26th January 2016 11:40 pm

Mocking External Apis in Python

It's quite common to have to integrate an external API into your web app for some of your functionality. However, it's a really bad idea to have requests be sent to the remote API when running your tests. At best, it means your tests may fail due to unexpected circumstances, such as a network outage. At worst, you could wind up making requests to paid services that will cost you money, or sending push notifications to clients. It's therefore a good idea to mock these requests in some way, but it can be fiddly.

In this post I'll show you several ways you can mock an external API so as to prevent requests being sent when running your test suite. I'm sure there are many others, but these have worked for me recently.

Mocking the client library

Nowadays many third-party services realise that providing developers with client libraries in a variety of languages is a good idea, so it's quite common to find a library for interfacing with a third-party service. Under these circumstances, the library itself is usually already thoroughly tested, so there's no point in you writing additional tests for that functionality. Instead, you can just mock the client library so that the request is never sent, and if you need a response, then you can specify one that will remain constant.

I recently had to integrate Stripe with a mobile app backend, and I used their client library. I needed to ensure that I got the right result back. In this case I only needed to use the `Token` object's `create()` method. I therefore created a new

Recent Posts

What I Want in a PHP CMS

Flow Typed AJAX Responses With React Hooks

Caching the Laravel User Provider With a Decorator

The Trouble With Integrated Static Analysis

Don't Use Stdclass

About me

I'm a web and mobile app developer based in Norfolk. My skillset includes Python, PHP and Javascript, and I have extensive experience working with CodeIgniter, Laravel, Zend Framework, Django, Phonegap and React.js.

Links

GitHub

Twitter

Stack Overflow

Dev.to

`MockToken` class that inherited from `Token`, and overrode its `create()` method so that it only accepted one card number and returned a hard-coded response for it:

```
from stripe.resource import Token, convert_to_stripe_objec
from stripe.error import CardError


class MockToken(Token):

    @classmethod
    def create(cls, api_key=None, idempotency_key=None,
                stripe_account=None, **params):
        if params['card']['number'] != '4242424242424242':
            raise CardError('Invalid card number', None, 4

        response = {
            "card": {
              "address_city": None,
              "address_country": None,
              "address_line1": None,
              "address_line1_check": None,
              "address_line2": None,
              "address_state": None,
              "address_zip": None,
              "address_zip_check": None,
              "brand": "Visa",
              "country": "US",
              "cvc_check": "unchecked",
              "dynamic_last4": None,
              "exp_month": 12,
              "exp_year": 2017,
              "fingerprint": "49gS1c4YhLaGEQbj",
              "funding": "credit",
              "id": "card_17XXdZGzvyST06Z022EiG1zt",
              "last4": "4242",
              "metadata": {},
              "name": None,
              "object": "card",
              "tokenization_method": None
          },
            "client_ip": "192.168.1.1",
            "created": 1453817861,
            "id": "tok_42XXdZGzvyST06Z0LA6h5gJp",
            "livemode": False,
            "object": "token",
            "type": "card",
            "used": False
        }
        return convert_to_stripe_object(response, api_key,
```

Much of this was lifted straight from the source code for the library. I then wrote a test for the payment endpoint and patched the `Token` class:

```python
class PaymentTest(TestCase):
    @mock.patch('stripe.Token', MockToken)
    def test_payments(self):
        data = {
            "number": '1111111111111111',
            "exp_month": 12,
            "exp_year": 2017,
            "cvc": '123'
        }
        response = self.client.post(reverse('payments'), d
        self.assertEqual(response.status_code, status.HTTP
```

This replaced `stripe.Token` with `MockToken` so that in this test, the response from the client library was always going to be the expected one.

If the response doesn't matter and all you need to do is be sure that the right method would have been called, this is easier. You can just mock the method in question using `MagicMock` and assert that it has been called afterwards, as in this example:

```python
class ReminderTest(TestCase):
    def test_send_reminder(self):
        # Mock PushService.create_message()
        PushService.create_message = mock.MagicMock(name="

        # Call reminder task
        send_reminder()

        # Check user would have received a push notificati
        PushService.create_message.assert_called_with([{'t
```

Mocking lower-level requests

Sometimes, no client library is available, or it's not worth using one as you only have to make one or two requests. Under these circumstances, there are ways to mock the actual request to the external API. If you're using the `requests` module, then there's a `responses` module that's ideal for mocking the API request.

Suppose we have the following code:

```
import json, requests

def send_request_to_api(data):
    # Put together the request
    params = {
        'auth': settings.AUTH_KEY,
        'data': data
    }
    response = requests.post(settings.API_URL, data={'para
    return response
```

Using `responses` we can mock the response from the server in our test:

```
class APITest(TestCase):
    @responses.activate
    def test_send_request(self):
        # Mock the API
        responses.add(responses.POST,
            settings.API_URL,
            status=200,
            content_type="application/json",
            body='{"item_id": "12345678"}')

        # Call function
        data = {
            "surname": "Smith",
            "location": "London"
        }
        send_request_to_api(data)

        # Check request went to correct URL
        assert responses.calls[0].request.url == settings.
```

Note the use of the `@responses.activate` decorator. We use `responses.add()` to set up each URL we want to be able to mock, and pass through details of the response we want to return. We then make the request, and check that it was made as expected.

You can find more details of the `responses` module here.

Summary

I'm pretty certain that there are other ways you can mock an external API in Python, but these ones have worked for me recently. If you use another method, please feel free to share it in the comments.

python tdd mock

3

2 Comments Matthew Daly's Blog Disqus' Privacy Policy 1 Login

Recommend 1 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

arpanette • 2 years ago

This is really helping me understand mocking -- I appreciate you sharing!

^ | ˅ • Reply • Share ›

Martin • 3 years ago

Super helpful, thanks for sharing!

^ | ˅ • Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Do Not Sell My Data

< Learning More About React.js and Flux | My Experience Using PHP 7 in Production >