Pytest Tricks for Better Python Tests

Fri, Dec 21, 2018

Pytest is my Python testing framework of choice. Very easy to use, and makes tests look much better.

In this article I'll show you some cool tricks I have incorporated into my test suites using Pytest.

Environment Variables

When your application must work with defined environment variables, the testing environment must have these variables defined as well, even if the values are not real.

With Pytest we can easily configure any necessary environment variables in our test environment. We simply create a fixture in `conftest.py` that will be loaded automatically:

```
# conftest.py

import pytest


@pytest.fixture(autouse=True)
def env_setup(monkeypatch):
    monkeypatch.setenv('MY_SETTING', 'some-value')
    monkeypatch.setenv('ANOTHER_SETTING', 'some-value')
```

`monkeypatch` is a built-in pytest fixture that allows us to set environment variables in the test runs. By enabling the `autouse` option, our custom environment setup fixture will be automatically called in every test without having to include it explicitly using the usual dependency injection mechanism.

AWS Mock Fixtures

We can create reusable fixtures that mock AWS services, using the awesome moto library:

```
# conftest.py

import pytest
import moto
import boto3


TEST_BUCKET_NAME = 'test-bucket'
TEST_DYNAMO_TABLE_NAME = 'test-dynamodb-table'


@pytest.fixture
def s3_bucket():
    with moto.mock_s3():
        boto3.client('s3').create_bucket(Bucket=TEST_BUCKET_NAME)
        yield boto3.resource('s3').Bucket(TEST_BUCKET_NAME)


@pytest.fixture
def dynamodb_table():
    with moto.mock_dynamodb2():
        boto3.client('dynamodb').create_table(
            AttributeDefinitions=[
```

```
                {'AttributeName': 'id', 'AttributeType': 'S'}
            ],
            TableName=TEST_DYNAMO_TABLE_NAME,
            KeySchema=[{'AttributeName': 'id', 'KeyType': 'HASH'}],
            ProvisionedThroughput={
                'ReadCapacityUnits': 123,
                'WriteCapacityUnits': 123,
            },
        )
        yield boto3.resource('dynamodb').Table(TEST_DYNAMO_TABLE_NAME)
```

The `yield` keyword in the fixture will allow this fixture to be kept alive during the whole execution of a test that uses it. Using these fixtures in tests is very straight-forward:

```
class MyTest:
    def test_upload(self, s3_bucket):
        # Call some function that uploads some files
        # to S3 under a 'directory' denoted by an id
        my_func()

        # Assert that the file is in the bucket
        files_in_bucket = s3_bucket.objects.filter(Prefix=f'{SOME_ID}/')
        assert len(list(files_in_bucket)) == 3
```

Fixtures That Accept Arguments

There are times when I want to write some sort of fixture factory that can produce fixtures with different properties depending on parameters passed to it.

For example, let's say we want a fixture that represents some JSON payload, but some values should be generated according to some identifier that is passed to the fixture.

A real example of this would be a JSON payload of an Amazon SQS event that gets triggered when some file is uploaded to a S3 bucket:

```
# conftest.py

import json

import pytest


@pytest.fixture
def sqs_event_payload():
    def _payload(id):
        s3_key = f'{id}/{id}.pdf'

         msg = {
            'Records': [
                {
                    's3': {
                        'bucket': {'name': TEST_BUCKET_NAME},
                        'object': {'key': s3_key},
                    }
                }
            ]
        }
        body = {'Message': json.dumps(msg)}
        return {'Records': [{'body': json.dumps(body)}]}
```

```
    return _payload
```

The trick here is using an inner function to enable the “parameterization” of the fixture. We can then use it in a test like this:

```
class MyTest:
    def test_something(self, sqs_event_payload):
        payload = sqs_event_payload('1093450983456')

        # Send the payload to a function we want to test.
        # For example, a Lambda handler function
        r = handler(event=payload, context=None)

        # Assert something here...
```

This allows us to be able to specify a dynamic ID to the fixture. This way we don't have to worry about using the same ID in every test.

One drawback of this trick though, is that we have to call the fixture as a function first to be able to use it, as opposed to the normal way of using fixtures by just referring to them.

Mocking Files

If your application deals with processing certain types of files such as images or PDF files, then you will need to create appropriate mocks of these files in your tests.

Image Files

Here is how I mock image files:

```
from io import BytesIO
from PIL import Image

import pytest


@pytest.fixture
def image_mock():
    file = BytesIO()
    image = Image.new('RGBA', size=(50, 50), color=(155, 0, 0))
    image.save(file, 'png')
    file.name = 'test.png'
    file.seek(0)
    return file
```

PDF Files

Here is how I mock PDF files using PyPDF2:

```
import pytest
import PyPDF2


@pytest.fixture
def pdf_file(tmpdir_factory):
    file_name = 'pdftest.pdf'
```

```
    # Fill with blank pages
    writer = PyPDF2.PdfFileWriter()
    for _ in range(num_pages):
        writer.addBlankPage(width=100, height=100)

    # You can also throw in some bookmarks
    writer.addBookmark(title='Intro', pagenum=0)
    writer.addBookmark(title='Index', pagenum=1)
    writer.addBookmark(title='Epilogue', pagenum=2)

    fn = tmpdir_factory.mktemp('tmp', numbered=True).join(file_name)
    with open(fn, 'wb') as pdf:
        writer.write(pdf)

    return fn
```

If you need more customization, you can apply the parameterized fixture principles I discussed in the previous section.

You can also combine this trick with the AWS S3 mock previously mentioned so that you can simulate file uploads to S3. In your tests you could do the following:

```
class MyTest:
    def test_something(self, s3_bucket, image_mock):
        # Upload a mock image to S3
        s3_bucket.put_object(Key='my_image.png', Body=image_mock)
```

python pytest tdd

Comments