Tornado REST Client Documentation

Release 0.0.3

Matt Wise, Mikhail Simin

Jun 17, 2017

Contents

The `tornado_rest_client` framework provides a quick and easy way to build generic API clients for JSON REST-based APIs. The framework provides robust and reliable retry mechanisms, error handling and exception raising all within a simple to use class structure.

The basic purpose of the *api* package is to provide you with a few simple inheritable classes where all you need to do is fill in a few variables to get back a usable API client.

Every API client you build will be a combination of two objects – a *RestClient* and a *RestConsumer*.

CHAPTER 1

RestClient

A *RestClient* object is a very simple object that exposes one public *fetch()* method (that's wrapped in the coroutine() wrapper) used to fire off HTTP calls through a tornado.httpclient.AsyncHTTPClient object.

CHAPTER 2

RestConsumer

The *RestConsumer* class does the real leg work. At the root of it, the object self-configures itself with a supplied *CONFIG* dictionary that defines `http_methods`, `path` and possible `attrs`. The `http_methods` and `path` work together to tell the object exactly what path it will call out to, and what methods it supports. The `attrs` provide links to nested methods that return other *RestConsumer* objects.

If you consider an API that may have the following endpoints:

- **GET /**: Returns 200 if API is up
- **GET /cats**: Returns a `array` of cat names
- **POST /cats**: Push a new name to the `array` of cat names
- **GET /cats/random**: Returns a single random cat name

You can define your *CONFIG* dict like this:

```
class CatAPI(api.RestConsumer):
    ENDPOINT = 'http://my_cat_service'
    CONFIG = {
        # Handles GET /
        'path': '/',
        'http_methods': {'get': {}},
        # Creates a series of methods that return other RestConsumers
        'attrs': {
            # Handles GET /cats, POST /cats
            'cat_api': {
                'path': '/cats',
                'new': True,
                'http_methods': {
                    'get': {},
                    'post': {},
                },
                # Now, handles the random cat endpoint
                'attrs': {
                    'random': {
                        'path': '/cats/random',
```

```
                        'new': True,
                        'http_methods': {
                            'get': {}
                        }
                    },
                    'get': {
                        'path': '/cats/%id%',
                        'http_methods': {
                            'get': {}
                        }
                    }
                }
            }
        }
    }
```

Now, instantiating this object would provide methods that look like this:

```
>>> cats = CatAPI()
>>> cats
CatAPI(/)
>>> cats.cat_api
CatAPI(/cats)
>>> cats.cat_api.random
CatAPI(/cats/random)
>>> cats.cat_api.random.http_get()
<tornado.concurrent.Future object at 0x101f9e390>
>>> yield cats.cat_api.random().http_get()
'Bob Marley!'
>>> yield cats.cat_api.http_post(cat_name='Skippy')
{ "status": "ok" }
>>> yield cats.cat_api.get(id='Bobby').http_get()
{ "cat": "Bobby" }
```

There are more details available inside the various doc modules below...

Getting Started

Getting started with `tornado_rest_client` is easy.

- Define the API methods you plan to support
- Build any custom functions that you need
- Ship it!

Install the test-specific dependencies

```
(.venv) $ pip install -r tornado_rest_client/requirements.test.txt
...
(.venv) $ cd tornado_rest_client
(.venv) $ python setup.py test
...
```

Testing

Unit Tests

The code is 100% unit test coverage complete, and no pull-requests will be accepted that do not maintain this level of coverage. That said, it's possible (*likely*) that we have not covered every possible scenario in our unit tests that could cause failures. We will strive to fill out every reasonable failure scenario.

Integration Tests

Because it's hard to predict cloud failures, we provide integration tests for most of our modules. These integration tests actually go off and execute real operations in your accounts, and rely on particular environments being setup in order to run. credentials are all correct.

Executing the tests

```
PYFLAKES_NODOCTEST=True python setup.py integration pep8 pyflakes
```

Simple API Access Objects

Most of the APIs out there leverage basic REST with JSON or XML as the data encoding method. Since these APIs behave similarly, we can define the API URLs and HTTP methods inside a `dict`, without writing any actual python methods.

HTTPBin RestConsumer

```
HTTPBIN = {
    'path': '/',
    'http_methods': {'get': {}},
    'attrs': {
        'get': {
            'path': '/get',
            'http_methods': {'get': {}},
        },
        'post': {
            'path': '/post',
            'http_methods': {'post': {}},
        },
        'put': {
            'path': '/put',
            'http_methods': {'put': {}},
        },
        'delete': {
            'path': '/delete',
            'http_methods': {'delete': {}},
        },
    }
}


class HTTPBinRestClient(api.RestConsumer):
```

```
    CONFIG = HTTPBIN
    ENDPOINT = 'http://httpbin.org'


class HTTPBinGetThenPost(object):
    def __init__(self, \*args, \**kwargs):
        super(HTTPBinGetThenPost, self).__init__(\*args, \**kwargs)
        self._api = HTTPBinRestClient(timeout=60)

    @gen.coroutine
    def execute(self):
        yield self._api.get().http_get()
        yield self._api.post().http_post(foo='bar')
```

Exception Handling in HTTP Requests

The *fetch()* method has been wrapped in a *retry()* decorator that allows you to define different behaviors based on the exceptions returned from the fetch method. For example, you may want to handle an `HTTPError` exception with a `401` error code differently than a `503` error code.

You can customize the exception handling by subclassing the *RestClient*:

```
class MyRestClient(api.RestClient):
    EXCEPTIONS = {
        httpclient.HTTPError: {
            # These do not retry, they immediately raise an exception
            '401': my.CustomException(),
            '403': exceptions.InvalidCredentials,
            '500': my.UnretryableError(),
            '502': exceptions.InvalidOptions,

            # This indicates a retry should happen
            '503': None,

            # This acts as a catch-all
            '': MyException,
        }
    }
```

Module Documentation

tornado_rest_client.api

This package provides a quick way of creating custom API clients for JSON-based REST APIs. The majority of the work is in the creation of a *RestConsumer.CONFIG* dictionary for the class. This dictionary dynamically configures the object at instantiation time with the appropriate `coroutine()` wrapped HTTP fetch methods.

class tornado_rest_client.api.**RestConsumer**(*name=None*, *config=None*, *client=None*, **args*, ***kwargs*)

Async REST API Consumer object.

The generic RestConsumer object (with no parameters passed in) looks at the *CONFIG* dictionary and dynamically generates access methods for the various API methods.

The *GET*, *PUT*, *POST* and *DELETE* methods optionally listed in `CONFIG['http_methods']` represent the possible types of HTTP methods that the `CONFIG['path']` supports. For each one of these listed, a `coroutine()` wrapped `http_get()`, `http_put()`, `http_post()`, or `http_delete()` method will be created.

For each item listed in `CONFIG['attrs']`, an access method is created that creates and returns a new RestConsumer object that's configured for this endpoint. These methods are not asynchronous, but are non-blocking.

Parameters

- **name** (*str*) – Name of the resource method (default: None)
- **config** (*dict*) – The dictionary object with the configuration for this API endpoint call.
- **client** (`RestClient`) – The *RestClient* compatible object used to actually fire off HTTP requests.
- **kwargs** (*dict*) – Any named arguments that should be passed along in the web request through the *replace_path_tokens()* method. This allows for string replacement in URL paths, like `/api/%resource_id%/terminate` to have the `%resource_id%` token replaced with something you've passed in here.

ENDPOINT = None

The URL of the API Endpoint. (for example: http://httpbin.org)

CONFIG = {}

The configuration dictionary for the REST API. This dictionary consists of a root object that has three possible named keys: `path`, `http_methods` and `attrs`.

•*path*: The API Endpoint that any of the HTTP methods should talk to.

•*http_methods*: A dictionary of HTTP methods that are supported.

•*attrs*: A dictionary of other methods to create that reference other API URLs.

•*new*: Set to True if you want to create an access property rather an access method. Only works if your path has no token replacement in it.

This data can be nested as much as you'd like

```
>>> CONFIG = {
...     'path': '/', 'http_methods': {'get': {}},
...     'new': True,
...     'attrs': {
...         'getter': {'path': '/get', 'htpt_methods': {'get': {}}},
...         'poster': {'path': '/post', 'htpt_methods': {'post': {}}},
...     }
... }:
```

replace_path_tokens(*path*, *tokens*)

Search and replace `%xxx%` with values from tokens.

Used to replace any values of `%xxx%` with `'xxx'` from tokens. Can replace one, or many fields at aonce.

Parameters

- **path** (*str*) – String of the path
- **tokens** (*dict*) – A dictionary of tokens to search through.

Returns A modified string

_create_http_methods()

Create `coroutine()` wrapped HTTP methods.

Iterates through the methods described in `self._http_methods` and creates `coroutine()` wrapped access methods that perform these actions.

_create_consumer_methods()
Creates access methods to the attributes in `self._attrs`.

Iterates through the attributes described in `self._attrs` and creates access methods that return *RestConsumer* objects for those attributes.

class tornado_rest_client.api.**RestClient**(*client=None*, *headers=None*, *timeout=None*)
Simple Async REST client for the RestConsumer.

Implements a `AsyncHTTPClient`, some convinience methods for URL escaping, and a single *fetch()* method that can handle GET/POST/PUT/DELETEs.

Parameters **headers** (*dict*) – Headers to pass in on every HTTP request

EXCEPTIONS = {<class 'tornado.httpclient.HTTPError'>: {'': <class 'tornado_rest_client.exceptions.RecoverableFailure
Dictionary describing the exception handling behavior for HTTP calls. The dictionary should look like this:

```
>>> {
...     <exception type... aka httpclient.HTTPError>: {
...         `<string to match in exception.message>`: <raises exc>,
...         '<this string triggers a retry>': None,
...         '': <all other strings trigger this exception>
...     }
```

_generate_escaped_url(*url*, *args*)
Generates a fully escaped URL string.

Sorts the arguments so that the returned string is predictable and in alphabetical order. Effectively wraps the `tornado.httputil.url_concat()` method and properly strips out `None` values, as well as lowercases `Bool` values.

Parameters

- **url** (*str*) – The URL to append the arguments to
- **args** (*dict*) – Key/Value arguments. Values should be primitives.

Returns URL encoded string like this: `<url>?foo=bar&abc=xyz`

fetch(**args*, ***kwargs*)
Executes a web request asynchronously and yields the body.

Parameters

- **url** (*str*) – The full url path of the API call
- **params** (*dict*) – Arguments (k/v pairs) to submit either as POST data or URL argument options.
- **method** (*str*) – GET/PUT/POST/DELETE
- **auth_username** (*str*) – HTTP auth username
- **auth_password** (*str*) – HTTP auth password

Yields String of the returned text from the web service.

class tornado_rest_client.api.**SimpleTokenRestClient**(*tokens*, **args*, ***kwargs*)
Bases: *tornado_rest_client.api.RestClient*

Simple RestClient with a token for HTTP authentication.

Used in most simple APIs where a token is provided to the end user.

Parameters **tokens** (*dict*) – A dict with the token name/value(s) to append to every web request.

tornado_rest_client.api.**retry**(*func=None*, *retries=3*, *delay=0.25*)

Coroutine-compatible retry decorator.

This decorator provides a simple retry mechanism that compares the exceptions it received against a configuration list stored in the calling-object(*RestClient.EXCEPTIONS*), and then performs the action defined in that list. For example, an `HTTPError` with a '500' code might want to retry 3 times. On the otherhand, a `401`/`403` might want to throw an `InvalidCredentials` exception.

Examples:

```
>>> @gen.coroutine
... @retry
... def some_func(self):
...     yield ...
```

```
>>> @gen.coroutine
... @retry(retries=5):
... def some_func(self):
...     yield ...
```

tornado_rest_client.api.**create_http_method**(*name*, *http_method*)

Creates the *GET*/*PUT*/*DELETE*/*POST* function for a RestConsumer.

This method is called by *RestConsumer._create_http_methods()* to create a method for the *RestConsumer* object with the appropriate name and HTTP method (`http_get()`, `http_put()`, `http_delete()`, `http_post()`)

Parameters

- **name** (*str*) – Full name of the function to create (ie, `http_get`)
- **http_method** (*str*) – Name of the method (ie, `get`)

Returns A method appropriately configured and named.

tornado_rest_client.api.**create_consumer_method**(*name*, *config*)

Creates a method that returns a configured RestConsumer object.

RestConsumer objects themselves can have references to other RestConsumer objects. For example, the `Slack` object has no `http_*()` methods itself, but it does have methods like `auth_test()` which return a fresh *RestConsumer* object that points to the `/api/auth.test` API endpoint and provide `http_post()` as a function

The method created here accepts any args (`*args, **kwargs`) and passes them on to the *RestConsumer* object being created. This allows for passing in unique resource identifiers (ie, the `%res%` in `/v2/rooms/%res%/history`).

Parameters

- **name** (*str*) – The name of the method to create (ie, `auth_test`)
- **config** (*dict*) – The dictionary of *CONFIG* data specific to the API endpoint that we are configuring (should include `path` and `http_methods` keys).

Returns A method that returns a fresh RestConsumer object

class tornado_rest_client.api.**RestConsumer**(*name=None*, *config=None*, *client=None*, **args*, ***kwargs*)

Async REST API Consumer object.

The generic RestConsumer object (with no parameters passed in) looks at the *CONFIG* dictionary and dynamically generates access methods for the various API methods.

The *GET*, *PUT*, *POST* and *DELETE* methods optionally listed in `CONFIG['http_methods']` represent the possible types of HTTP methods that the `CONFIG['path']` supports. For each one of these listed, a `coroutine()` wrapped `http_get()`, `http_put()`, `http_post()`, or `http_delete()` method will be created.

For each item listed in `CONFIG['attrs']`, an access method is created that creates and returns a new RestConsumer object that's configured for this endpoint. These methods are not asynchronous, but are non-blocking.

Parameters

- **name** (*str*) – Name of the resource method (default: None)
- **config** (*dict*) – The dictionary object with the configuration for this API endpoint call.
- **client** (RestClient) – The *RestClient* compatible object used to actually fire off HTTP requests.
- **kwargs** (*dict*) – Any named arguments that should be passed along in the web request through the *replace_path_tokens()* method. This allows for string replacement in URL paths, like `/api/%resource_id%/terminate` to have the `%resource_id%` token replaced with something you've passed in here.

ENDPOINT = None

The URL of the API Endpoint. (for example: http://httpbin.org)

CONFIG = {}

The configuration dictionary for the REST API. This dictionary consists of a root object that has three possible named keys: `path`, `http_methods` and `attrs`.

•*path*: The API Endpoint that any of the HTTP methods should talk to.

•*http_methods*: A dictionary of HTTP methods that are supported.

•*attrs*: A dictionary of other methods to create that reference other API URLs.

•*new*: Set to True if you want to create an access property rather an access method. Only works if your path has no token replacement in it.

This data can be nested as much as you'd like

```
>>> CONFIG = {
...     'path': '/', 'http_methods': {'get': {}},
...     'new': True,
...     'attrs': {
...         'getter': {'path': '/get', 'htpt_methods': {'get': {}}},
...         'poster': {'path': '/post', 'htpt_methods': {'post': {}}},
...     }
... }:
```

replace_path_tokens(*path*, *tokens*)

Search and replace `%xxx%` with values from tokens.

Used to replace any values of `%xxx%` with `'xxx‘` from tokens. Can replace one, or many fields at aonce.

Parameters

- **path** (*str*) – String of the path
- **tokens** (*dict*) – A dictionary of tokens to search through.

Returns A modified string

class tornado_rest_client.api.**RestClient**(*client=None*, *headers=None*, *timeout=None*)
Simple Async REST client for the RestConsumer.

Implements a AsyncHTTPClient, some convinience methods for URL escaping, and a single *fetch()* method that can handle GET/POST/PUT/DELETEs.

Parameters **headers** (*dict*) – Headers to pass in on every HTTP request

EXCEPTIONS = {<class 'tornado.httpclient.HTTPError'>: {'': <class 'tornado_rest_client.exceptions.RecoverableFailure
Dictionary describing the exception handling behavior for HTTP calls. The dictionary should look like this:

```
>>> {
...     <exception type... aka httpclient.HTTPError>: {
...         `<string to match in exception.message>`: <raises exc>,
...         '<this string triggers a retry>': None,
...         '': <all other strings trigger this exception>
...     }
```

fetch(**args*, ***kwargs*)
Executes a web request asynchronously and yields the body.

Parameters

- **url** (*str*) – The full url path of the API call
- **params** (*dict*) – Arguments (k/v pairs) to submit either as POST data or URL argument options.
- **method** (*str*) – GET/PUT/POST/DELETE
- **auth_username** (*str*) – HTTP auth username
- **auth_password** (*str*) – HTTP auth password

Yields String of the returned text from the web service.

class tornado_rest_client.api.**SimpleTokenRestClient**(*tokens*, **args*, ***kwargs*)
Simple RestClient with a token for HTTP authentication.

Used in most simple APIs where a token is provided to the end user.

Parameters **tokens** (*dict*) – A dict with the token name/value(s) to append to every web request.

tornado_rest_client.clients.slack

A simple Slack API client that provides basic message sending capabilities. Note, many more functions can be added to this class, but initially its very simple.

Usage:

```
>>> api = slack.Slack(token='unittest')
>>> auth_ok = yield api.auth_test().http_post()
>>> print('Auth OK? %s' % api.check_results(auth_ok))
>>> ret = yield api.chat_postMessage().http_post(
...     channel='#systems',
...     text='This is a test message',
...     username='Matt',
...     parse='none',
...     link_names=1,
...     unfurl_links=True,
```

```
...     unfurl_media=True)
>>> print ('Message sent? %s' % api.check_results(ret))
```

class tornado_rest_client.clients.slack.**Slack**(**args*, ***kwargs*)

Bases: *tornado_rest_client.api.RestConsumer*

Simple Slack API Client.

This example API client has very limited functionality – basically it implements the `/api/auth.test` and `/api/chat.postMessage` functions.

auth_test()

Accesses https://api.slack.com/api/auth.test

http_post()

chat_postMessage()

Accesses https://api.slack.com/api/chat.postMessage

http_post(channel, text, username[, as_user, parse, link_names, attachments, unfurl_links, unfurl_media, icon_url, icon_emoji])

check_results(*result*)

Returns True/False if the result was OK from Slack.

The Slack API avoids using standard error codes, and instead embeds error codes in the return results. This method returns True or False based on those results.

Parameters **result** (*dict*) – A return dict from Slack

Raises

- **InvalidCredentials** – if the creds are bad
- **Error** – exception on any other value
- **RequestFailure** – response with no `ok` field

Returns If the API call succeeded or failed without error

Return type bool

replace_path_tokens(*path*, *tokens*)

Search and replace `%xxx%` with values from tokens.

Used to replace any values of `%xxx%` with `'xxx‘` from tokens. Can replace one, or many fields at aonce.

Parameters

- **path** (*str*) – String of the path
- **tokens** (*dict*) – A dictionary of tokens to search through.

Returns A modified string

- genindex
- modindex
- search
- genindex
- modindex
- search

Python Module Index

a

c

v

Index

Symbols

A

C

E

F

H

R

S

T