September 6, 2020

#Python | #pytest | #blog

Testing HTTP client with pytest

Other pytest articles:

Why testing is important

Types of tests

Test driven Development

Hello, World!

Selecting tests with pytest

Testing HTTP client with pytest

Testing database with pytest

Advanced fixtures with pytest

Pytest plugins

Now let's move to checking if the number exists or not. For that, we are going to employ a 3rd party API. According to API docs:

- It's a REST API
- We need to use HTTP GET

- We provide a number in query parameters
- The result is a json `{'existing': True | False}`

I'm going to create this 3rd party API myself and run it from my local environment so we can see the access logs. Using access logs we can make sure if API is called or not.

API is a simple Flask application with a condition. It knows that a certain number is correct and all other numbers are incorrect. In the real world, I would use something like Nexmo number insight:

Copy

```
from flask import Flask, request, jsonify

app = Flask(__name__) # 1

@app.route('/') # 2
def hello():
    number = request.args.get("number")
    if number == '+3155512345':
        return jsonify({'exists': True})
    else:
        return jsonify({'exists': False}) # 3

app.run(port=8080)
```

1. That how Flask application is started - see more at https://flask.palletsprojects.com/
2. Out single route (URL) we will call
3. API return a simple json

Let's write 2 tests for existing and nonexisting numbers first.

Copy

```
from api import check_existence

def test_existing_number():
    number = '+3155512345' # 1
    assert check_existence(number) # 2


def test_non_existing_number():
    number = '+31'
    assert not check_existence(number)
```

1. Input number to check

2. The result of `check_existence` should return True or False

Now let's implement the code that will call the API. I'm going to use the requests library to call api. I'm going to catch low level exceptions and reraise our own application level exception here here.

Copy

```
import requests
from normalize import NumberValidationException


def check_existence(number):
    try:
        response = requests.get('http://127.0.0.1:8080/', params={'number': number}) #
        response.raise_for_status() # 2
        return response.json()['exists'] # 3
    except (requests.exceptions.HTTPError, KeyError) as e: # 4
        raise NumberValidationException('something went wrong when calling api') from
```

1. HTTP call to API with an input number as an URL parameter
2. Instruct HTTP client to raise an exception is return code is not 200 OK
3. Return the boolean result for received json
4. Mask low-level `requests` exceptions and re-raise application specific

Let's run these two tests and have a look at access logs - it's clear that we are calling the API. But unit tests should avoid doing it and we need to run them in isolation.

Copy

```
127.0.0.1 - - [25/Jun/2020 20:12:41] "GET /?number=%2B3155512345 HTTP/1.1" 200 -
127.0.0.1 - - [25/Jun/2020 20:12:41] "GET /?number=%2B31 HTTP/1.1" 200 -
```

Fortunately pytest has a solution for that. Let's first implement two classes for a response for existing and nonexisting numbers. The classes should have the same methods as an actual response class from requests library - or at least the ones that we use in our code.

Copy

```
class MockResponseExisting: # 1
    @staticmethod
    def json(): # 2
        return {'exists': True}

    @staticmethod
    def raise_for_status(): # 3
        pass
```

```
class MockResponseNonExisting: # 4
    @staticmethod
    def json():
        return {'exists': False}

    @staticmethod
    def raise_for_status():
        pass
```

1. Generic class that will represent a response from `requests` library
2. One of the method that is called in our `check_existence` function code - this is an emulation
3. Same as 3 but for another method we call in `check_existence` function
4. Another class that represents a response with different result

Now we need to change the test case to use the patch. This monkey patching trick will replace the get method from `requests` library with our own - `MockResponseExisting` and `MockResponseNonExisting`. Now if we run the tests and check the access logs we see that our tests didn't hit the real server - that is what we actually want from unit tests.

Copy

```
def test_existing_number(monkeypatch): # 1
    def mock_get(*args, **kwargs): # 2
        return MockResponseExisting()

    monkeypatch.setattr(requests, "get", mock_get) # 3

    number = '+3155512345'
    assert check_existence(number)


def test_non_existing_number(monkeypatch):
    def mock_get(*args, **kwargs):
        return MockResponseNonExisting()

    monkeypatch.setattr(requests, "get", mock_get)

    number = '+31'
    assert not check_existence(number)
```

1. For our test case function we put an argument `monkeypatch` that is made available by pytest itself
2. A helper function - it's just returns an instance of the mock response class

3. Monkey patching `requests` module - it replaces method in this module with our helper function on the fly

It looks that some code in our test_api.py is duplicated. Pytest provides a powerful feature called fixtures that will help to avoid duplication in that case.

Fixtures can be found in other testing frameworks and are used to set up the environment for testing.

First let's create a single class for response. Then let's create a fixture - a function decorated with pytest.fixture - called mock_response.

Copy

```
class MockResponse: # 1
    def __init__(self, result):
        self.result = result

    def json(self):
        return {'exists': self.result}

    @staticmethod
    def raise_for_status():
        pass


@pytest.fixture # 2
def mock_response(monkeypatch): # 3
    def mock_get(*args, **kwargs):
        return MockResponse(True) if kwargs['params']['number'] == '+3155512345' else

    monkeypatch.setattr(requests, "get", mock_get)
```

1. Same class as `MockResponseExisting` and `MockResponseNonExisting` above but a bit more generic
2. That's how we define that this function is a fixture
3. Fixture code itself - it's just some parts of our test case code

Now in the test cases we may use this function as a parameter, so pytest will invoke it every time for every test case. This makes test case code more clean - there is no setup code in the test case - only the logic.

Copy

```
def test_existing_number(mock_response): # 1
    number = '+3155512345' # 2
```

```
    assert check_existence(number)


def test_non_existing_number(mock_response):
    number = '+31'
    assert not check_existence(number)
```

1. Pass fixture as a parameter to the test case code- pytest will do the rest
2. No monkey patching in our test case code - it's done in fixture now

Mocking HTTP APIs is a pretty common task and Python community created a library called responses https://github.com/getsentry/responses - it allows us to avoid creating all this boilerplate code for mock_response ourselves.

Let's create a simple test using a response library. First we need to decorate the test case with responses.activate. Then it comes up with the results we expect from api. And again there are no calls to the real server.

Copy

```
import responses # 1

@responses.activate # 2
def test_non_existing_number_responses():
    responses.add(responses.GET, 'http://127.0.0.1:8080/', # 3
                  json={'exists': False}, status=200)
    number = '+31'
    assert not check_existence(number)
```

1. This is 3rd party tool, install with `pip install responses`
2. Decorator to activate `responses` for this particular test case
3. Configure `responses` - it will do monkey patching on the fly underneath

Of course we can use responses patches as fixtures:

Copy

```
from urllib.parse import quote


@pytest.fixture
def api_response(): # 1
    with responses.RequestsMock(assert_all_requests_are_fired=False) as rsps:
        rsps.add(responses.GET, 'http://127.0.0.1:8080/?number='+quote('+31'), # 2
                 json={'exists': False}, status=200)
```

```
        rsps.add(responses.GET, 'http://127.0.0.1:8080/?number=' + quote('+3155512345'
                 json={'exists': True}, status=200)
        rsps.add(responses.GET, 'http://127.0.0.1:8080/?number=' + quote('+311234555')
                 json={'exists': True}, status=200)
        yield rsps # 3
```

1. Define a new fixture
2. Configure `responses`, by passing URLs that we call in test cases and expected responses from these URLS
3. Since `responses` is in a separate fixture from test case code, there is no need to have `@responses.activate`, but using context manager nad yield is needed to active the tool

And then use this fixture automatically for a test case with marking the function with usefixtures decorator:

Copy

```
@pytest.mark.usefixtures("api_response") # 1
def test_non_existing_number_responses_fixture(): # 2
    number = '+31'
    assert not check_existence(number)
```

1. Mark a test case with this decorator so fixture is invoked automatically
2. No need to pass fixture as an argument if it's used automatically

Now we can move all created fixtures to `conftest.py` so they can be shared with other test cases. `Conftest.py` is a special file for pytests - You don't need to import the fixture you want to use in a test, it automatically gets discovered by pytest.

In this unit you've learned what mocks are, how to use pytest fixtures. In the next one you'll learn how to test database interfaces and how dependency injection can help.

Similar articles:

- Advanced fixtures with pytest
- Hello, World!
- Pytest plugins
- Selecting tests with pytest
- Test driven Development