Pretenders Documentation

Release 1.4.4

Carles Barrobés and Alex Couper

Jul 01, 2017

Contents

Pretenders are Mocks for network applications. They are mainly designed to be used in system integration tests or manual tests where there is a need to simulate the behaviour of third party software that is not necessarily under your control.

CHAPTER 1

Installation

Simply type:

```
$ pip install pretenders
```

If you want to run the UI, install like this, so that extra dependencies for the frontend are included:

```
$ pip install pretenders[ui]
```

CHAPTER 2

Example usage

Start the server to listen on all network interfaces:

```
$ python -m pretenders.server.server --host 0.0.0.0 --port 8000
```

If you prefer, you can run the pretenders server in docker:

```
docker run -d --name pretenders -p 8000 pretenders/pretenders:1.4
```

HTTP mock in a test case

Sample HTTP mocking test case:

```
from pretenders.client.http import HTTPMock
from pretenders.common.constants import FOREVER

# Assume a running server
# Initialise the mock client and clear all responses
mock = HTTPMock('localhost', 8000)

# For GET requests to /hello reply with a body of 'Hello'
mock.when('GET /hello').reply('Hello', times=FOREVER)

# For the next three POST or PUT to /somewhere, simulate a BAD REQUEST status code
mock.when('(POST|PUT) /somewhere').reply(status=400, times=3)

# For the next request (any method, any URL) respond with some JSON data
mock.reply('{"temperature": 23}', headers={'Content-Type': 'application/json'})

# For the next GET request to /long take 100 seconds to respond.
mock.when('GET /long').reply('', after=100)

# If you need to reply different data depending on request body
```

```
# Regular expression to match certain body could be provided
mock.when('POST /body_requests', body='1.*').reply('First answer', times=FOREVER)
mock.when('POST /body_requests', body='2.*').reply('Second answer', times=FOREVER)

# Your code is exercised here, after setting up the mock URL
myapp.settings.FOO_ROOT_URL = mock.pretend_url
...

# Verify requests your code made
r = mock.get_request(0)
assert_equal(r.method, 'GET')
assert_equal(r.url, '/weather?city=barcelona')
```

HTTP mocking for remote application

Sometimes it is not possible to alter the settings of a running remote application on the fly. In such circumstances you need to have a predetermined url to reach the http mock on so that you can configure correctly ahead of time.

Let's pretend we have a web app that on a page refresh gets data from an external site. We might write some tests like:

```
from pretenders.client.http import HTTPMock
from pretenders.constants import FOREVER

mock = HTTPMock('my.local.server', 9000, timeout=20, name="third_party")

def setup_normal():
    mock.reset()
    mock.when("GET /important_data").reply(
        '{"account": "10000", "outstanding": "10.00"}',
        status=200,
        times=FOREVER)

def setup_error():
    mock.reset()
    mock.when("GET /important_data").reply('ERROR', status=500, times=FOREVER)


@with_setup(setup_normal)
def test_shows_account_information_correctly():
    # Get the webpage
    ...
    # Check that the page shows things correctly as we expect.
    ...

@with_setup(setup_error)
def test_application_handles_error_from_service():
    # Get the webpage
    ...
    # Check that the page gracefully handles the error that has happened
    # in the background.
    ...
```

If you have a test set like the one above you know in advance that your app needs to be configured to point to:

```
http://my.local.server:9000/mockhttp/third_party
```

instead of the actual third party's website.

SMTP mock in a test case

Sample SMTP mocking test case:

```
# Create a mock smtp service
smtp_mock = SMTPMock('localhost', 8000)

# Get the port number that this is faking on and
# assign as appropriate to config files that the system being tested uses
settings.SMTP_HOST = "localhost:{0}".format(smtp_mock.pretend_port)

# ...run functionality that should cause an email to be sent

# Check that an email was sent
email_message = smtp_mock.get_email(0)
assert_equals(email_message['Subject'], "Thank you for your order")
assert_equals(email_message['From'], "foo@bar.com")
assert_equals(email_message['To'], "customer@address.com")
assert_true("Your order will be with you" in email_message.content)
```

CHAPTER 3

Source code

Sources can be found at https://github.com/pretenders/pretenders

Contributions are welcome.

CHAPTER 4

Contents

HTTP Pretenders

Mock HTTP server

The pretenders HTTP server provides 2 subsets of functionality:

- The configuration service, for pre-setting responses and reading back the recorded information. This is the part that your test code will interact with, or that you will be presetting manually, in the case of manual tests.
- The mocked or replay service, for regular operation (which plays back pre-set responses). This is the part that your application under test will talk to.

The configuration service

A RESTful service to pre-program responses or expectations. Presets and History are exposed as resources. Typical HTTP methods are accepted (POST to add element to collection, PUT to modify an element, GET collection or element, DELETE collection or element).

What follows is a description of the server's *wire* protocol, i.e. at the HTTP level.

Resource URLs:

- `/preset/` - the collection of presets
- `/history/` - the collection of recorded requests, where you can also target individual items, e.g.:
  - `/history/0` - the first recorded request

The presets contain all the information necessary to later produce a pre-canned response. The mapping is as follows:

- response body = preset request body
- response status = preset header `X-Pretend-Response-Status`
- response headers = all request headers, excluding those beginning with `X-Pretend-`

It also contains optional matching information to determine which pre-canned responses apply to which requests: * URL matcher (regex) = preset header `X-Pretend-Match-Url` * HTTP methods to match (regex) = preset header `X-Pretend-Match-Method`

Preset responses are returned by the mock service in the order they have been preset, as long as they match, i.e. the first precanned response in the first matching preset (both URL and method) will be returned.

History will contain all requests that have been issued to the mock service. History can be queried with HTTP GET, and responses will contain information from the saved requests:

- The body will be the original request body
- The method used will be in a header `X-Pretend-Request-Header`
- The (relative) URL will be in a header `X-Pretend-Request-Path`

The mocked service

Requests to the mocked service will return the preset responses in order. All request information is then stored for future verification. Stored data includes:

- HTTP Method (GET, POST...)
- Relative URL (`/city/association?n=12`)
- HTTP Headers of the request
- Body

The Python client

Pretenders comes with a python client which can be used to configure HTTP mocks

class `pretenders.client.http.`**`HTTPMock`**(*host*, *port*, *timeout=None*, *name=None*)

A mock HTTP server as seen from the test writer.

The test will first preset responses on the server, and after execution it will enquire the received requests.

Example usage:

```
from pretenders.client.http import HTTPMock
mock = HTTPMock('localhost', 8000)
mock.when('GET /hello').reply('Hello')
# run tests... then read received responses:
r = mock.get_request(0)
assert_equal(r.method, 'GET')
assert_equal(r.url, '/hello?city=barcelona')
```

`get_request`(*sequence_id=None*)

Get a stored request issued to the mock server, by sequence order.

If sequence_id is not given return the whole history of requests to this mock.

`pretend_url`

The full URL of the pretend server.

`reply`(*body=''*, *status=200*, *headers={}*, *times=1*, *after=0*)

Set the pre-canned reply for the preset.

Parameters **`after`** – The http mock server will delay for `after` seconds before replying. Defaults to 0.

when(*rule=''*, *headers=None*, *body=None*)

Set the match rule which is the first part of the Preset.

Parameters

- **rule** – String incorporating the method and url to match eg "GET url/to/match"
- **headers** – An optional dictionary of headers to match.

Note: `rule` is matched as a regular expression and can therefore be set to match more than one request. eg. `r'^GET /something/([a-zA-Z0-9\-_]*)/?'`

Also note that it is always seen as a regex and therefore to match `"GET /foo?bar=1"` you would need to use something like:

```
'GET /foo\?bar=1'
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index