASS Add test

GET sub
localhost:5000/calc/sub?a=10&b=9

PASS Sub test

GET mul
localhost:5000/calc/mul?a=10&b=9

PASS Mul test

GET random
localhost:5000/calc/random?a=1&b=10

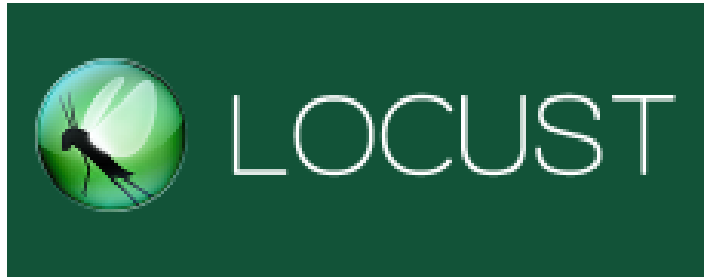
PASS Random test

Locust

An open-source load testing tool.

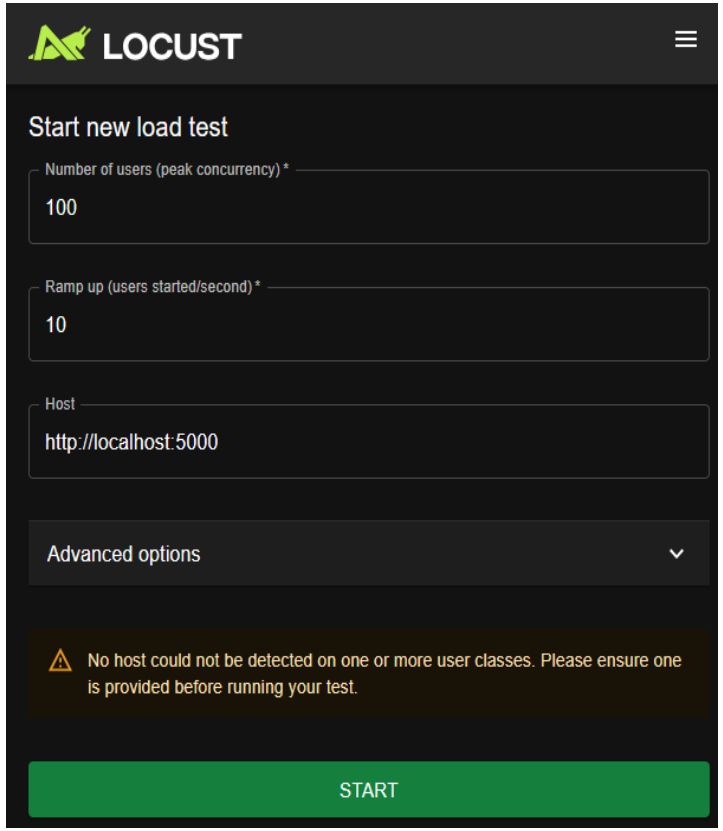
It uses a **`locustfile.py`** to define simulated users' behaviours (you have one with a first test example in the **`microase`** folder).

What you have to do is:



- Write tests in the **`locustfile.py`**.
- Start **`microase`** with docker compose.
- Run locust: `locust` in the root folder (new terminal).
- Browse to http://localhost:8089, set up and run your tests!

Stress your microservice!

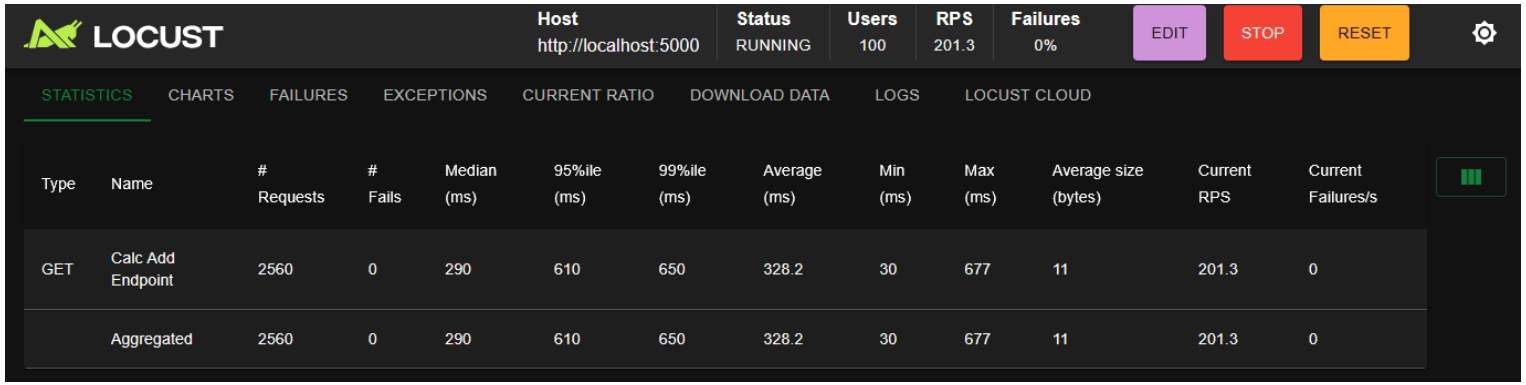


http://localhost:8089

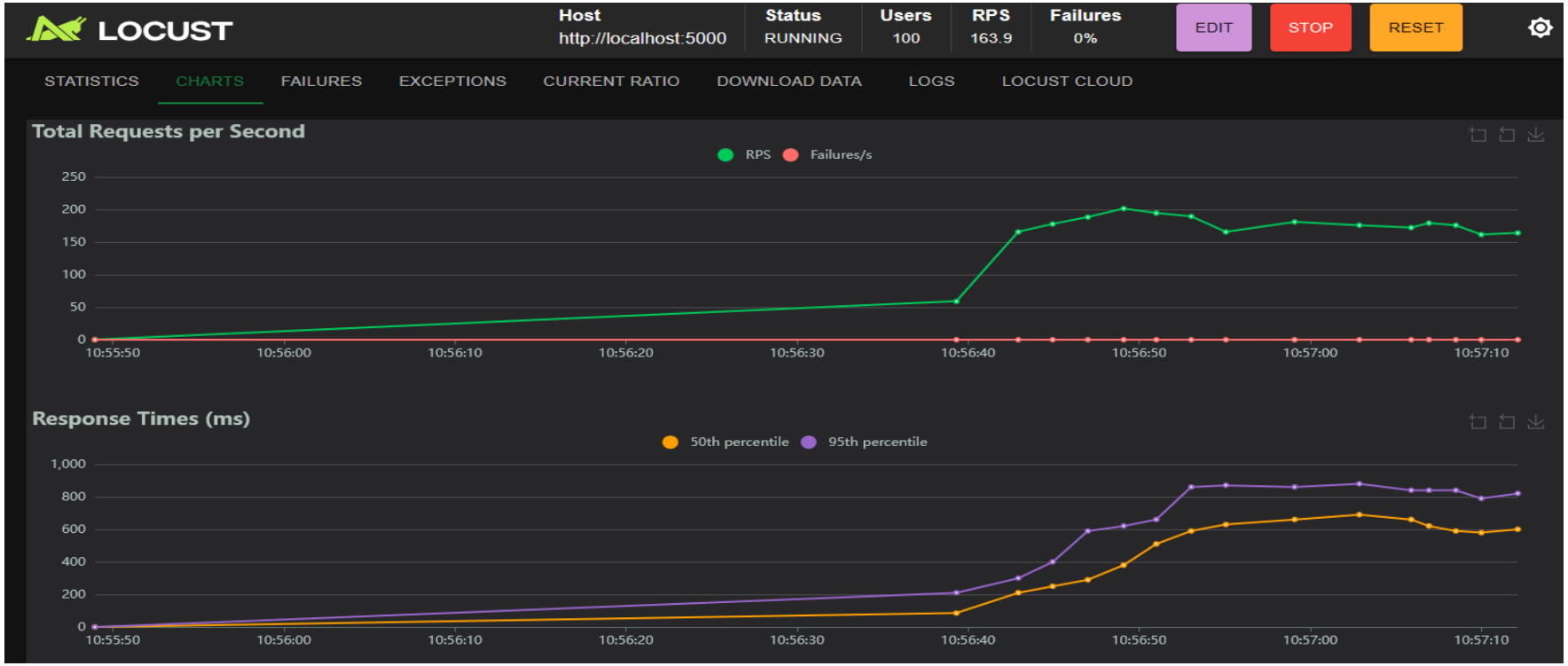
Select number of users for max concurrency and spawn rate per number of users started per second.

Do not forget to add the host's protocol (i.e. http://)!

Check the results!



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	Calc Add Endpoint	2560	0	290	610	650	328.2	30	677	11	201.3	0
	Aggregated	2560	0	290	610	650	328.2	30	677	11	201.3	0



Locustfile

```
from locust import HttpUser, task, between

class QuickstartUser(HttpUser):

    @task()
    def add(self):
        for a in range(1,11):
            self.client.get(f"/calc/add?a={a}&b=9", name = "Calc Add Endpoint")
```

This is an example to test the **`add`** endpoint 10 times for each user with **`a`** different a parameter.

- A valid **`locustfile`** has a class that extends **`User`** class (e.g. **`HttpUser`**).
- Each test is annotated with **`@task()`** representing a simulated user which performs that task.

Look for the bottlenecks

- Analyse locust’s stats and graphs.
- Spot the bottlenecks.
- Check the code of the endpoints in the **`gateway`** to resolve the possible problem.

In general, if you have a service that performs poorly, you can scale it by means of:

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

Recap

First Part:

1. Add mock code to the `calc` and `string` microservices.
2. Check the OpenAPI files to see the API of the microservices.
3. Write Postman collections to test the endpoints.
4. Execute the `calc` and `string` in isolation and run Postman tests.
5. Run the architecture with docker compose without using mock code.
6. Test the `gateway` endpoints with Postman.

Second Part:

1. Complete the locustfile.
2. Run locust for the performance test.

Each time you find a bug, correct it, rebuild and re-run.

Can you find all the bugs in my code?

Try to find the bugs in the code I provided by testing the microservices.

It is a simple application, the AI can find them easily without testing. The challenge of today is to find them via testing.

I will give you the list of bugs at the end of the class.

They are in `calc`, `string` and `gateway` code.

Lab take away

- ❑ Write and run unit tests to discover bugs in microservices.
- ❑ Write and run performance tests to discover bottlenecks microservice applications.

Project take away

- ❑ You will have to write unit tests of the project microservices.
- ❑ You will have to write performance test for the project, expecially to test gacha rolls rarity.

List of known bugs

- `calc`:
  - 0 as input is not accepted by any route with input **`a`** and **`b`**.
  - `/sub` there is no return value if one input is missing.
- `string`:
  - `/upper` and `/lower` use string “0” as default parameter for no input
  - all operations: accept non-alphanumeric values
- `gateway`:
  - when **`mod`** operation is used, **`b`** is always set at 1
  - The concatenation operation adds a quote (') character to the operand **`b`**
  - all operations: when parameters are empty it sends requests with parameters equal to **`None`**
- Performance: `/str/lower` has a 1 second delay in the **`gateway`**