Placebo Documentation

Release 0.0.1

Huseyin Yilmaz

October 24, 2016

Contents

1 Introduction **1**
1.1 Why this is useful . . . 1

2 Installation **5**

3 Usage **7**
3.1 Implementation example . . . 7

4 Impementing Placebo classes **9**
4.1 Static placebo classes . . . 9
4.2 Dynamic placebo classes . . . 9
4.3 Placebo properties . . . 10

5 Using placebo classes as decorator **13**

6 Getting a placebo instance **15**

7 Accessing the last mocked request **17**

8 Mocking with regex url **19**
8.1 Regex urls in httmock backend . . . 19
8.2 Regex urls in httpretty backend . . . 19

9 Backends **21**
9.1 Implementing a custom backend . . . 21

10 Caveats **23**
10.1 Separate url types for different backends . . . 23
10.2 Httppretty backend problems . . . 23

CHAPTER 1

Introduction

Placebo is a tool for mocking external API's in python applications. It uses httmock or httpretty as mocking backend. Placebo provides an highly compossible and reusable interface to those backends.

1.1 Why this is useful

Consider following function:

```
def get_movie(title, year):
    """Sends a request to omdbapi to get movie data.

    If there is a problem with connection, returns None.
    """
    url = 'http://www.omdbapi.com/'

    params = {'t': title,
              'y': year,
              'plot': 'short',
              'r': 'json'}
    response = requests.get(url, params=params)
    if not response.ok:
        return None
    resp = response.json()
    return {
        'language': resp['Language'],
        'director': resp['Director'],
        'rated': resp['Rated'],
        'title': resp['Title']}
```

Let's think about how we could test this function. A classic way to test this code is following:

```
class RawAPITests(TestCase):

    @mock.patch('requests.get')
    def test_get_movie_with_valid_response(self, fake_get):
        fake_response = fake_get.return_value
        fake_response.ok = True
        fake_response.json.return_value = {
            'Actors': 'Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving',  # noqa
            'Awards': 'Won 4 Oscars. Another 33 wins & 44 nominations.',
            'Country': 'USA',
            'Director': 'Lana Wachowski, Lilly Wachowski',
```

```
            'Genre': 'Action, Sci-Fi',
            'Language': 'English',
            'Metascore': '73',
            'Plot': 'A computer hacker learns from mysterious rebels about the true nature of his rea
            'Poster': 'http://ia.media-imdb.com/images/M/MV5BMTkxNDYxOTA4M15BM15BanBnXkFtZTgwNTk0NzQx
            'Rated': 'R',
            'Released': '31 Mar 1999',
            'Response': 'True',
            'Runtime': '136 min',
            'Title': 'The Matrix',
            'Type': 'movie',
            'Writer': 'Lilly Wachowski, Lana Wachowski',
            'Year': '1999',
            'imdbID': 'tt0133093',
            'imdbRating': '8.7',
            'imdbVotes': '1,204,431'}

        movie = get_movie('matrix', 1999)
        self.assertEqual(movie,
                         {'director': 'Lana Wachowski, Lilly Wachowski',
                          'rated': 'R',
                          'language': 'English',
                          'title': 'The Matrix'})

    @mock.patch('requests.get')
    def test_get_movie_with_500_response(self, fake_get):
        fake_response = fake_get.return_value
        fake_response.ok = False
        fake_response.json.return_value = ''

        movie = get_movie('matrix', 1999)
        self.assertEqual(movie, None)
```

These tests are fine. But they have two main disadvantages: First, code for mocking and actual function invocation is done in the same place which makes it hard to read. Second, if we keep writing similar tests we will probably copy the data for every test.

The purpose of placebo is to separate data from the actual tests. So It would be a lot easier to reason about. Placebo mocks will also be reusable and composable.

Here is how same code could be implemented with placebo:

First we create a placebo object for that endpoint.

```
class GetMovieValidResponse(Placebo):

    url = 'http://www.omdbapi.com/'
    body = json.dumps({
        'Actors': 'Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving',  # noqa
        'Awards': 'Won 4 Oscars. Another 33 wins & 44 nominations.',
        'Country': 'USA',
        'Director': 'Lana Wachowski, Lilly Wachowski',
        'Genre': 'Action, Sci-Fi',
        'Language': 'English',
        'Metascore': '73',
        'Plot': 'A computer hacker learns from mysterious rebels about the true nature of his reality
        'Poster': 'http://ia.media-imdb.com/images/M/MV5BMTkxNDYxOTA4M15BM15BanBnXkFtZTgwNTk0NzQxMTE(
        'Rated': 'R',
        'Released': '31 Mar 1999',
```

```
        'Response': 'True',
        'Runtime': '136 min',
        'Title': 'The Matrix',
        'Type': 'movie',
        'Writer': 'Lilly Wachowski, Lana Wachowski',
        'Year': '1999',
        'imdbID': 'tt0133093',
        'imdbRating': '8.7',
        'imdbVotes': '1,204,431'})

    expected_api_response = {'director': 'Lana Wachowski, Lilly Wachowski',
                             'rated': 'R',
                             'language': 'English',
                             'title': 'The Matrix'}
```

After having all the data in place, we can use our placebo to decorate our test methods like this.

```
class omdbapiTests(TestCase):
    """Omdb api test cases"""

    @GetMovieValidResponse.decorate
    def test_get_movie_valid_response(self):
        movie = get_movie('matrix', 1999)
        self.assertEqual(movie, GetMovieValidResponse.expected_api_response)

    @GetMovieValidResponse.decorate(status=500)
    def test_get_movie_500_response(self):
        movie = get_movie('matrix', 1999)
        self.assertEqual(movie, None)
```

In first method, we directly used the placebo object. In the second method we changed the status of the object to 500 and tested the error case. Notice how logic for mocking the endpoint and test is seperated. We also reused same object for testing the valid response and error case.

As a matter of fact, Placebo object is not only useful for testing. Since main interface is a decorator pattern, you can use it on any callable you want, like views in your web application. That way you can develop your applications against mock data or simulate error cases on your development environment very easily.

CHAPTER 2

Installation

Placebo can be installed using pip

```
$ pip install python-placebo
```

Or source code can be downloaded from github.

In order to use placebo, you should also install a backend of your choice. Currently there are httmock and httpretty backends. We recommend to use httmock if you are only using requests library. Otherwise use httpretty.

```
$ pip install httmock

$ # or

$ pip install httpretty
```

CHAPTER 3

Usage

3.1 Implementation example

(Detailed description will be in Implementing placebo classes section)

Basic usage of placebo can be as follows:

```
class SimplePlacebo(Placebo):
    url = 'http://www.acme.com/items/'
    body = '[{"id": 1}, {"id": 2}, {"id": 3}]'
```

When we decorate a callable with this placebo class, every 'GET' request to http://www.acme.com/items/ url will return 200 response with following body '[{"id": 1}, {"id": 2}, {"id": 3}]'.

We can use this placebo in following test:

```
@SimplePlacebo.decorate
def test_get_list_valid(self):
    api = ItemAPIClient()
    result = api.get_items()
    self.assertEqual(result,
                     [{"id": 1}, {"id": 2}, {"id": 3}])
```

Defaut value for status code is 200 and default value for http method is 'GET'. So we did not need to specify those values in our class. If we wanted to specify all fields, we could do something like this:

```
class SimplePlaceboWithAllFields(Placebo):
    url = 'http://www.acme.com/items/'
    body = '[{"id": 1}, {"id": 2}, {"id": 3}]'
    status = 200
    method = 'GET'
    headers = {'custom-header': 'custom'}
```

In placebo class, "url, body, status, method, headers attributes" can be used to define the mock request. method and url is used to figure out which requests should be mocked. Requests that does not match with given url and methods will go to real backend. "body, status, headers" attributes are used as matching request's content.

There are 2 different ways those attributes can be specified. First, by adding them to Placebo class. Second is update them on decorator. Following tests updates already defined class with diffent status and body.

```
@SimplePlacebo.decorate(status=500)
def test_get_list_error(self):
    api = ItemAPIClient()
    with self.assertRaises(ItemException):
```

```
        api.get_items()

 @SimplePlacebo.decorate(body='invalid-body')
def test_get_list_invalid_body_error(self):
    api = ItemAPIClient()
    with self.assertRaises(ItemException):
        api.get_items()
```

CHAPTER 4

Impementing Placebo classes

4.1 Static placebo classes

A placebo class can have following properties.

```
class SimplePlaceboWithAllFields(Placebo):
    url = 'http://www.acme.com/items/'
    method = 'GET'
    body = '[{"id": 1}, {"id": 2}, {"id": 3}]'
    status = 200
    headers = {'custom-header': 'custom'}

    backend = httprettybackend.get_decorator
```

1. *url*: Url that will be matched to decide if placebo mock is applied. It can be a string, urlparse.ParseResult or urlparse.SplitResult.
2. *method*: HTTP method that will be matched to decide whether placebo mock is applied. It should be a string like GET, POST, PUT, DELETE. Default value for method is GET.
3. *body*: If mock object is applied body will be used as response body. It should be of type string.
4. *status*: If mock object is applied status will be used as http status code of response. It should be an integer like 200, 404 or 500. Default value for status is 200.
5. *headers*: If mock is applied headers will be used as http headers. Type of headers should be dictionary. (Keys should be header names and values should be header values, both strings.)
6. *backend*: Backends provide actual functionality of placebo. Currently there are two different backends that are supported by default – httpretty and httmock. By default httmock is tried. If it cannot be imported, httpretty is tried. Backend is basically a callable that gets a placebo object as argument and returns a decorator that mocks the current apis.

4.2 Dynamic placebo classes

Previous pacebo class has static properties with already defined values. Most of the properties of placebo object can also be defined as methods therefore values can be calculated on the fly. Here is an example placebo object that returns a mock response with id it receives. If id is not an integer, it returns 404 response. Even though those kind of placebo objects are not suitable for tests, they are very usefull for development.

```
class DynamicPlacebo(Placebo):

    backend = httmockbackend

    url_regex = re.compile('^http://www.acme.com/items/(?P<item_id>\d+)/$')

    def url(self):
        return parse.ParseResult(
            scheme='http',
            netloc=r'www\.acme\.com',
            path=r'^/items/(\w+)/$',
            params='',
            query='',
            fragment='')

    def method(self):
        return 'GET'

    def body(self, request_url, request_headers, request_body):
        url = request_url.geturl()
        regex_result = self.url_regex.match(url)
        if regex_result:
            item_id = int(regex_result.groupdict()['item_id'])
            return json.dumps({'id': int(item_id)})
        else:
            return ''

    def headers(self, request_url, request_headers, request_body):
        return {}

    def status(self, request_url, request_headers, request_body):
        """If item_id is not integer return 404."""

        url = request_url.geturl()
        regex_result = self.url_regex.match(url)
        # if item_id is not a number return 404
        if regex_result:
            status = 200
        else:
            status = 404
        return status
```

As seen in the example, almost all the properties of the Placebo object can be written as callables. When we implement those attributes as callables, it might be important to know when those methods are invoked. Some are evaluated for each request, therefore receives request specific informations like url, headers, body. Those attributes are body, headers and status. Rest of the properties are evaluated only once when we apply the decorator (usually on initialization) therefore they do not receive any extra information about the request. Those attributes are url and method. Only property that cannot be implemented as method is `backend`. The reason for that is `backend` is of type callable, so we cannot distinguish backends from callables that return backends. Another way to think about this is some attributes are used to filter requests. So they do not receive any request information. Other are used to form a response.Those attributes gets request information to create their mock responses.

4.3 Placebo properties

This section aims to describe each placebo property in detail.

- *url* : url is used to decide whether current placebo needs to be applied on current request. This property is used only once during initialization time. It can have `str`, `unicode`, `urlparse.ParseResult` or `urlparse.SplitResult` as type. `str` can also be set to a method that returns values of one of the types listed above.
- *method*: method is also used to decide whether current placebo needs to be applied on current request. It is used only once during initialization. It can have `str` or `unicode` types. It can contain one of following values: 'GET', 'POST', 'PUT', 'DELETE',..etc.. Alternatively, `method` can be set to a callable that returns one of the value types above.
- *status*: Status represents http status of response. If placebo is matched with current request, a mock response for that request will be created with the status code of this attribute. This attribute needs to be of type `int` with values like 200, 203, 400, 404, 500, 503 etc. This attribute can also be set to a callable. Since this attribute is used to create a response, this callable will need to get 3 additional attributes that describes the request. Those arguments are request_url, request_header and request_body. (See examples above.)
- *body*: This attribute is used to create the body of the response. It should be of type `str` or `unicode`. It can also be set to a callable which will be called for every request. This callable needs to accept request_url, request_header, request_body arguments.
- *headers*: This attribute is used to create the header of the response. It should be of type `dict` (keys are header names and values are header values, both strings)
- *backend*: This is a meta attribute instead of a filter or response attribute. It is the backend that `Placebo` object will use to create mock objects. Currently there are 2 backends: `backends.httpprettybackend.get_decorator` and `placebo.backends.httmockbackend.get_decorator`. Unlike all the other attributes, `backend` attribute cannot be set to a callable that returns backends. It needs to get a Placebo instance as argument and return a decorator that applies that placebo object. More explanation can be found in the "Implementing backends" section.)

CHAPTER 5

Using placebo classes as decorator

Interface for a placebo class is a decorator. To use a placebo class, decorate method of class is used to decorate a callable.

```
@SimplePlacebo.decorate
def function_to_mock(arg1, arg2):
    ...
```

Placebo decorator can also accepts attributes. All placebo attributes explained above can be provided as an argument to decorator method.

```
@SimplePlacebo.decorate(url='http://www.example.com/api/item',
                        body='response body')
def function_to_mock(arg1, arg2):
    ...
```

Placebo decorator can accept following arguments:

```
@Placebo.decorate(url='http://www.example.com/api/item',
                  body='response body'
                  status=200,
                  method='GET',
                  headers={'custom-header': 'custom_value},
                  backend=httmockbackend.get_decorator,
                  arg_name='item_mock' # arg_name will be explained in getting placebo instance secti
                  )
def function_to_mock(arg1, arg2, item_mock):
    ...
```

CHAPTER 6

Getting a placebo instance

In the placebo interface, Placebo class is used to decorate callables. When we decorate a callable with a placebo class, an object for that class is instantiated and used to decorate current callable. So each decorated callable gets its own placebo instance to hold its mock information. Because object instantiation is handled by Placebo, there is no direct access to actual instance for each callable. But for some edge cases,there is a need to access placebo instances. In those cases arg_name attribute of decorator can be used. If arg_name argument is specified current Placebo instance will be passed to decorated callable as a keyword argument with given name. See "Accessing the last mocked request" section for a usecase.

```
@SimplePlacebo.decorate(arg_name='simple_mock')
def function_to_mock(arg1, arg2, simple_mock):
    ...
```

CHAPTER 7

Accessing the last mocked request

Some times, we might want to access the last mocked request. There are 2 ways to do this: The easiest way is to access last request from the class.

```
@SimplePlacebo.decorate
def function_to_mock(arg1, arg2):
    #
    # Do api call
    #
    items = get_items()
    #
    # Get last request
    #
    last_request = SimplePlacebo.last_request
    #
    # Now we have last request so we can extract
    # last request info from it to use in our tests.
    #
    # Get request url as string
    request_url = last_request.url
    # Get request url as urlparse.ParseResult
    request_parsed_url = last_request.parsed_url
    # Get request body
    request_body = last_request.body
    # Get request headers
    request_headers = last_request.headers
    # Get query parameters
    request_query_params = last_request.query
    ...
```

Accesing the placebo object through the class like this is really easy and does not require any change on rest of the code (*last_request = SimplePlacebo.last_request*). But there is a downside to this aproach: Since last_request here is a class attribute, it is shared by all instances. So, if get_items call fails to do a request, we can still have a last_request attribute on class becuase another test might be using same Placebo class and could already have registered a `last_request` before our test is run.

To solve this problem, you can use the second way of getting the last mocked request – By accessing the `last_requet` attribute of a Placebo instance. That way you can access the `last_request` that is only mocked by the current instance.

To get `last_request` from an instance, first we need to access the instance of Placebo we want to use. Here is an example:

```
@SimplePlacebo.decorate(arg_name='first_page_mock')
@SimplePlacebo.decorate(url='http://www.example.com/api/item?page=2',
                        arg_name='second_page_mock')
def function_to_mock(arg1, arg2,
                     first_page_mock,
                     second_page_mock):
    #
    # Do api call
    #
    items = get_items()
    items = get_items(page=2)
    #
    # Get last request
    #
    last_request = first_page_mock.last_request
    last_request2 = second_page_mock.last_request
    last_request_class = SimplePlacebo.last_request
    #
    # Since last request is request for second page,
    # last_request on class will be the request for second
    # page.
    self.assertIs(last_request2, last_request_class)
    #
    # Now we have last request so we can extract
    # last request info from it to use in our tests.
    #
    # Get request url as string
    request_url = last_request.url
    # Get request url as urlparse.ParseResult
    request_parsed_url = last_request.parsed_url
    # Get request body
    request_body = last_request.body
    # Get request headers
    request_headers = last_request.headers
    # Get query parameters
    request_query_params = last_request.query
    ...
```

Here we used the same decorator for first and second pages. So we needed to access the relevant instance so we could inspect both requests.

CHAPTER 8

Mocking with regex url

For some cases, we might want to mock a url using a regular expression.

Unfortunately each backend has its own way of implementing regular expressions and implementations are incompatible with each other. For that reason, there is no generic way of describing regex urls. In placebo, we choose to delegate regex urls to backends. So each backends has its own version of regex support.

8.1 Regex urls in httmock backend

To use regex url with Httmock, url attribute must be type of `urlparse.ParseResult` or `urlparse.SplitResult`. If the `url` attribute is `str`, `unicode` type or a compiled `regex` instance, regex mock will not work. Here is an example placebo with regex url:

```
from six.moves.urllib import parse

class DynamicPlaceboForHttMock(Placebo):

    url = parse.ParseResult(
            scheme='http',
            netloc=r'www\.acme\.com',
            path=r'^/items/(\d+)/$',
            params='',
            query='',
            fragment='')

    ...
```

In this placebo object, we are we are mocking all urls with format "http://www.acme.com/items/<item_id>/"

(See *tests/placebo_tests.py* file or *examples/* directory for different examples.)

8.2 Regex urls in httpretty backend

Httpretty backend, requires url attribute to be a compiled regex pattern instance. If regex url is of type `urlparse.ParseResult` or `urlparse.SplitResult`, regex will not work. Here is an example url for httpretty.

```
import re

class DynamicPlaceboForHttMock(Placebo):
```

```
    url = re.compile('^http(s)?://www.acme.com/items/\d+/$')
    ...
```

(See *tests/placebo_tests.py* file or *examples/* directory for different examples.)

To summarize, httmock requires regex url to be a ParseResult instance. httpretty on the other hand, requires url attribute to be a compiled regex instance.

CHAPTER 9

Backends

Placebo depends on other 3rd party libraries for mocking functionality. Backends are integration points for placebo to use those libraries. For now placebo integrates with 2 backends: `httmock` and `httpretty`.

Because there are multiple backends, backend libraries are not in placebo's requirements list. At least one of them must be installed explicitly before using placebo.

By default, placebo tries to import httmock backend if it is not successfull (meaning httmock is not installed.). httpretty backend is used. If httpretty is not installed either initialization will fail or placebo will raise an error.

If both libraries are installed and we want to use a non-default backend, the backend attribute of the placebo object can be used to specify which backend to use.

9.1 Implementing a custom backend

A backend is a callable that accepts a placebo instance as an argument and returns a decorator. Placebo object has special methods for backends to consume placebo data. Those methods are:

```
class Placebo(object):
    def _get_url(self):
        ...
    def _get_method(self):
        ...
    def _get_status(self, url, headers, body):
        ...
    def _get_body(self, url, headers, body):
        ...
    def _get_headers(self, url, headers, body):
        ...
```

From those methods, _get_url and _get_method can be invoked on decorator initialization. But rest of the methods must be called for each request. Therefore, they can only be invoked from inside the decorator. (See *placebo/backends/* directory for example implementations.)

CHAPTER 10

Caveats

10.1 Separate url types for different backends

Placebo interface mostly has a backend agnostic interface. You can switch from one backend to another and your mocks should keep working as expected. Unfortunately, placebo interface is broken for regex urls. Each backend expects its urls regexes in different formats. In practice, this is usually not a problem since tests do not usually use regex mocks and switching backends is not a common practice. Also rewriting urls using different formats is not that hard to do. Still, this behaviour is broken by design. So we will try to fix this in the future.

10.2 Httppretty backend problems

httpretty monkey patches the socket interface. Because of that implementation there are some caveats that httpretty brings.

First problem happens when multiple mocks match current request. It that case we want the first applied mock to be chosen. Unfortunately, because of the way httpretty works, mock is being chosen randomly. In different systems different mocks can be chosen. So as a solution, a project that must use httpretty instead of httmock must not apply intersecting mocks. This problem can only appear with heavy use of regex urls.

Last httmock problem is sometimes when pdb is called while httpretty is active, it disables the mocks. This is not a common behavior. That happened to me couple times and I cannot reproduce it. So it is not a reason to use httpretty backend.