API Wrapper Documentation

Release 0.1.7

Ardy Dedase

February 09, 2017

Contents

1 API Wrapper **3**
- 1.1 Overview . . . 3
- 1.2 Installation . . . 3
- 1.3 Getting started with a simple request . . . 3
- 1.4 More features including polling . . . 4

2 Installation **5**

3 Usage **7**
- 3.1 Basic Usage . . . 7
- 3.2 Polling . . . 8
- 3.3 Response callbacks . . . 10

4 Contributing **11**
- 4.1 Types of Contributions . . . 11
- 4.2 Get Started! . . . 12
- 4.3 Pull Request Guidelines . . . 12
- 4.4 Tips . . . 13

5 Credits **15**
- 5.1 Development Lead . . . 15
- 5.2 Contributors . . . 15

6 History **17**

7 0.1.0 (2015-01-11) **19**

8 Indices and tables **21**

Contents:

CHAPTER 1

API Wrapper

Simple API Wrapper

- Free software: BSD license
- Documentation: https://apiwrapper.readthedocs.org.

1.1 Overview

Recently noticed a pattern and repeated pieces of code in Python API wrappers for simple requests and polling. A separate Python package will minimize code duplication and encourage de-coupling of logic from the API request functions.

1.2 Installation

At the command line:

```
$ easy_install apiwrapper
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv apiwrapper
$ pip install apiwrapper
```

1.3 Getting started with a simple request

```
# as a helper class
from apiwrapper import APIWrapper

my_api = APIWrapper()
url = 'https://api.github.com/users/ardydedase/repos'
resp = my_api.make_request(url=url)
print(resp)

# as a parent class
class GithubAPI(APIWrapper):
    def get_repos(self, username):
        """
```

```
        Uses `make_request` method
        """
        url = "https://api.github.com/users/{username}/repos".format(username=username)
        return self.make_request(url, method='get', headers=None, data=None, callback=None)
```

1.4 More features including polling

Read the docs: https://apiwrapper.readthedocs.org/en/latest/usage.html

Or use *apiwrapper/apiwrapper.py* as a reference. Implementation is straightforward.

CHAPTER 2

Installation

At the command line:

```
$ easy_install apiwrapper
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv apiwrapper
$ pip install apiwrapper
```

CHAPTER 3

Usage

3.1 Basic Usage

To use API Wrapper in a project:

```
from apiwrapper import APIWrapper
```

Use it as a helper:

```
my_api = APIWrapper()
url = 'https://api.github.com/users/ardydedase/repos'
resp = my_api.make_request(url, method='get', headers=None, data=None, callback=None).parsed
print(resp)
```

Use it as a parent class:

```
class GithubAPI(APIWrapper):
    def get_repos(self, username):
        """
        Uses 'make_request' method
        """
        url = "https://api.github.com/users/{username}/repos".format(username=username)
        return self.make_request(url, method='get', headers=None, data=None, callback=None).parsed
```

Parameter reference for *make_request()*:

```
def make_request(self, url, method='get', headers=None, data=None,
                 callback=None, errors=STRICT, verify=False, **params):
    """
    Reusable method for performing requests.
    :param url - URL to request
    :param method - request method, default is 'get'
    :param headers - request headers
    :param data - post data
    :param callback - callback to be applied to response,
                      default callback will parse response as json object.
    :param errors - specifies communication errors handling mode, possible
                    values are:
                     * strict (default) - throw an error as soon as one
                       occurred
                     * graceful - ignore certain errors, e.g. EmptyResponse
                     * ignore - ignore all errors and return a result in
                                any case.
                                NOTE that it DOES NOT mean that no
```

```
                                exceptions can be
                                raised from this method, it mostly ignores
                                communication
                                related errors.
                        * None or empty string equals to default
    :param verify - whether or not to verify SSL cert, default to False
    :param timeout - the timeout of the request in second, default to None
    :param params - additional query parameters for request
    """
```

3.2 Polling

APIWrapper's built-in polling method makes it convenient to declare polling methods and calls. Its flexibility allows a number of options including switching between JSON and XML response types.

In this *poll* method example, let's use Skyscanner's API.

Let's start by importing *APIWrapper* class and all the error modes available in the apiwrapper package:

```
from apiwrapper import (
    APIWrapper,
    STRICT,
    GRACEFUL,
    IGNORE)
```

Next will be to declare the *Flights* class that will inherit our *APIWrapper* parent class. The parent APIWrapper class is initizialized with *response_format='json'*. The *api_key* is a private property so we don't have to pass it as an argument every time we call *make_request*:

```
class Flights(APIWrapper):
    """
    Skyscanner Flights Live Pricing
    http://business.skyscanner.net/
    portal/en-GB/Documentation/FlightsLivePricingList
    """
    API_HOST = 'http://partners.api.skyscanner.net'
    PRICING_SESSION_URL = '{api_host}/apiservices/pricing/v1.0'.format(
        api_host=API_HOST)

    def __init__(self, api_key):
        self.api_key = api_key
        super(Flights, self).__init__(response_format='json')
```

Wrap the *make_request* method from *APIWrapper* and inject the *apikey* only if it is not available in the request url:

```
def make_request(self, url, method='get', headers=None,
                 data=None, callback=None, errors=STRICT,
                 verify=False, **params):
    """
    Call the `make_request` method from apiwrapper.
    So we can inject the apikey when it is not available.
    """
    if 'apikey' not in url.lower():
        params.update({
            'apiKey': self.api_key
        })
    return super(Flights, self).make_request(url, method, headers,
```

```
                                                    data, callback, errors,
                                                    verify, **params)
```

The *create_session* method prepares the API's polling session and returns the polling url *poll_url*. This method uses the *make_request* method declared above. It also makes use of the *_headers()* method from *APIWrapper*:

```
def create_session(self, **params):
    """
    Create the session
    date format: YYYY-mm-dd
    location: ISO code.
    After creating the session,
    this method will return the poll_url.
    """
    service_url = self.PRICING_SESSION_URL
    return self.make_request(service_url,
                             method='post',
                             headers=self._headers(),
                             callback=lambda resp: resp.headers[
                                 'location'],
                             data=params)
```

This boolean method *is_poll_complete_callback* will be passed as a callback parameter in the *APIWrapper.poll* method call. *is_poll_complete_callback* will receive the *poll* response from *poll* method as a parameter. This method will then use the *poll_resp* value to check whether the polling is complete or not and returns a boolean:

```
def _is_poll_complete_callback(self, poll_resp):
    """
    Checks the condition in poll response to determine if it is complete
    and no subsequent poll requests should be done.
    """
    if poll_resp.parsed is None:
        return False
    success_list = ['UpdatesComplete', True, 'COMPLETE']
    status = None
    if self.response_format == 'xml':
        status = poll_resp.parsed.find('./Status').text
    elif self.response_format == 'json':
        status = poll_resp.parsed.get(
            'Status', poll_resp.parsed.get('status'))
    if status is None:
        raise RuntimeError('Unable to get poll response status.')
    return status in success_list
```

And lastly, the *get_result* method polls the API using the URL that was returned from *create_session*. Notice that we are passing *_is_poll_complete_callback* as an argument to the *is_poll_complete_callback* parameter in the *poll* method. After the poll is complete, the *get_result* method will return the flight search result:

```
def get_result(self, errors=STRICT, **params):
    """
    Get all results, no filtering,
    etc. by creating and polling the session.
    """
    service_url = self.create_session(**params)
    return self.poll(service_url, errors=errors, is_poll_complete_callback=self._is_poll_complete_cal
```

Now that the *Flights* class is ready. The *get_result* method can be called as follows:

```
from datetime import datetime, timedelta

datetime_format = '%Y-%m-%d'
outbound_datetime = datetime.now() + timedelta(days=7)
inbound_datetime = outbound_datetime + timedelta(days=3)
outbound_date = outbound_datetime.strftime(datetime_format)
inbound_date = inbound_datetime.strftime(datetime_format)

flights_service = Flights(<skyscanner_api_key>)
result = flights_service.get_result(
    errors=GRACEFUL,
    country='UK',
    currency='GBP',
    locale='en-GB',
    originplace='SIN-sky',
    destinationplace='KUL-sky',
    outbounddate=outbound_date,
    inbounddate=inbound_date,
    adults=1).parsed
```

Parameter reference for *poll()*:

```
def poll(self, url, initial_delay=2, delay=1, tries=20, errors=STRICT, is_complete_callback=None, **p
    """
    Poll the URL
    :param url - URL to poll, should be returned by 'create_session' call
    :param initial_delay - specifies how many seconds to wait before the first poll
    :param delay - specifies how many seconds to wait between the polls
    :param tries - number of polls to perform
    :param errors - errors handling mode, see corresponding parameter in 'make_request' method
    :param params - additional query params for each poll request
    """
```

3.3 Response callbacks

callback parameter in *make_request* method. It passes the *Response* object as an argument:

```
class GithubAPI(APIWrapper):
    def _my_callback(self, resp):
        """
        'resp' is a Response object returned from `requests` library
        """
        return resp.json()


    def get_repos(self, username):
        """
        Uses 'make_request' method
        """
        url = "https://api.github.com/users/{username}/repos".format(username=username)
        return self.make_request(url, method='get', headers=None, data=None, callback=self._my_callba
```

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/ardydedase/apiwrapper/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

4.1.4 Write Documentation

API Wrapper could always use more documentation, whether as part of the official API Wrapper docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/ardydedase/apiwrapper/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *apiwrapper* for local development.

1. Fork the *apiwrapper* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/apiwrapper.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv apiwrapper
$ cd apiwrapper/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 apiwrapper tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/ardydedase/apiwrapper/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_apiwrapper
```

CHAPTER 5

Credits

5.1 Development Lead

- Ardy Dedase <ardy.dedase@gmail.com>

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

CHAPTER 7

0.1.0 (2015-01-11)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search