David Cramer's Blog

Mocking Python Requests with Responses

20 May 2014

My main project at Dropbox has been a new automated build system (we call it Changes), mostly focused on code quality. Right now it's a very thick layer on top of Jenkins, which means we do a significant amount of HTTP requests between the two. I'm also a pretty lazy developer, and I hate testing my code (manually, at least). This left us with a pretty tricky problem in Python: reasonable HTTP mocks.

I've done this in the past a few ways. Generally it ends up looking something like this:

```
@mock.patch('urllib2.urlopen')
def test_simple(mock_urlopen):
    mock_urlopen.return_value = HttpResponse('{"message": "ok!"}')

    # .. do some tests ..

    # simple assertions
    mock_urlopen.assert_called_once_with('http://foo.com')
```

Eventually this leads to write a few abstractions around mocking urllib2 since theres a number of things we need to test (most of which require significantly more boilerplate code):

- Various responses (both successful and not)
- Query strings and POST bodies
- Headers, side effects, etc.

We end up down a rabbit hole pretty quickly.

Exploring HTTPretty

Early on we decided we would use the Requests library for the project. While we didn't need a majority of the features, there were a couple of nice abstractions that we were certainly going to make use of.

As we began exploring solutions for mocking HTTP requests we ran into HTTPretty. The library itself brought a great API into a world of extreme complications.

A simple example:

```
@httpretty.activate
def test_yipit_api_returning_deals():
    httpretty.register_uri(httpretty.GET, "http://api.yipit.com/v1/deals/",
                           body='[{"title": "Test Deal"}]',
                           content_type="application/json")

    response = requests.get('http://api.yipit.com/v1/deals/')

    assert response.json() == [{"title": "Test Deal"}]
```

Under the hood the library mocks out low level sockets and reimplements HTTP protocols. Unfortunately as great as it sounded, we couldn't quite get it working for our test cases. It's all possible that the issues are resolved now, but this situation led us to look for alternatives, eventually rolling our own.

Introducing Responses

After digging into the Requests internals we quickly realized we could write a mock adapter that would act very similar to the API presented in HTTPretty. In the end, that led us to building Responses, a mock library for Requests.

Responses works almost identically to HTTPretty, albeit with less features (pull requests welcome!):

```
@responses.activate
def test_yipit_api_returning_deals():
    responses.add(responses.GET, "http://api.yipit.com/v1/deals/",
                  body='[{"title": "Test Deal"}]',
                  content_type="application/json")

    response = requests.get('http://api.yipit.com/v1/deals/')

    assert response.json() == [{"title": "Test Deal"}]
```

As you can see, the code is almost identical. There's a slight namespace change and we register mocks using the `add` method.

Under the Hood

Internally Responses actually does very little. In fact, the current version is under 200 lines of code. A few simple structures for storing mocked requests, some processing logic, and an API that ends up looking very similar to the sessions code within Requests.

The meat of it is handled in two very legible functions, firstly, a hook which replaces the `Session.send` API mechanism:

```
# slightly truncated to keep the blog post bearable
    def _on_request(self, request, **kwargs):
        match = self._find_match(request)

        headers = {
            'Content-Type': match['content_type'],
        }
        if match['adding_headers']:
            headers.update(match['adding_headers'])

        response = HTTPResponse(
            status=match['status'],
            body=BufferIO(match['body']),
            headers=headers,
            preload_content=False,
        )

        adapter = HTTPAdapter()

        response = adapter.build_response(request, response)
        if not match['stream']:
            response.content

        return response
```

Secondly, a helper function for the request handler which simply looks at a list of maps for a registered response:

```
def _find_match(self, request):
        url = request.url
        url_without_qs = url.split('?', 1)[0]

        for match in self._urls:
            if request.method != match['method']:
                continue

            if match['match_querystring']:
                if not re.match(re.escape(match['url']), url):
                    continue
            else:
                if match['url'] != url_without_qs:
                    continue
            return match

        return None
```

And of course, this all gets wired up using the wonderful mock library:

```
def start(self):
        import mock
        self._patcher = mock.patch('requests.Session.send', self._on_request)
        self._patcher.start()
```

In the Real World

While our integration code for Jenkins is much less bearable, Responses has made writing tests that accurately represent Jenkins very easy. As an example, here's a chunk of one of our monolithic integration tests:

```
class JenkinsIntegrationTest(BaseTestCase):
    """
    This test should ensure a full cycle of tasks completes successfully within
    the jenkins builder space.
    """
    @mock.patch('changes.config.redis.lock', mock.MagicMock())
    @eager_tasks
    @responses.activate
    def test_full(self):
        responses.add(
            responses.POST, 'http://jenkins.example.com/job/server/build/api/json/',
            body='',
            status=201)
        responses.add(
            responses.GET, 'http://jenkins.example.com/queue/api/xml/?xpath=%2Fqueue%2Fitem%5Baction%2Fparameter%2Fname%3D%22CHANGES_BID%22+and+action%2Fparameter%2Fvalue%3D%2281d1596fd4d642f4a6bdf86c45e014e8%22%5D%2Fid',
            body=self.load_fixture('fixtures/GET/queue_item_by_job_id.xml'),
            match_querystring=True)
        responses.add(
            responses.GET, 'http://jenkins.example.com/queue/item/13/api/json/',
            body=self.load_fixture('fixtures/GET/queue_details_building.json'))
        responses.add(
            responses.GET, 'http://jenkins.example.com/job/server/2/api/json/',
            body=self.load_fixture('fixtures/GET/job_details_with_test_report.json'))
        responses.add(
            responses.GET, 'http://jenkins.example.com/job/server/2/testReport/api/json/',
            body=self.load_fixture('fixtures/GET/job_test_report.json'))
        responses.add(
            responses.GET, 'http://jenkins.example.com/job/server/2/logText/progressiveHtml/?start=0',
            match_querystring=True,
            adding_headers={'X-Text-Size': '7'},
            body='Foo bar')
        responses.add(
            responses.GET, 'http://jenkins.example.com/computer/server-ubuntu-10.04%20(ami-746cf244)%20(i-836023b7)/config.xml',
            body=self.load_fixture('fixtures/GET/node_config.xml'))
```

While one could argue that this could be solved by mocking out the Jenkins API and using simple dependency injection (and they'd be right), there's a lot of value in removing some levels of abstractions to simplify code. We could even take this example one step further and move all of our response mocks into a generic `JenkinsMockServer` which wraps `responses.start()` and `responses.stop()` to give us a more realistic and reusable picture of the remote server.

Tell Us What You think

We've had very positive feedback to the library since we published it to GitHub in November. It's pretty young and built primarily to solve our needs, but I'd love to hear how others are solving these same kinds of problems without adding huge layers of complexity.

Choosing Angular.js Be a Problem Solver Blog Archives

Comments

Comments for this thread are now closed.

2 Comments David Cramer's Blog 1 Login

Recommend 1 Share Sort by Best

rosscdh · 2 years ago
Very nice, would be great to be able to record all outgoing requests; and their actual responses (to build the test suite)
1 · Share ›

David Shawley · 2 years ago
We went a slightly different way and spin up an HTTPServer instance that records requests and can be preloaded with responses. Having a separate thread running for the test can be error prone. I'll have to investigate switching out our method for responses. I'm also interested in checking out changes. Thanks for putting both of these up on github.
· Share ›

Subscribe Add Disqus to your site Add Disqus Add Privacy

Copyright © 2016 David Cramer — Need exceptional error logging? Try out Sentry