Requests Mock Documentation

Release 1.11.0

Jamie Lennox

Jun 08, 2023

Contents

Contents:

CHAPTER 1

Overview

The requests library has the concept of pluggable transport adapters. These adapters allow you to register your own handlers for different URIs or protocols.

The *requests-mock* library at its core is simply a transport adapter that can be preloaded with responses that are returned if certain URIs are requested. This is particularly useful in unit tests where you want to return known responses from HTTP requests without making actual calls.

As the requests library has very limited options for how to load and use adapters *requests-mock* also provides a number of ways to make sure the mock adapter is used. These are only loading mechanisms, they do not contain any logic and can be used as a reference to load the adapter in whatever ways works best for your project.

1.1 Installation

```
pip install requests-mock
```

CHAPTER 2

Using the Mocker

The mocker is a loading mechanism to ensure the adapter is correctly in place to intercept calls from requests. Its goal is to provide an interface that is as close to the real requests library interface as possible.

`requests_mock.Mocker` takes optional parameters:

real_http (bool) If `True` then any requests that are not handled by the mocking adapter will be forwarded to the real server (see *Real HTTP Requests*), or the containing Mocker if applicable (see *Nested Mockers*). Defaults to `False`.

json_encoder (json.JSONEncoder) If set uses the provided json encoder for all JSON responses compiled as part of the mocker.

session (requests.Session) If set, only the given session instance is mocked (see *Mocking specific sessions*).

2.1 Activation

Loading of the Adapter is handled by the `requests_mock.Mocker` class, which provides two ways to load an adapter:

2.1.1 Context Manager

The Mocker object can work as a context manager.

```
>>> import requests
>>> import requests_mock

>>> with requests_mock.Mocker() as m:
...     m.get('http://test.com', text='resp')
...     requests.get('http://test.com').text
...
'resp'
```

2.1.2 Decorator

Mocker can also be used as a decorator. The created object will then be passed as the last positional argument.

```
>>> @requests_mock.Mocker()
... def test_function(m):
...     m.get('http://test.com', text='resp')
...     return requests.get('http://test.com').text
...
>>> test_function()
'resp'
```

If the position of the mock is likely to conflict with other arguments you can pass the *kw* argument to the Mocker to have the mocker object passed as that keyword argument instead.

```
>>> @requests_mock.Mocker(kw='mock')
... def test_kw_function(**kwargs):
...     kwargs['mock'].get('http://test.com', text='resp')
...     return requests.get('http://test.com').text
...
>>> test_kw_function()
'resp'
```

2.1.3 Contrib

The contrib module also provides ways of loading the mocker based on other frameworks. These will require additional dependencies but may provide a better experience depending on your tests setup.

See *Additional Loading* for these additions.

2.2 Class Decorator

Mocker can also be used to decorate a whole class. It works exactly like in case of decorating a normal function. When used in this way they wrap every test method on the class. The mocker recognise methods that start with *test* as being test methods. This is the same way that the *unittest.TestLoader* finds test methods by default. It is possible that you want to use a different prefix for your tests. You can inform the mocker of the different prefix by setting *requests_mock.Mocker.TEST_PREFIX*:

```
>>> requests_mock.Mocker.TEST_PREFIX = 'foo'
>>>
>>> @requests_mock.Mocker()
... class Thing(object):
...     def foo_one(self, m):
...         m.register_uri('GET', 'http://test.com', text='resp')
...         return requests.get('http://test.com').text
...     def foo_two(self, m):
...         m.register_uri('GET', 'http://test.com', text='resp')
...         return requests.get('http://test.com').text
...
>>>
>>> Thing().foo_one()
'resp'
>>> Thing().foo_two()
'resp'
```

This behavior mimics how patchers from *mock* library works.

2.3 Methods

The mocker object can be used with a similar interface to requests itself.

```
>>> with requests_mock.Mocker() as mock:
...     mock.get('http://test.com', text='resp')
...     requests.get('http://test.com').text
...
'resp'
```

The following functions exist for the common HTTP methods:

- `delete()`
- `get()`
- `head()`
- `options()`
- `patch()`
- `post()`
- `put()`

As well as the basic:

- `request()`
- `register_uri()`

These methods correspond to the HTTP method of your request, so to mock POST requests you would use the `post()` function. Further information about what can be matched from a request can be found at *Request Matching*

2.4 Real HTTP Requests

If `real_http` is set to `True` then any requests that are not handled by the mocking adapter will be forwarded to the real server, or the containing Mocker if applicable (see *Nested Mockers*).

```
>>> with requests_mock.Mocker(real_http=True) as m:
...     m.register_uri('GET', 'http://test.com', text='resp')
...     print(requests.get('http://test.com').text)
...     print(requests.get('http://www.google.com').status_code)
...
'resp'
200
```

New in 1.1

Similarly when using a mocker you can register an individual URI to bypass the mocking infrastructure and make a real request. Note this only works when using the mocker and not when directly mounting an adapter.

```
>>> with requests_mock.Mocker() as m:
...     m.register_uri('GET', 'http://test.com', text='resp')
...     m.register_uri('GET', 'http://www.google.com', real_http=True)
...     print(requests.get('http://test.com').text)
...     print(requests.get('http://www.google.com').status_code)
...
'resp'
200
```

2.5 JSON Encoder

In python's json module you can customize the way data is encoded by subclassing the `JSONEncoder` object and passing it to encode. A common example of this might be to use *DjangoJSONEncoder <https://docs.djangoproject.com/en/3.2/topics/serialization/#djangojsonencoder>* for responses.

You can specify this encoder object either when creating the `requests_mock.Mocker` or individually at the mock creation time.

```
>>> import django.core.serializers.json.DjangoJSONEncoder as DjangoJSONEncoder
>>> with requests_mock.Mocker(json_encoder=DjangoJSONEncoder) as m:
...     m.register_uri('GET', 'http://test.com', json={'hello': 'world'})
...     print(requests.get('http://test.com').text)
```

or

```
>>> import django.core.serializers.json.DjangoJSONEncoder as DjangoJSONEncoder
>>> with requests_mock.Mocker() as m:
...     m.register_uri('GET', 'http://test.com', json={'hello': 'world'}, json_
↪encoder=DjangoJSONEncoder)
...     print(requests.get('http://test.com').text)
```

2.6 Nested Mockers

New in 1.8

When nesting mockers the innermost Mocker replaces all others. If `real_http` is set to `True`, at creation or for a given resource, the request is passed to the containing Mocker. The containing Mocker can in turn:

- serve the request;
- raise `NoMockAddress`;
- or pass the request to yet another Mocker (or to the unmocked `requests.Session`) if `real_http` is set to `True`.

```
>>> url = "https://www.example.com/"
>>> with requests_mock.Mocker() as outer_mock:
...     outer_mock.get(url, text='outer')
...     with requests_mock.Mocker(real_http=True) as middle_mock:
...         with requests_mock.Mocker() as inner_mock:
...             inner_mock.get(url, real_http=True)
...             print(requests.get(url).text)
...
'outer'
```

Most of the time nesting can be avoided by making the mocker object available to subclasses/subfunctions.

Warning: When starting/stopping mockers manually, make sure to stop innermost mockers first. A call from an active inner mocker with a stopped outer mocker leads to undefined behavior.

2.7 Mocking specific sessions

New in 1.8

`requests_mock.Mocker` can be used to mock specific sessions through the `session` parameter.

```
>>> url = "https://www.example.com/"
>>> with requests_mock.Mocker() as global_mock:
...     global_mock.get(url, text='global')
...     session = requests.Session()
...     print("requests.get before session mock:", requests.get(url).text)
...     print("session.get before session mock:", session.get(url).text)
...     with requests_mock.Mocker(session=session) as session_mock:
...         session_mock.get(url, text='session')
...         print("Within session mock:", requests.get(url).text)
...         print("Within session mock:", session.get(url).text)
...     print("After session mock:", requests.get(url).text)
...     print("After session mock:", session.get(url).text)
...
'requests.get before session mock: global'
'session.get before session mock: global'
'requests.get within session mock: global'
'session.get within session mock: session'
'requests.get after session mock: global'
'session.get after session mock: global'
```

Note: As an alternative, `requests_mock.Adapter` instances can be mounted on specific sessions (see *Adapter Usage*).

CHAPTER 3

Request Matching

Whilst it is preferable to provide the whole URI to `requests_mock.Adapter.register_uri()` it is possible to just specify components.

The examples in this file are loaded with:

```
>>> import requests
>>> import requests_mock
>>> adapter = requests_mock.Adapter()
>>> session = requests.Session()
>>> session.mount('mock://', adapter)
```

Note: The examples within use this syntax because request matching is a function of the adapter and not the mocker. All the same arguments can be provided to the mocker if that is how you use *requests_mock* within your project, and use the

```
mock.get(url, ...)
```

form in place of the given:

```
adapter.register_uri('GET', url, ...)
```

If you are not familiar with requests' adapters (see *Adapter Usage*), prefer the mocker approach (see *Using the Mocker*).

Note: By default all matching is case insensitive. This can be adjusted by passing case_sensitive=True when creating a mocker or adapter, or globally by doing:

```
requests_mock.mock.case_sensitive = True
```

for more, see: *Case Insensitivity*

3.1 Simple

The most simple way to match a request is to register the URL and method that will be requested with a textual response. When a request is made that goes through the mocker this response will be retrieved.

```
.. >>> adapter.register_uri('GET', 'mock://test.com/path', text='resp')
.. >>> session.get('mock://test.com/path').text
.. 'resp'
```

3.2 Path Matching

You can specify a protocol-less path:

```
.. >>> adapter.register_uri('GET', '//test.com/', text='resp')
.. >>> session.get('mock://test.com/').text
.. 'resp'
```

or you can specify just a path:

```
.. >>> adapter.register_uri('GET', '/path', text='resp')
.. >>> session.get('mock://test.com/path').text
.. 'resp'
.. >>> session.get('mock://another.com/path').text
.. 'resp'
```

3.3 Query Strings

Query strings provided to a register will match so long as at least those provided form part of the request.

```
>>> adapter.register_uri('GET', '/?a=1', text='resp')
>>> session.get('mock://test.com/?a=1&b=2').text
'resp'
```

We can also match an empty query string.

```
>>> adapter.register_uri('GET', '/?a', text='resp')
>>> session.get('mock://test.com/?a').text
'resp'
```

If any part of the query string is wrong then it will not match.

```
>>> session.get('mock://test.com/?a=3')
Traceback (most recent call last):
   ...
requests_mock.exceptions.NoMockAddress: No mock address: GET mock://test.com/?a=3
```

This can be a problem in certain situations, so if you wish to match only the complete query string there is a flag *complete_qs*:

```
>>> adapter.register_uri('GET', '/8?a=1', complete_qs=True, text='resp')
>>> session.get('mock://test.com/8?a=1&b=2')
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
   ...
requests_mock.exceptions.NoMockAddress: No mock address: GET mock://test.com/8?a=1&b=2
```

3.4 Matching ANY

There is a special symbol at *requests_mock.ANY* which acts as the wildcard to match anything. It can be used as a replace for the method and/or the URL.

```
>>> adapter.register_uri(requests_mock.ANY, 'mock://test.com/8', text='resp')
>>> session.get('mock://test.com/8').text
'resp'
>>> session.post('mock://test.com/8').text
'resp'
```

```
>>> adapter.register_uri(requests_mock.ANY, requests_mock.ANY, text='resp')
>>> session.get('mock://whatever/you/like').text
'resp'
>>> session.post('mock://whatever/you/like').text
'resp'
```

3.5 Regular Expressions

URLs can be specified with a regular expression using the python `re` module. To use this you should pass an object created by `re.compile()`.

The URL is then matched using `re.regex.search()` which means that it will match any component of the url, so if you want to match the start of a URL you will have to anchor it.

```
.. >>> import re
.. >>> matcher = re.compile('tester.com/a')
.. >>> adapter.register_uri('GET', matcher, text='resp')
.. >>> session.get('mock://www.tester.com/a/b').text
.. 'resp'
```

If you use regular expression matching then *requests-mock* can't do its normal query string or path-only matching. That will need to be part of the expression.

3.6 Request Headers

A dictionary of headers can be supplied such that the request will only match if the available headers also match. Only the headers that are provided need match, any additional headers will be ignored.

```
>>> adapter.register_uri('POST', 'mock://test.com/headers', request_headers={'key':
↪'val'}, text='resp')
>>> session.post('mock://test.com/headers', headers={'key': 'val', 'another': 'header
↪'}).text
'resp'
>>> resp = session.post('mock://test.com/headers')
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
   ...
requests_mock.exceptions.NoMockAddress: No mock address: POST mock://test.com/headers
```

3.7 Additional Matchers

As distinct from *Custom Matching* below, we can add an additional matcher callback that lets us do more dynamic matching in addition to the standard options. This is handled by a callback function that takes the request as a parameter:

```
>>> def match_request_text(request):
...     # request.text may be None, or '' prevents a TypeError.
...     return 'hello' in (request.text or '')
...
>>> adapter.register_uri('POST', 'mock://test.com/additional', additional_
↪matcher=match_request_text, text='resp')
>>> session.post('mock://test.com/additional', data='hello world').text
'resp'
>>> resp = session.post('mock://test.com/additional', data='goodbye world')
Traceback (most recent call last):
   ...
requests_mock.exceptions.NoMockAddress: No mock address: POST mock://test.com/
↪additional
```

Using this mechanism lets you do custom handling such as parsing yaml or XML structures and matching on features of that data or anything else that is not directly handled via the provided matchers rather than build in every possible option to *requests_mock*.

3.8 Custom Matching

Internally, calling `register_uri()` creates a *matcher* object for you and adds it to the list of matchers to check against.

A *matcher* is any callable that takes a `requests.Request` and returns a `requests.Response` on a successful match or `None` if it does not handle the request.

If you need more flexibility than provided by `register_uri()` then you can add your own *matcher* to the `Adapter`. Custom *matchers* can be used in conjunction with the inbuilt *matchers*. If a matcher returns `None` then the request will be passed to the next *matcher* as with using `register_uri()`.

```
>>> def custom_matcher(request):
...     if request.path_url == '/test':
...         resp = requests.Response()
...         resp.status_code = 200
...         return resp
...     return None
...
>>> adapter.add_matcher(custom_matcher)
>>> session.get('mock://test.com/test').status_code
200
>>> session.get('mock://test.com/other')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
requests_mock.exceptions.NoMockAddress: No mock address: POST mock://test.com/other
```

CHAPTER 4

Creating Responses

Note: The examples within use this syntax because response creation is a function of the adapter and not the mocker. All the same arguments can be provided to the mocker if that is how you use *requests_mock* within your project, and use the

```
mock.get(url, ...)
```

form in place of the given:

```
adapter.register_uri('GET', url, ...)
```

If you are not familiar with requests' adapters (see *Adapter Usage*), prefer the mocker approach (see *Using the Mocker*).

4.1 Registering Responses

Responses are registered with the `requests_mock.Adapter.register_uri()` function on the adapter.

```
>>> adapter.register_uri('GET', 'mock://test.com', text='Success')
>>> resp = session.get('mock://test.com')
>>> resp.text
'Success'
```

`register_uri()` takes the HTTP method, the URI and then information that is used to build the response. This information includes:

status_code The HTTP status response to return. Defaults to 200.

reason The reason text that accompanies the Status (e.g. 'OK' in '200 OK')

headers A dictionary of headers to be included in the response.

cookies A CookieJar containing all the cookies to add to the response.

To specify the body of the response there are a number of options that depend on the format that you wish to return.

json A python object that will be converted to a JSON string.

text A unicode string. This is typically what you will want to use for regular textual content.

content A byte string. This should be used for including binary data in responses.

body A file like object that contains a *.read()* function.

raw A prepopulated `urllib3.response.HTTPResponse` to be returned.

exc An exception that will be raised instead of returning a response.

These options are named to coincide with the parameters on a `requests.Response` object. For example:

```
>>> adapter.register_uri('GET', 'mock://test.com/1', json={'a': 'b'}, status_code=200)
>>> resp = session.get('mock://test.com/1')
>>> resp.json()
{'a': 'b'}

>>> adapter.register_uri('GET', 'mock://test.com/2', text='Not Found', status_
↪code=404)
>>> resp = session.get('mock://test.com/2')
>>> resp.text
'Not Found'
>>> resp.status_code
404
```

It only makes sense to provide at most one body element per response.

4.2 Dynamic Response

A callback can be provided in place of any of the body elements. Callbacks must be a function in the form of

```
def callback(request, context):
```

and return a value suitable to the body element that was specified. The elements provided are:

request The `requests.Request` object that was provided.

context An object containing the collected known data about this response.

The available properties on the *context* are:

headers The dictionary of headers that are to be returned in the response.

status_code The status code that is to be returned in the response.

reason The string HTTP status code reason that is to be returned in the response.

cookies A `requests_mock.CookieJar` of cookies that will be merged into the response.

These parameters are populated initially from the variables provided to the `register_uri()` function and if they are modified on the context object then those changes will be reflected in the response.

```
>>> def text_callback(request, context):
...     context.status_code = 200
...     context.headers['Test1'] = 'value1'
...     return 'response'
...
```

(continues on next page)

(continued from previous page)

```
>>> adapter.register_uri('GET',
...                      'mock://test.com/3',
...                      text=text_callback,
...                      headers={'Test2': 'value2'},
...                      status_code=400)
>>> resp = session.get('mock://test.com/3')
>>> resp.status_code, resp.headers, resp.text
(200, {'Test1': 'value1', 'Test2': 'value2'}, 'response')
```

4.3 Response Lists

Multiple responses can be provided to be returned in order by specifying the keyword parameters in a list. If the list is exhausted then the last response will continue to be returned.

```
>>> adapter.register_uri('GET', 'mock://test.com/4', [{'text': 'resp1', 'status_code
↪': 300},
...                                                {'text': 'resp2', 'status_code
↪': 200}])
>>> resp = session.get('mock://test.com/4')
>>> (resp.status_code, resp.text)
(300, 'resp1')
>>> resp = session.get('mock://test.com/4')
>>> (resp.status_code, resp.text)
(200, 'resp2')
>>> resp = session.get('mock://test.com/4')
>>> (resp.status_code, resp.text)
(200, 'resp2')
```

Callbacks work within response lists in exactly the same way they do normally;

```
>>> adapter.register_uri('GET', 'mock://test.com/5', [{'text': text_callback}]),
>>> resp = session.get('mock://test.com/5')
>>> resp.status_code, resp.headers, resp.text
(200, {'Test1': 'value1', 'Test2': 'value2'}, 'response')
```

4.4 Raising Exceptions

When trying to emulate a connection timeout or SSLError you need to be able to throw an exception when a mock is hit. This can be achieved by passing the *exc* parameter instead of a body parameter.

```
>>> adapter.register_uri('GET', 'mock://test.com/6', exc=requests.exceptions.
↪ConnectTimeout),
>>> session.get('mock://test.com/6')
Traceback (most recent call last):
   ...
ConnectTimeout:
```

4.5 Handling Cookies

Whilst cookies are just headers they are treated in a different way, both in HTTP and the requests library. To work as closely to the requests library as possible there are two ways to provide cookies to requests_mock responses.

The most simple method is to use a dictionary interface. The Key and value of the dictionary are turned directly into the name and value of the cookie. This method does not allow you to set any of the more advanced cookie parameters like expiry or domain.

```
>>> adapter.register_uri('GET', 'mock://test.com/7', cookies={'foo': 'bar'}),
>>> resp = session.get('mock://test.com/7')
>>> resp.cookies['foo']
'bar'
```

The more advanced way is to construct and populate a cookie jar that you can add cookies to and pass that to the mocker.

```
>>> jar = requests_mock.CookieJar()
>>> jar.set('foo', 'bar', domain='.test.com', path='/baz')
>>> adapter.register_uri('GET', 'mock://test.com/8', cookies=jar),
>>> resp = session.get('mock://test.com/8')
>>> resp.cookies['foo']
'bar'
>>> resp.cookies.list_paths()
['/baz']
```

CHAPTER 5

Known Issues

5.1 Case Insensitivity

By default matching is done in a completely case insensitive way. This makes sense for the protocol and host components which are defined as insensitive by RFCs however it does not make sense for path.

A byproduct of this is that when using request history the values for path, qs etc are all lowercased as this was what was used to do the matching.

To work around this when building an Adapter or Mocker you do

```
with requests_mock.mock(case_sensitive=True) as m:
    ...
```

or you can override the default globally by

```
requests_mock.mock.case_sensitive = True
```

It is recommended to run the global fix as it is intended that case sensitivity will become the default in future releases.

Note that even with case_sensitive enabled the protocol and netloc of a mock are still matched in a case insensitive way.

5.2 Cookies in Sessions

If a mocked response sets a cookie, the cookie will be accessible through the Response object as expected.

If, however, the mocked response is requested through a Session, the cookie will not reach the Session instance.

The problem is that the Requests library adds cookies to Session objects by reading the `httplib` response object, not the `requests` Response object. Mock responses naturally don't use `httplib`.

To fix this issue, the *requests-mock* library needs to convert all acceptable forms of mocked cookies into `Set-Cookie` headers to enable the Requests library to properly process them.

This issue is being tracked on GitHub:

https://github.com/jamielennox/requests-mock/issues/17

CHAPTER 6

Request History

The object returned from creating a mock or registering a URI in an adapter is capable of tracking and querying the history of requests that this mock responded to.

6.1 Called

The easiest way to test if a request hit the adapter is to simply check the `called` property or the `call_count` property.

```
>>> import requests
>>> import requests_mock

>>> with requests_mock.mock() as m:
...     m.get('http://test.com', text='resp')
...     resp = requests.get('http://test.com')
...
>>> m.called
True
>>> m.call_count
1
```

6.2 Request Objects

The history of objects that passed through the *mocker/adapter* can also be retrieved.

```
>>> history = m.request_history
>>> len(history)
1
>>> history[0].method
'GET'
```

(continues on next page)

(continued from previous page)

```
>>> history[0].url
'http://test.com/'
```

The alias *last_request* is also available for the last request to go through the mocker.

This request object is a wrapper around a standard `requests.Request` object with some additional information that make the interface more workable (as users do not generally deal with the `Request` object).

These additions include:

text The data of the request converted into a unicode string.

json The data of the request loaded from json into python objects.

qs The query string of the request. See `urllib.parse.parse_qs()` for information on the return format.

hostname The host name that the request was sent to.

port The port the request was sent to.

```
>>> m.last_request.scheme
'http'
>>> m.last_request.hostname
'test.com'
```

The following parameters of the `requests.request()` call are also exposed via the request object:

timeout How long to wait for the server to send data before giving up.

allow_redirects Set to `True` if POST/PUT/DELETE redirect following is allowed.

proxies Dictionary mapping protocol to the URL of the proxy.

verify Whether the SSL certificate will be verified.

cert The client certificate or cert/key tuple for this request.

Note: That the default values of these attributes are the values that are passed to the adapter and not what is passed to the request method. This means that the default for `allow_redirects` is `None` (even though that is interpretted as `True`) if unset, whereas the default for verify is `True`, and the default for proxies is the empty dict.

6.3 Reset History

For mocks, adapters, and matchers, the history can be reset. This can be useful when testing complex code with multiple requests.

For mocks, use the `reset_mock()` method.

```
>>> m.called
True
>>> m.reset_mock()
>>> m.called
False
>>> m.call_count
0
```

For adapters and matchers, there is a `reset()` method. Resetting the adapter also resets the associated matchers.

```
>>> adapter = requests_mock.adapter.Adapter()
>>> matcher = adapter.register_uri('GET', 'mock://test.com', text='resp')
>>> session = requests.Session()
>>> session.mount('mock://', adapter)
>>> session.get('mock://test.com')
>>> adapter.called
True
>>> adapter.reset()
>>> adapter.called
False
>>> matcher.called  # Reset adapter also resets associated matchers
False
```

However, resetting the matcher does not reset the adapter.

```
>>> session.get('mock://test.com')
>>> matcher.called
True
>>> matcher.reset()
>>> matcher.called
False
>>> adapter.called  # Reset matcher does not reset adapter
True
```

CHAPTER 7

Adapter Usage

7.1 Creating an Adapter

The standard requests means of using a transport adapter is to mount it on a created session. This is not the only way to load the adapter, however the same interactions will be used.

When mounting an adapter, keep in mind that:

- for a given URL, requests will use the adapter associated to the longest matching prefix;
- session are created with adapters for the `http://` and `https://` prefixes (and thus adapters mounted on `http` or `https` will never be used);
- requests only prepares URLs for *http* schemes (start with http and only contains letters, numbers, and the + and - signs). In particular `params` won't work with the `mock://` scheme, but will with `http+mock://`.

If you are not familiar with adapters, prefer the mocker approach (see *Using the Mocker*).

```
>>> import requests
>>> import requests_mock

>>> session = requests.Session()
>>> adapter = requests_mock.Adapter()
>>> session.mount('mock://', adapter)
```

At this point any requests made by the session to a URI starting with *mock://* will be sent to our adapter.

CHAPTER 8

Additional Loading

Common additional loading mechanism are supported in the `requests_mock.contrib` module.

These modules may require dependencies outside of what is provided by *requests_mock* and so must be provided by the including application.

8.1 Fixtures

Fixtures provide a way to create reusable state and helper methods in test cases.

To use the *requests-mock* fixture your tests need to have a dependency on the fixtures library. This can be optionally installed when you install *requests-mock* by doing:

```
pip install requests-mock[fixture]
```

The fixture mocks the `requests.Session.get_adapter()` method so that all requests will be served by the mock adapter.

The fixture provides the same interfaces as the adapter.

```
>>> import requests
>>> from requests_mock.contrib import fixture
>>> import testtools

>>> class MyTestCase(testtools.TestCase):
...
...     TEST_URL = 'http://www.google.com'
...
...     def setUp(self):
...         super(MyTestCase, self).setUp()
...         self.requests_mock = self.useFixture(fixture.Fixture())
...         self.requests_mock.register_uri('GET', self.TEST_URL, text='respA')
...
...     def test_method(self):
```

(continues on next page)

(continued from previous page)

```
...         self.requests_mock.register_uri('POST', self.TEST_URL, text='respB')
...         resp = requests.get(self.TEST_URL)
...         self.assertEqual('respA', resp.text)
...         self.assertEqual(self.TEST_URL, self.requests_mock.last_request.url)
...
```

8.2 pytest

pytest has its own method of registering and loading custom fixtures. *requests-mock* provides an external fixture registered with pytest such that it is usable simply by specifying it as a parameter. There is no need to import *requests-mock* it simply needs to be installed and specify the argument *requests_mock*.

The fixture then provides the same interface as the `requests_mock.Mocker` letting you use *requests-mock* as you would expect.

```
>>> import pytest
>>> import requests

>>> def test_url(requests_mock):
...     requests_mock.get('http://test.com', text='data')
...     assert 'data' == requests.get('http://test.com').text
...
```

If you are unfamiliar with how pytest decorators work then please *read the fixture documentation* first as it means that you should no longer use the *@requests_mock.Mocker* syntax that is present in the documentation examples. This confusion between how unittest and pytest work is the biggest source of complaint and is not a *requests-mock* inherent problem.

8.2.1 Configuration

Some options are available to be read from pytest's configuration mechanism.

These options are:

requests_mock_case_sensitive: (bool) Turn on case sensitivity in path matching.

8.2.2 Background

This section was initially copied from StackOverflow

pytest doesn't play along with function decorators that add positional arguments to the test function. pytest considers all arguments that:

- aren't bound to an instance or type as in instance or class methods;
- don't have default values;
- aren't bound with functools.partial;
- aren't replaced with unittest.mock mocks

to be replaced with fixture values, and will fail if it doesn't find a suitable fixture for any argument. So stuff like

```
import functools
import pytest


def deco(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        args += ('spam',)
        return func(*args, **kwargs)
    return wrapper


@deco
def test_spam(spam_arg):
    assert True
```

will fail, and this is exactly what requests-mock does. A workaround to that would be passing the mocker via keyword args:

```
import pytest
import requests_mock


@requests_mock.Mocker(kw='m')
def test_with_mock_and_fixtures(capsys, **kwargs):
    m = kwargs['m']
    ...
```

however at this point it would simply be easier to use the provided pytest decorator.

CHAPTER 9

Release Notes

9.1 Release Notes

9.1.1 1.11.0

New Features

- Exposes some public type aliases (for type hinting only, they can't be instanciated) for the types intended to be used by *requests_mock* users. The following types are now exposed: - *requests_mock.Context* used in callbacks - *requests_mock.Request* used in callbacks, which is a *requests.PreparedRequest* proxy. - *requests_mock.Callback[T]* which is the callbacks type.

Bug Fixes

- Some typing inconsistencies have been fixed. Especially for *request* object in signatures which is in fact a *requests_mock.Request* object.
- Fix incompatibility with urllib3 >2.0.0. In 2.0.0 they default to enforcing content length checking on returned bodies in responses from the previous default of false. However the flag is still available so for compatibility we can just default the other way.

9.1.2 1.10.0

New Features

- You can now set the JSON encoder for use by the json= parameter on either the mocker or an individual mocked response. This will make it easier to work with systems that encode in a specific way.

Bug Fixes

- In some multithreading situations the mocker might try and replace an already patched object from a different thread. To make this thread safe we will put a lock around the mocker so that only one thread can work with the mocker at any time. This has to be a reentrant lock as the same thread can go through it multiple times.
- A series of type hint fixes, most importantly the register_uri return value is not a response but a usable Matcher object.

9.1.3 1.9.3

Bug Fixes

- Improvements to some type annotations that were making it confusing to use the library in an IDE.

9.1.4 1.9.2

Bug Fixes

- As part of 1.9.0 we started quoting unsafe URL characters. This was done incorrectly that meant we requoted existing quoted strings. Fixes

9.1.5 1.9.1

Bug Fixes

- A py.typed file is required to enable type annotations for the package. Add to pack for re-release.

9.1.6 1.9.0

New Features

- Add python type hints for python 3.

Bug Fixes

- Issue 144 - When performing a read we handled an empty byte return as an indication to close the file pointer. This is not true when you actually ask for a zero byte read so we should allow it.
- Fix [#148](https://github.com/jamielennox/requests-mock/issues/158). When you have a non url-safe character in your URL it is quoted by requests however requests-mock just treated it as a normal string.

9.1.7 1.8.0

Prelude

Makes explicit the behaviour of nested mocking.

New Features

- Add a reset/reset_mock function to the mocker and components so that users can clear the calls and history from the Mocker without clearing all the matching that has been used.
- Nested mocks now has a defined behaviour which is essentially a classic nesting behaviour. A mock works as normal, but in the same way that real_http=True will escalate to the outer send (typically the real send) if there is a mocker defined outside that will handle the calls.
- Allow mocking of only a single Session object. While there has always been the option of adding just the Adapter to an existing Session, this is more difficult to work with than the Mocker workflow that is typically used. You can now pass a Session object to a Mocker and it will provide the standard workflow but limited to just that object.
- When creating a response we now default the reason string to the appropriate string based on the status code. This can still be provided manually.
- Make real_http variable on Mocker object public. This means you can change the value of the parameter after the object has been created. This will allow setting it in situations like pytest where you may not be able to set __init__ time options.

Bug Fixes

- This fixes all known nesting bugs. They may not be perfect, however this is now the explicit defined behaviour and so all related bugs are closed in favour of this.
- 124 when using *response.iter_content(None)* the iterator would keep returning b" and never finish. In most code paths the b" would indicate that the stream is finished, but using None we have to close it ourselves.

9.1.8 1.7.0

New Features

- You can now match on the empty query string value like */path?a*.

Bug Fixes

- Pins version of requests to <3 to prepare for new release of requests in future.

9.1.9 1.6.0

Prelude

Increase the minimum required requests version to 2.3

Critical Issues

- The minimum version of requests has been increase to 2.3. This simply ensures that all documented features of requests-mock are actually available. This version of requests is still quite old and if this is an issue you should either pin requests-mock to <1.6 or preferably update requests.

Bug Fixes

- Remove weakref objects from the request/response that will allow the objects to be pickled with the regular python mechanisms.
- If you specified a charset in the Content-Type of a response it would be ignored and overriden with either 'utf-8' or None depending on the type of response content passed in. If you pass this value we should honour it and perform the encoding correctly.

9.1.10 1.5.2

Prelude

Fix py.test plugin with py.test < 3.0

Bug Fixes

- Fixed a bug relating to how the pytest version was being discovered that meant new versions of pytest were being treated as old versions and would receive bad configuration.
- The py.test plugin was broken when using py.test < 3.0. The version of py.test that ships in EPEL is only 2.7 so we need to make sure we support at least that version.

9.1.11 1.5.1

New Features

- The stream parameter is recorded when the request is sent and available in request history in the same was as parameters like verify or timeout.

9.1.12 1.5.0

Prelude

The primary repository is now at https://github.com/jamielennox/requests-mock

New Features

- Added pytest fixture for native integration into pytest projects.

Other Notes

- In this release the main repository was moved off of OpenStack provided infrastructure and onto github at https://github.com/jamielennox/requests-mock. OpenStack has been a great home for the project however requests-mock is a general python project with no specific relationship to OpenStack and the unfamiliar infrastructure was limiting contributes from the wider community.

9.1.13 1.3.0

New Features

- Allow specifying an *additional_matcher* to the mocker that will call a function to allow a user to add their own custom request matching logic.

9.1.14 1.1.0

Prelude

Add a called_once property to the mockers.

It is now possible to make URL matching and request history not lowercase the provided URLs.

Installing the requirements for the 'fixture' contrib package can now be done via pip with *pip install requests-mock[fixture]*

New Features

- A called_once property was added to the adapter and the mocker. This gives us an easy way to emulate mock's assert_called_once.
- You can pass case_sensitive=True to an adapter or set *requests_mock.mock.case_sensitive = True* globally to enable case sensitive matching.
- Added 'fixture' to pip extras so you can install the fixture requirements with *pip install requests-mock[fixture]*

Upgrade Notes

- It is recommended you add *requests_mock.mock.case_sensitive = True* to your base test file to globally turn on case sensitive matching as this will become the default in a 2.X release.

Bug Fixes

- Reported in bug #1584008 all request matching is done in a case insensitive way, as a byproduct of this request history is handled in a case insensitive way. This can now be controlled by setting case_sensitive to True when creating an adapter or globally.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search