
Placebo Documentation

Release 0.0.1

Huseyin Yilmaz

August 13, 2016

1	Introduction	1
1.1	Why this is useful	1
2	Installation	5
3	Usage	7
3.1	Implementation example	7
4	Implementing 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 url in httmock backend	19
8.2	Regex url 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	Httpretty backend problems	23

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 composable 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 re
'Poster': 'http://ia.media-imdb.com/images/M/MV5BMTkxNDYxOTA4M15BM15BanBnXkFtZTgwNTk0NzQ
'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)
```

This tests are fine. But they has 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.

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 functions 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 separated. We also reused same object for testing the valid response and error case.

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

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 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 search httpretty
```


3.1 Implementation example

(Detailed description will be in Implementing placebo classes section)

Basic usage of placebo can be following

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

When we decorate a function 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}])
```

Default 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()
```

Implementing 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 if 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 type of 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 a dictionary. (Keys should be header names and values should be header values.)
6. *backend*: Backends provides actual functionality of placebo. Currently there are two different backends are supported by default. `httpretty` and `httmock`. By default `httmock` is tried if it cannot be imported `httpretty` is tried. *backend* is basically a function that gets a placebo object as argument and mocks the current apis.

4.2 Dynamic placebo classes

Previous placebo 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 objects 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 useful 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 Placebo object can be written as methods. Some properties are evaluated for each request therefore receives url, headers, body. Rest of the properties are evaluated only once on initialization therefore does not receive any extra information about the request. Only property that cannot be implemented as method is backend. The reason for that is backend has a type of function so we cannot distinguish backends from methods that returns backends.

4.3 Placebo properties

This section aims to describe each placebo properties in detail.

- *url* : url is used to decide if current placebo needs to be applied on current request. This property is used only once in initialization time. It can have *str*, *unicode*, *urlparse.ParseResult* or *urlparse.SplitResult* type. *str* can also be implemented as a method and method should return one of the types listed above.
- *method*: method is also used to decide if current placebo needs to be applied on current request. It is used only once in initialization. It can have *str* or *unicode* types. It can have one of following values: 'GET', 'POST', 'PUT', 'DELETE', ..etc.. alternatively, method can be implemented as a method that returns one of the values 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 status code of this attribute. This attribute needs to be type of *int* with values like 200, 203, 400, 404, 500, 503 etc. This attribute can also be implemented as a method. Since this attribute is used to create a response, method version of status will be getting 3 additional attribute that describes request. those arguments are *request_url*, *request_header*, *request_body*. (See examples about for usage example.)
- *body*: body attribute is used to create body of response. It should have type of *str* or *unicode*. It can also be implemented as a method. body method will be called for every request. So, method version gets *request_url*, *request_header*, *request_body* arguments.
- *headers*: headers attribute is used as a header of response. It should be type of dict (keys are header names and values are header values.)
- *backend*: Backend property is a meta property instead of a filter or response property. It is a backend that Placebo object is used to create mock objects. Currently there are 2 backends. Those are `backends.httprettybackend.get_decorator` and `placebo.backends.httmockbackend.get_decorator`. backend attribute cannot be implemented as a method. Type of backend is a function. (function that gets a Placebo instance as argument and returns a decorator that applies that placebo object. More explanation can be found in Implementing backends section.)

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 function.

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

Placebo decorator can also accepts attributes. All placebo attributes explains above as argument.

```
@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 sect.
                  )
def function_to_mock(arg1, arg2, item_mock):
    ...
```

Getting a placebo instance

In placebo interface, Placebo class is used to decorate functions. When we decorate a function with a placebo class, an object for that class is instantiated and used to decorate current function. So each decorated function gets its own object to hold its mock information. Because object instantiation is handled by Placebo, there is no direct access to actual instance for each function. But for some edge cases, there is a need to access placebo objects. 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 function as a keyword argument with given name. See *Accessing the last mocked request* section for usecase.

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

Accessing the last mocked request

Some times, we might want to access to last mocked request. There is 2 ways to do this. We can access last request that is located on 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
    ...
```

Accessing placebo object through class is really easy and does not require any change on rest of our 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 and already register a last request before our test is run. To solve this problem Placebo instances also have a *last_request*. That way you can access *last_request* only mocked by current instance.

To get *last_request* from instance, first we need access to 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 same Decorator for first and second page. So we needed to access the instance for them so we could inspect both requests.

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 expression and each implementation is incompatible with the others. 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 implementation.

8.1 Regex url in httmock backend

Httmock requires url attribute to have type of *urlparse.ParseResult* or *urlparse.SplitResult* to use backends. If url attribute is in String type of regex, mock will not work.

```
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 url in httpretty backend

Httpretty backend, requires url attribute to have type of regex pattern. If regex url has type of *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.)

Backends

Placebo depends on other 3rd libraries for mocking functionality. Backends are integration points for placebo to use those libraries. For now placebo comes with 2 backends. Those are httmock backend and httpretty backend.

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

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

If both libraries are installed and we want to use different backend, backend attribute of placebo can be used to specify which backends to use.

9.1 Implementing a custom backend

Backend is a function 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. There for they can only be invoked from inside the decorator. (See *placebo/backends/* directory for example implementations.)

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 keep working as expected. Unfortunately, placebo interface is broken for regex urls. Each backend expects its regex urls in different type. 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 in different type is not really a hard task. 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 sockets. Because of that implementation there are some caveats that httpretty brings.

First problem happens when multiple mocks matches current request. In that case we want first applied mock to be chosen. Unfortunately, because of the way httpretty works, mock 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 raise with heavy use of regex urls.

Last httmock problem is sometimes pdb calls while httpretty is active, 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.