tapioca-wrapper Documentation

Release 2.3

Filipe Ximenes

Jun 14, 2017

Contents

Contents:

CHAPTER 1

About

tapioca-wrapper provides an easy way to make explorable python API wrappers. APIs wrapped by Tapioca follow a simple interaction pattern that works uniformly so developers don't need to learn how to use a new coding interface/style for each service API.

CHAPTER 2

Quickstart

Using a tapioca package

Yes, you are in the right place

There is a good chance you found this page because you clicked a link from some python package called **tapioca-SOMETHING**. Well, welcome! You are in the right place. This page will teach you the basics of how to use the package that sent you here. If you didn't arrive here from another package, then please keep reading. The concepts learned here apply to any tapioca-**package** available.

What's tapioca?

tapioca is an *API wrapper maker*. It helps Python developers creating packages for APIs (like the *Facebook Graph API* or the *Twitter REST API*. You can find a full list of available API packages made with tapioca *here*.

All wrappers made with tapioca follow a simple interaction pattern that works uniformly, so once you learn how tapioca works, you will be able to work with any tapioca package available.

Getting started

We will use `tapioca-facebook` as example to guide us through tapioca concepts. Let's install `tapioca-facebook`:

```
$ pip install tapioca-facebook
```

To better experience tapioca, we will also use IPython:

```
$ pip install ipython
```

Let's explore!

Go to https://developers.facebook.com/tools/explorer/, click "Get Access Token", select all "User Data Permissions" and "Extended Permissions", and click "Get Access Token". This will give you a temporary access token to play with Facebook API. In case it expires, just generate a new one.

TapiocaClient object

This is how you initialize your tapioca client:

```
from tapioca_facebook import Facebook

api = Facebook(access_token='{your_genereated_access_token}')
```

If you are using IPython, you can now list available endpoints by typing `api.` and pressing `tab`.

```
>>> api.
api.user_likes                  api.page_blocked                api.page_locations
api.page_statuses                api.user_applications_developer  api.user_friends
api.user_invitable_friends       api.user_photos                  api.user_videos
api.object                       api.page_conversations           api.page_milestones
...
```

Resources

Those are the available endpoints for the Facebook API. As we can see, there is one called `user_likes`. Let's take a closer look.

Type `api.user_likes?` and press `enter`.

```
In [3]: api.user_likes?
...
Docstring:
Automatic generated __doc__ from resource_mapping.
Resource: {id}/likes
Docs: https://developers.facebook.com/docs/graph-api/reference/v2.2/user/likes
```

As we can see, the `user_likes` resource requires an `id` to be passed to the URL. Let's do it:

```
api.user_likes(id='me')
```

Fetching data

To request the current user likes, its easy:

```
likes = api.user_likes(id='me').get()
```

To print the returned data:

```
In [9]: likes().data
OUT [9]: {
        'data': [...],
        'paging': {...}
}
```

Exploring data

We can also explore the returned data using the IPython `tab` auto-complete:

```
In [9]: likes.
likes.data    likes.paging
```

Iterating over data

You can iterate over returned data:

```
likes = api.user_likes(id='me').get()

for like in likes.data:
        print(like.id().data)
```

Items passed to the `for` loop will be wrapped in tapioca so you still have access to all features.

TapiocaClientExecutor object

Whenever you make a "call" on a `TapiocaClient`, it will return an `TapiocaClientExecutor` object. You will use the executor every time you want to perform an action over data you possess.

We did this already when we filled the URL parameters for the `user_like` resource (calling it and passing the argument `id='me'`). In this new object, you will find many methods to help you play with the data available.

Here is the list of the methods available in a `TapiocaClientExecutor`:

Making requests

Tapioca uses the requests library to make requests so HTTP methods will work just the same (get()/post()/put()/delete()/head()/options()). The only difference is that we don't need to pass a URL since tapioca will take care of this.

```
likes = api.user_likes(id='me').get()
```

URL params

To pass query string parameters in the URL, you can use the `` `params` `` parameter:

```
likes = api.user_likes(id='me').get(
        params={'limit': 5})
```

This will return only 5 results.

Body data

If you need to pass data in the body of your request, you can use the `` `data` `` parameter. For example, let's post a message to a Facebook wall:

```
# this will only work if you have a post to wall permission
api.user_feed(id='me').post(
        data={'message': 'I love tapiocas!! S2'})
```

Please read requests for more detailed information about how to use HTTP methods.

Accessing raw data

Use `data` to return data contained in the Tapioca object.

```
>>> likes = api.user_likes(id='me').get()
>>> likes().data
{
        'data': [...],
        'paging': {...}
}
>>> this will print only the array contained
>>> # in the 'data' field of the response
>>> likes.data().data
>>> [...]
```

Dynamically fetching pages

Many APIs use a paging concept to provide large amounts of data. This way, data is returned in multiple requests to avoid a single long request. Tapioca is built to provide an easy way to access paged data using the `pages()` method:

```
likes = api.user_likes(id='me').get()

for like in likes().pages():
        print(like.name().data)
```

This will keep fetching user likes until there are none left. Items passed to the `for` loop will be wrapped in tapioca so you still have access to all features.

This method also accepts `max_pages` and `max_items` parameters. If both parameters are used, the `for` loop will stop after `max_pages` are fetched or `max_items` are yielded, whichever comes first:

```
for item in resp().pages(max_pages=2, max_items=40):
        print(item)
# in this example, the for loop will stop after two pages are fetched or 40 items are␣
↪yielded,
# whichever comes first.
```

Accessing wrapped data attributes

It's possible to access wrapped data attributes on executor. For example, it's possible to reverse a wrapped list:

```
likes = api.user_likes(id='me').get()

likes_list = likes.data
likes_list().reverse()
# items in the likes_list are now in reverse order
# but still wrapped in a tapioca object
```

Opening documentation in the browser

If you are accessing a resource, you can call `open_docs` to open the resource documentation in a browser:

```
api.user_likes().open_docs()
```

Opening any link in the browser

Whenever the data contained in a tapioca object is a URL, you can open it in a browser by using the `open_in_browser()` method.

For more information on what wrappers are capable of, please refer to the *features* section.

CHAPTER 3

Features

Here are some features tapioca supports. The wrapper you are using may support them or not, it will depend on the *tapioca-wrapper* version it is tied to and if the developer implemented the methods needed to support the feature. Either way, if you find yourself in a situation where you need one of these features, clone the wrapper, update the *tapioca-wrapper* version to the latest one, implement the features you need and submit a pull request to the developer. You will be helping a lot of people!

TapiocaClient

The first object you get after you instanciate a tapioca wrapper is an instance of the `TapiocaClient` class. This class is capable of accessing the API endpoints of the wrapper and traversing respose objects. No other action besides those can be achieved from a `TapiocaClient`. To retrieve the raw data returned from the API call you will need to transform it in a `TapiocaClientExecutor`.

TODO: add examples

Default URL params

Sometimes URLs templates need parameters that will be repeated across all call. For example, your user id:

```
http://www.someapi.com/{user_id}/resources/
http://www.someapi.com/{user_id}/resources/{resource_id}/
http://www.someapi.com/{user_id}/other-resources/{other_id}/
```

In this cases you can instantiate the wrapper passing a `default_url_params` parameter, and they will be used automatically to fill URL templates.

```
cli = MyWrapper(access_token='some_token', default_url_params={'user_id': 123456})
cli.resources() # http://www.someapi.com/123456/resources/
```

Using an existing requests.Session

Requests provides access to a number of advanced features by letting users maintain a Session object.

To use these features you can create a `TapiocaClient` with an existing session by passing it to the new client as the `session` paramter:

```
session = requests.Session()
cli = MyWrapper(access_token='some_token', session=session)
    cli.resources() # http://www.someapi.com/123456/resources/
```

This allows us to perform some interesting operations without having to support them directly in `TapiocaClient`. For example caching for github requests using cachecontrol:

```
from cachecontrol import CacheControl
from cachecontrol.caches import FileCache
import requests
import tapioca_github

session = CacheControl(requests.Session(), cache=FileCache('webcache'))
gh = tapioca_github.Github(client_id='some_id', access_token='some_token',␣
↪session=session)
response  = gh.repo_single(owner="vintasoftware", repo="tapioca-wrapper").get()
repo_data = response().data
```

This will cache the E-tags provided by github to the folder *webcache*.

TapiocaClientExecutor

Everytive you `call` in `TapiocaClient` you will get a `TapiocaClientExecutor`. Here are the features available in a `TapiocaClientExecutor`:

Accessing raw response data

To access the raw data contained in the executor, use the `data` **attribute**. To access the raw response, use the `response` **attribute**. To access the status code of the response, use the `status_code` **attribute**. If during the request the `Auth refreshing` process was executed, the returned value from it will be accessible in the `refresh_data` **attribute**.

TODO: add examples

HTTP calls

Executors have access to make HTTP calls using the current data it possesses as the URL. requests library, is used as the engine to perform API calls. Every key word parameter you pass to: `get()`, `post()`, `put()`, `patch()`, `delete()` methods will be directly passed to the request library call. This means you will be using `params={'myparam': 'paramvalue'}` to send querystring arguments in the url and `data={'datakey': 'keyvalue'}` to send data in the body of the request.

TODO: add examples

Auth refreshing (*)

Some clients needs to update its token once they have expired. If the clients supports, you might instantiate it passing `refresh_token_by_default=True` or make any HTTP call passing `refresh_auth=True` (both defaults to `False`). Note that if your client instance have `refresh_token_by_default=True`, then you don't need to explicity set it on HTTP calls.

TODO: add examples

*the wrapper you are current using may not support this feature

Pagination (*)

Use `pages()` method to call an endpoint that returns a collection of objects in batches. This will make your client automatically fetch more data untill there is none more left. You may use `max_pages` and/or `max_items` to limit the number of items you want to iterate over.

TODO: add examples

*the wrapper you are current using may not support this feature

Open docs (*)

When accessing an endpoint, you may want to read it's documentation in the internet. By calling `open_docs()` in a python interactive session, the doc page will be openned in a browser.

TODO: add examples

*the wrapper you are current using may not support this feature

Open in the browser (*)

Whenever the data contained in the executor is a URL, you can directly open it in the browser from an interactive session by calling `open_in_browser()`

TODO: add examples

*the wrapper you are current using may not support this feature

Exceptions

Tapioca built in exceptions will help you to beautifuly catch and handle whenever there is a client or server error. Make sure the wrapper you are using correctly raises exceptions, the developer might not have treated this. Please refer to the *exceptions* for more information about exceptions.

Serializers

Serializers will help you processing data before it is sent to the endpoint and transforming data from responses into python objects. Please refer to the *serializers* for more information about serializers.

CHAPTER 4

Serializers

Serializer classes are capable of performing serialization and deserialization of data.

Serialization is the transformation of data in a native format (in our case Python data types) into a serialized format (e.g. text). For example, this could be transforming a native Python `Datetime` instance containing a date into a string.

Deserialization is the transformation of data in a serialized format (e.g. text) into a native format. For example, this could be transforming a string containing a date into a native Python `Datetime` instance.

Usage

Serialization

Data serialization is done in the background when tapioca is executing the request. It will traverse any data structure passed to the `data` param of the request and convert Python data types into serialized types.

```
>>> reponse = cli.the_resource().post(data={'date': datetime.today()})
```

In this example, `datetime.today()` will be converted into a string formatted date just before the request is executed.

Deserialization

To deserialize data, you need to transform your client into an executor and then call a deserialization method from it:

```
>>> reponse = cli.the_resource().get()
>>> print(response.created_at())
<TapiocaClientExecutor object
2015-10-25T22:34:51+00:00>
>>> print(respose.created_at().to_datetime())
2015-10-25 22:34:51+00:00
```

```
>>> print(type(respose.created_at().to_datetime()))
datetime.datetime
```

Swapping the default serializer

Clients might have the default `SimpleSerializer`, some custom serializer designed by the wrapper creator, or even no serializer. Either way, you can swap it for one of your own even if you were not the developer of the library. For this, you only need to pass the desired serializer class during the client initialization:

```
from my_serializers import MyCustomSerializer

cli = MyServiceClient(
        access_token='blablabla',
        serializer_class=MyCustomSerializer)
```

Built-ins

class `SimpleSerializer`

`SimpleSerializer` is a very basic and generic serializer. It is included by default in adapters unless explicitly removed. It supports serialization from *Decimal* and *datetime* and deserialization methods to those two types as well. Here is it's full code:

```
class SimpleSerializer(BaseSerializer):

def to_datetime(self, value):
        return arrow.get(value).datetime

def to_decimal(self, value):
        return Decimal(value)

def serialize_decimal(self, data):
        return str(data)

def serialize_datetime(self, data):
        return arrow.get(data).isoformat()
```

As you can see, `datetime` values will be formatted to iso format.

Writing a custom serializer

To write a custom serializer, you just need to extend the `BaseSerializer` class and add the methods you want. But you can also extend from `SimpleSerializer` to inherit its functionalities.

Serializing

To allow serialization of any desired data type, add a method to your serializer named using the following pattern: `serialize_ + name_of_your_data_type_in_lower_case`. For example:

```
class MyCustomDataType(object):
        message = ''

class MyCustomSerializer(SimpleSerializer):

        def serialize_mycustomdatatype(self, data):
                return data.message
```

Deserializing

Any method starting with `to_` in your custom serializer class will be available for data deserialization. It also accepts key word arguments.

```
from tapioca.serializers import BaseSerializer

class MyCustomSerializer(BaseSerializer):

        def to_striped(self, value, **kwargs):
                return value.strip()
```

Here's a usage example for it:

```
from my_serializers import MyCustomSerializer

cli = MyServiceClient(
        access_token='blablabla',
        serializer_class=MyCustomSerializer)

response = cli.the_resource().get()

striped_data = response.the_data().to_striped()
```

CHAPTER 5

Exceptions

Catching API errors

Tapioca supports 2 main types of exceptions: `ClientError` and `ServerError`. The default implementation raises `ClientError` for HTTP response 4xx status codes and `ServerError` for 5xx status codes. Since each API has its own ways of reporting errors and not all of them follow HTTP recommendations for status codes, this can be overriden by each implemented client to reflect its behaviour. Both of these exceptions extend `TapiocaException` which can be used to catch errors in a more generic way.

class `TapiocaException`

Base class for tapioca exceptions. Example usage:

```
from tapioca.exceptions import TapiocaException

try:
        cli.fetch_something().get()
except TapiocaException, e:
        print("API call failed with error %s", e.status)
```

You can also access a tapioca client that contains response data from the exception:

```
from tapioca.exceptions import TapiocaException

try:
        cli.fetch_something().get()
except TapiocaException, e:
        print(e.client.error_message().data)
```

class `ClientError`

Default exception for client errors. Extends from `TapiocaException`.

class `ServerError`

Default exception for server errors. Extends from `TapiocaException`.

CHAPTER 6

Flavours

Available Flavours

Facebook

https://github.com/vintasoftware/tapioca-facebook

Twitter

https://github.com/vintasoftware/tapioca-twitter

Mandrill

https://github.com/vintasoftware/tapioca-mandrill

Parse

https://github.com/vintasoftware/tapioca-parse

Bitbucket

https://github.com/vintasoftware/tapioca-bitbucket

Disqus

https://github.com/marctc/tapioca-disqus

Harvest

https://github.com/vintasoftware/tapioca-harvest

CrunchBase

https://github.com/vu3jej/tapioca-crunchbase

Otter

https://github.com/vu3jej/tapioca-otter

GitHub

https://github.com/gileno/tapioca-github

Meetup

https://github.com/lightstrike/tapioca-meetup

Toggl

https://github.com/hackebrot/tapioca-toggl

Braspag

https://github.com/parafernalia/tapioca_braspag

Iugu

https://github.com/solidarium/tapioca-iugu

Instagram

https://github.com/vintasoftware/tapioca-instagram

Youtube

https://github.com/vintasoftware/tapioca-youtube

Your flavour

To create a new wrapper, please refer to *Building a wrapper*. Upload it to pypi and send a pull request here for it to be added to the list.

CHAPTER 7

Building a wrapper

Wrapping an API with Tapioca

The easiest way to wrap an API using tapioca is starting from the cookiecutter template.

To use it, install cookiecutter in your machine:

```
pip install cookiecutter
```

and then use it to download the template and run the config steps:

```
cookiecutter gh:vintasoftware/cookiecutter-tapioca
```

After this process, it's possible that you have a ready to go wrapper. But in most cases, you will need to customize stuff. Read through this document to understand what methods are available and how your wrapper can make the most of tapioca. Also, you might want to take a look in the source code of *other wrappers* to get more ideas.

In case you are having any difficulties, seek help on Gitter or send an email to contact@vinta.com.br .

Adapter

Tapioca features are mainly implemented in the `TapiocaClient` and `TapiocaClientExecutor` classes. Those are generic classes common to all wrappers and cannot be customized to specific services. All the code specific to the API wrapper you are creating goes in your adapter class, which should inherit from `TapiocaAdapter` and implement specific behaviours to the service you are working with.

Take a look in the generated code from the cookiecutter or in the tapioca-facebook adapter to get a little familiar with it before you continue. Note that at the end of the module you will need to perform the transformation of your adapter into a client:

```
Facebook = generate_wrapper_from_adapter(FacebookClientAdapter)
```

Plase refer to the *TapiocaAdapter class* document for more information on the available methods.

Features

Here is some information you should know when building your wrapper. You may choose to or not to support features marked with *(optional)*.

Resource Mapping

The resource mapping is a very simple dictionary which will tell your tapioca client about the available endpoints and how to call them. There's an example in your cookiecutter generated project. You can also take a look at tapioca-facebook's resource mapping.

Tapioca uses requests to perform HTTP requests. This is important to know because you will be using the method `get_request_kwargs` to set authentication details and return a dictionary that will be passed directly to the request method.

Formatting data

Use the methods `format_data_to_request` and `response_to_native` to correctly treat data leaving and being received in your wrapper.

TODO: add examples

You might want to use one of the following mixins to help you with data format handling in your wrapper:

- `FormAdapterMixin`
- `JSONAdapterMixin`
- `XMLAdapterMixin`

Exceptions

Overwrite the `process_response` method to identify API server and client errors raising the correct exception accordingly. Please refer to the *exceptions* for more information about exceptions.

TODO: add examples

Pagination (optional)

`get_iterator_list` and `get_iterator_next_request_kwargs` are the two methods you will need to implement for the executor `pages()` method to work.

TODO: add examples

Serializers (optional)

Set a `serializer_class` attribute or overwrite the `get_serializer()` method in your wrapper for it to have a default serializer.

```
from tapioca import TapiocaAdapter
from tapioca.serializers import SimpleSerializer

class MyAPISerializer(SimpleSerializer):
```

```
        def serialize_datetime(self, data):
                return data.isoformat()


class MyAPIAdapter(TapiocaAdapter):
        serializer_class = MyAPISerializer
        ...
```

In the example, every time a `datetime` is passed to the parameters of an HTTP method, it will be converted to an ISO formatted `string`.

It's important that you let people know you are providing a serializer, so make sure you have it documented in your *README*.

```
## Serialization
- datetime
- Decimal

## Deserialization
- datetime
- Decimal
```

Please refer to the *serializers* for more information about serializers.

Refreshing Authentication (optional)

You can implement the `refresh_authentication` and `is_authentication_expired` methods in your Tapioca Client to refresh your authentication token every time it expires.

`is_authentication_expired` receives an error object from the request method (it contains the server response and HTTP Status code). You can use it to decide if a request failed because of the token. This method should return `True` if the authentication is expired or `False` otherwise (default behavior).

`refresh_authentication` receives `api_params` and should perform the token refresh protocol. If it is successfull it should return a truthy value (the original request will then be automatically tried). If the token refresh fails, it should return a falsy value (and the the original request wont be retried).

Once these methods are implemented, the client can be instantiated with `refresh_token_by_default=True` (or pass `refresh_token=True` in HTTP calls) and `refresh_authentication` will be called automatically.

```
    def is_authentication_expired(self, exception, *args, **kwargs):
            ....

def refresh_authentication(self, api_params, *args, **kwargs):
    ...
```

XMLAdapterMixin Configuration (only if required)

Additionally, the XMLAdapterMixin accepts configuration keyword arguments to be passed to the xmltodict library during parsing and unparsing by prefixing the xmltodict keyword with `xmltodict_parse__` or `xmltodict_unparse` respectively. These parameters should be configured so that the end-user has a consistent experience across multiple Tapioca wrappers irrespective of various API requirements from wrapper to wrapper.

Note that the end-user should **not** need to modify these keyword arguments themselves. See xmltodict docs and source for valid parameters.

Users should be able to construct dictionaries as defined by the xmltodict library, and responses should be returned in the canonical format.

Example XMLAdapterMixin configuration keywords:

```
class MyXMLClientAdapter(XMLAdapterMixin, TapiocaAdapter):
    ...
    def get_request_kwargs(self, api_params, *args, **kwargs):
        ...
        # omits XML declaration when constructing requests from dictionary
        kwargs['xmltodict_unparse__full_document'] = False
        ...
```

CHAPTER 8

TapiocaAdapter class

class TapiocaAdapter

Attributes

api_root

This should contain the base URL that will be concatenated with the resource mapping itens and generate the final request URL. You can either set this attribute or use the `get_api_root` method.

serializer_class

For more information about the `serializer_class` attribute, read the *serializers documentation*.

Methods

get_api_root (*self*, *api_params*)

This method can be used instead of the `api_root` attribute. You might also use it to decide which base URL to use according to a user input.

```
def get_api_root(self, api_params):
        if api_params.get('development'):
                return 'http://api.the-dev-url.com/'
        return 'http://api.the-production-url.com/'
```

get_request_kwargs (*self*, *api_params*, **args*, ***kwargs*)

This method is called just before any request is made. You should use it to set whatever credentials the request might need. The **api_params** argument is a dictionary and has the parameters passed during the initialization of the tapioca client:

```
cli = Facebook(access_token='blablabla', client_id='thisistheis')
```

For this example, api_params will be a dictionary with the keys `access_token` and `client_id`.

Here is an example of how to implement Basic Auth:

```
from requests.auth import HTTPBasicAuth

class MyServiceClientAdapter(TapiocaAdapter):
        ...
        def get_request_kwargs(self, api_params, *args, **kwargs):
                params = super(MyServiceClientAdapter, self).get_request_kwargs(
                        api_params, *args, **kwargs)

                params['auth'] = HTTPBasicAuth(
                        api_params.get('user'), api_params.get('password'))

                return params
```

process_response(*self*, *response*)

This method is responsible for converting data returned in a response to a dictionary (which should be returned). It should also be used to raise exceptions when an error message or error response status is returned.

format_data_to_request(*self*, *data*)

This converts data passed to the body of the request into text. For example, if you need to send JSON, you should use `json.dumps(data)` and return the response. **See the mixins section above.**

response_to_native(*self*, *response*)

This method receives the response of a request and should return a dictionay with the data contained in the response. **see the mixins section above.**

get_iterator_next_request_kwargs(*self*, *iterator_request_kwargs*, *response_data*, *response*)

Override this method if the service you are using supports pagination. It should return a dictionary that will be used to fetch the next batch of data, e.g.:

```
def get_iterator_next_request_kwargs(self,
                iterator_request_kwargs, response_data, response):
        paging = response_data.get('paging')
        if not paging:
                return
        url = paging.get('next')

        if url:
                iterator_request_kwargs['url'] = url
                return iterator_request_kwargs
```

In this example, we are updating the URL from the last call made. `iterator_request_kwargs` contains the paramenters from the last call made, `response_data` contains the response data after it was parsed by `process_response` method, and `response` is the full response object with all its attributes like headers and status code.

get_iterator_list(*self*, *response_data*)

Many APIs enclose the returned list of objects in one of the returned attributes. Use this method to extract and return only the list from the response.

```
def get_iterator_list(self, response_data):
        return response_data['data']
```

In this example, the object list is enclosed in the `data` attribute.

is_authentication_expired(*self*, *exception*, **args*, ***kwargs*)

Given an exception, checks if the authentication has expired or not. If so and `refresh_token_by_default=True` or the HTTP method was called with `refresh_token=True`, then it will automatically call `refresh_authentication` method and retry the original request.

If not implemented, `is_authentication_expired` will assume `False`, `refresh_token_by_default` also defaults to `False` in the client initialization.

refresh_authentication(self, api_params, *args, **kwargs):

Should do refresh authentication logic. Make sure you update *api_params* dictionary with the new token. If it successfully refreshs token it should return a truthy value that will be stored for later access in the executor class in the `refresh_data` attribute. If the refresh logic fails, return a falsy value. The original request will be retried only if a truthy is returned.

CHAPTER 9

Contributors

Thanks!

- Filipe Ximenes (filipeximenes@gmail.com)
- André Ericson (de.ericson@gmail.com)
- Luiz Sotero (luizsotero@gmail.com)
- Elias Granja Jr (contato@eliasgranja.com)
- Rômulo Collopy (romulocollopy@gmail.com)

CHAPTER 10

Changelog

1.4

- Adds support to Session requests

1.3

- `refresh_authentication` should return data about the refresh token process
- If a falsy value is returned by `refresh_authentication` the request wont be retried automatically
- Data returned by `refresh_authentication` is stored in the tapioca class and can be accessed in the executor via the attribute `refresh_data`

1.2.3

- `refresh_token_by_default` introduced to prevent passing `refresh_token` on every request.

1.1.10

- Fixed bugs regarding `request_kwargs` passing over calls
- Fixed bugs regarding external `serializer` passing over calls
- Wrapper instatiation now accepts `default_url_params`

1.1

- Automatic refresh token support
- Added Python 3.5 support
- Added support for `OrderedDict`
- Documentation cleanup

1.0

- Data serialization and deserialization
- Access CamelCase attributes using snake_case
- Dependencies are now tied to specific versions of libraries
- `data` and `response` are now attributes instead of methods in the executor
- Added `status_code` attribute to tapioca executor
- Renamed `status` exception attribute to `status_code`
- Fixed return for `dir` call on executor, so it's lot easier to explore it
- Multiple improvments to documentation

0.6.0

- Giving access to request_method in `get_request_kwargs`
- Verifying response content before trying to convert it to json on `JSONAdapterMixin`
- Support for `in` operator
- pep8 improvments

0.5.3

- Adding `max_pages` and `max_items` to `pages` method

0.5.1

- Verifying if there's data before json dumping it on `JSONAdapterMixin`

0.5.0

- Automatic pagination now requires an explicit `pages()` call
- Support for `len()`

- Attributes of wrapped data can now be accessed via executor
- It's now possible to iterate over wrapped lists

0.4.1

- changed parameters for Adapter's `get_request_kwargs`. Also, subclasses are expected to call `super`.
- added mixins to allow adapters to easily choose witch data format they will be dealing with.
- `ServerError` and `ClientError` are now raised on 4xx and 5xx response status. This behaviour can be customized for each service by overwriting adapter's `process_response` method.

Index