SMock Documentation

Release 0.0.7

bprinty

Feb 01, 2019

Contents

CHAPTER 1

Overview

The faux provides utilities for mocking responses from external services during testing. With *faux*, you can easily serve a directory structure mocking url endpoints for an externally managed service and use that server for testing.

For documentation on how to use this package, see the Usage section of the documentation.

CHAPTER 2

Content:

2.1 Installation

2.1.1 Through pip

```
$ pip install gems
```

2.1.2 Via GitHub

```
$ git clone http://github.com/bprinty/faux.git
$ cd faux
$ python setup.py install
```

2.1.3 Questions/Feedback

File an issue in the GitHub issue tracker.

2.2 Usage

The sections below detail different paradigms for using this library. In the documentation below, you'll learn how to use **faux** for: 1) mocking a filesystem during testing, 2) defining a test fixture for mocking an external service, 3) mocking dynamic requests with random data, and 4) pulling mock data from an external service for downstream testing.

2.2.1 Filesystem Mocking

For instance, if you have a directory structure that looks like the following:

```
├── _uuid
├── file
└── query/
    ├── data
    └── arg=test
```

With the following as contents of the files in that directory structure:

```
# _uuid
{
    "status": "ok",
    "city": "{{city}}"
}

# file
{
    "status": "ok",
    "month": "{{month}}",
}

# query/arg=test
{
    "status": "ok",
    "arg": "test",
    "digit": {{random_digit}}
}

# query/data
{
    "status": "ok",
    "data": "test"
}
```

You can serve the directory structure using (the *-P* option below is specifying a specific port):

```
~$ faux serve -P 1234 /path/to/directory
```

And endpoints mirroring that file structure will be available:

```
>>> import requests
>>> r = requests.get('http://localhost:1234/4db5fd8c-8aa6-4c29-b979-dab3ce71e64e')
>>> print(r.json())
{
    "status": "ok",
    "city": "Sacramento",
}

>>> r = requests.get('http://localhost:1234/file')
>>> print(r.json())
{
    "status": "ok",
    "month": "05"
}
```

(continues on next page)

(continued from previous page)

```
>>> r = requests.get('http://localhost:1234/query?arg=test')
>>> print(r.json())
{
    "status": "ok",
    "arg": "test",
    "digit": 4
}

>>> r = requests.get('http://localhost:1234/query/data')
>>> print(r.json())
{
    "status": "ok",
    "data": "test"
}
```

It's also worth noting (alluded to above) that you can mock arbitrary data in your responses using methods from the faker library. Items like `{{city}}` and `{{month}}` above were automatically and randomly filled without outputs from a `faker.Faker()` object during the request. For more information about the types of data you can fake, see the faker documentation.

One other special file above is the `_uuid` file, which will return data from the `_uuid` file whenever a uuid is included as part of the request.

2.2.2 Endpoint Mocking

Along with mocking endpoints via filesystem contents, you can also mock endpoints dynamically using the `faux` library. Here's and example of how to set up dynamic mocks:

```
# imports
from faux import Server

# set up app
app = Server(__name__, cache='/path/to/directory')

# define routes for testing
@app.route('/simple', methods=['GET', 'POST', 'PUT', 'DELETE'])
def simple():
    """
    Simple endpint with get/post
    """
    return {
        'status': 'ok',
        'uuid': '{{uuid}}',
        'name': '{{name}}',
        'address': '{{address}}'
    }

@app.route('/nested/<param>', methods=['GET', 'POST', 'PUT', 'DELETE'])
def nested(param):
    """
    Manage server state.
    """
    return {
        'status': 'ok',
        'param': param,
```

(continues on next page)

(continued from previous page)

```
        'company': '{{company}}',
        'number': '{{random_int}}',
    }

# run
if __name__ == '__main__':
    import time
    with app.run(port=1234, debug=True):
        while True:
            time.sleep(1)
```

Note that `faux` uses Flask under the hood to manage endpoint resolution and routing, so the API for this library is very similar to the Flask API. The code above will allow you mock all of the contents of a specified directory, and also the dynamic mocks you've configured with the `route` decorator:

```
>>> import requests
>>> r = requests.get('http://localhost:1234/query/data')
>>> print(r.json())
{
    "status": "ok",
    "data": "test"
}
>>>
>>> r = requests.get('http://localhost:1234/simple')
>>> print(r.json())
{
    "status": "ok",
    "uuid": "4db5fd8c-8aa6-4c29-b979-dab3ce71e64e",
    "name": "Gary Armstrong",
    "address": "97183 Orozco Islands Suite 483\nAndersonton, KS 57080"
}
>>>
>>> r = requests.get('http://localhost:1234/nested/test')
>>> print(r.json())
{
    "status": "ok",
    "param": "test",
    "company": "Perez PLC",
    "number": "8032",
}
```

2.2.3 Testing Fixtures

One of the most common paradigms for using this software is to mock a service during testing. To do so with this module, you can easily set up a py.test fixture that will run throughout your test session:

```
import unittest
import pytest

RESOURCES = '/path/to/testing/resources'

@pytest.fixture(scope='session')
def server():
    """
    Set up mock server for testing request caching.
```

(continues on next page)

(continued from previous page)

```
    """
    from faux import Server
    app = Server(__name__, cache=RESOURCES)
    with app.run(port=1234):
        yield
    return
```

Once you've defined the fixture, you can use it on a test class or function like so:

```
# test function
@pytest.mark.usefixtures("server")
def test_function():
    return


# test class
@pytest.mark.usefixtures("server")
class TestClass(unittest.TestCase):
    def test_method():
        return
```

With the code above, the server you're mocking will run throughout your testing session and will gracefully exit when the test session stops.

2.2.4 Caching Request Data

Along with serving a directory structure with request data, you can generate that directory structure by querying data from an existing server. For example, if we already had a service that provided the endpoints we tried to mock above, we could query and save that data in a directory structure (for mocking later on) like so:

```
>>> from faux import requests
>>> requests.cache('/path/to/cache/directory')
>>> requests.get('http://localhost:1234/file')
>>> requests.get('http://localhost:1234/query?arg=test')
>>> requests.get('http://localhost:1234/query/data')
>>> requests.post('http://localhost:1234/query', json={'data': 'test'})
```

And the contents of our cache directory will look like:

```
├── GET/
│   ├── _uuid
│   └── query/
│       ├── data
│       └── arg=test
└── POST/
    └── query/
        └── 91cc355
```

With the files above containing the data from those requests. After generating that cache directory, you can turn around and serve it for testing using `faux serve` or using a test fixture.

2.2.5 Command-Line

Along with the `serve` entrypoint, here is the full set of command-line options available from the *faux* entry-point:

```
~$ faux -h
usage: faux [-h] {version,status,serve} ...

positional arguments:
  {version,status,serve}

optional arguments:
  -h, --help            show this help message and exit
```

Starting a Server

To start a faux server with an existing directory, you can use the `serve` entrypoint:

```
~$ faux -h
usage: faux serve [-h] [-P PORT] [-n NAME] [-t TIMEOUT] [-l LOG_LEVEL] path

positional arguments:
  path                  Directory structure to serve.

optional arguments:
  -h, --help            show this help message and exit
  -P PORT, --port PORT  Port to run server on.
  -n NAME, --name NAME  Optional name for server.
  -t TIMEOUT, --timeout TIMEOUT
                        Timeout for stopping server (seconds).
  -l LOG_LEVEL, --log-level LOG_LEVEL
                        Logging verbosity (DEBUG, INFO, ERROR, WARNING,
                        CRITICAL, etc ...). Default is INFO
```

Example:

```
~$ faux serve -P 1234 -l INFO -t 100 /path/to/directory
```

Checking the Status of a Server

To check the status of a running server, you can use the `status` entrypoint:

```
~$ faux -h
usage: faux status [-h] [-S] [-H HOST] [-P PORT]

optional arguments:
  -h, --help            show this help message and exit
  -S, --ssl             Use ssl for connecting to server.
  -H HOST, --host HOST  Host to check.
  -P PORT, --port PORT  Port to check.
```

Example:

```
~$ faux status -P 1234
{'status': 'ok'}
```

2.2.6 Questions/Feedback

File an issue in the GitHub issue tracker.

2.3 API

2.3.1 Caching Request Data

faux.client.**get**(**args*, ***kwargs*)
Sends a GET request.

Parameters

- **url** – URL for the new Request object.
- **params** – (optional) Dictionary, list of tuples or bytes to send in the body of the Request.
- ****kwargs** – Optional arguments that request takes.

Returns Response object

Return type requests.Response

faux.client.**post**(**args*, ***kwargs*)
Sends a POST request.

Parameters

- **url** – URL for the new Request object.
- **data** – (optional) Dictionary, list of tuples, bytes, or file-like object to send in the body of the Request.
- **json** – (optional) json data to send in the body of the Request.
- ****kwargs** – Optional arguments that request takes.

Returns Response object

Return type requests.Response

faux.client.**put**(**args*, ***kwargs*)
Sends a PUT request.

Parameters

- **url** – URL for the new Request object.
- **data** – (optional) Dictionary, list of tuples, bytes, or file-like object to send in the body of the Request.
- **json** – (optional) json data to send in the body of the Request.
- ****kwargs** – Optional arguments that request takes.

Returns Response object

Return type requests.Response

faux.client.**delete**(**args*, ***kwargs*)
Sends a DELETE request.

Parameters

- **url** – URL for the new Request object.
- ****kwargs** – Optional arguments that request takes.

Returns Response object

Return type requests.Response

2.3.2 Mocking Servers

class faux.server.**Server**(**args*, ***kwargs*)
Object mimicking flask server to allow for spinning up server mock.

init()
Method for decorating custom url handlers on server.

logger
Expose flask logger so user can change settings.

TODO: update this class to use __getattr__ for defaulting to internal getattr(self.flask, item)

route(**args*, ***kwargs*)
Override flask route decorator to provide easier UX for return data. With these changes, users can simply return a dictionary or xml Element object instead of needing to craft a full response.

class faux.server.**Instance**(*app*, ***kwargs*)
Contextmanager for running managing server mock.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index