
aiotapioca-wrapper Documentation

Release 4.2.0

Filipe Ximenes

Sep 19, 2022

CONTENTS

1	About	3
2	Quickstart	5
2.1	Using a aiotapioca package	5
2.2	What's aiotapioca?	5
2.3	Getting started	5
2.4	TapiocaClient object	6
2.5	TapiocaClientExecutor object	7
3	Features	11
3.1	TapiocaClient	11
3.2	TapiocaClientExecutor	12
3.3	Exceptions	15
3.4	Serializers	15
4	Serializers	17
4.1	Usage	17
4.2	Built-ins	18
4.3	Writing a custom serializer	18
5	Exceptions	21
5.1	Catching API errors	21
6	Flavours	23
6.1	Available Flavours	23
6.2	Your flavour	25
7	Building a wrapper	27
7.1	Wrapping an API with AioTapioca	27
7.2	Adapter	27
7.3	Features	28
8	TapiocaAdapter class	31
8.1	Attributes	31
8.2	Methods	31
9	Contributors	35
9.1	Thanks!	35
10	Release v4.2.0 (2022-10-19)	37

11	Release v4.1.1 (2022-06-21)	39
12	Release v4.0.2 (2022-06-17)	41
13	Release v4.0.1 (2022-06-15)	43
14	Release v3.8.0 (2022-05-15)	45
15	Release v3.7.0 (2022-04-27)	47
16	Release v3.6.0 (2022-04-22)	49
17	Release v3.5.0 (2022-04-12)	51
18	Release v3.4.2 (2022-04-08)	53
19	Release v3.4.1 (2022-04-08)	55
20	Release v3.4.0 (2022-04-0)	57
21	Release v3.3.1 (2022-03-24)	59
22	Release v3.3.0 (2022-03-24)	61
23	Release v3.2.4 (2022-03-23)	63
24	Release v3.2.3 (2022-03-22)	65
25	Release v3.2.2 (2022-03-22)	67
26	Release v3.2.1 (2022-03-22)	69
27	Release v3.2.0 (2022-03-22)	71
28	Release v3.1.1 (2022-03-21)	73
29	Release v3.1.0 (2022-03-21)	75
30	Release v3.0.0 (2022-03-21)	77
31	Release v2.1.0 (2022-03-19)	79
32	Release v2.0.2 (2022-02-25)	81
33	Release v2.0.1 (2020-01-25)	83
34	Release v2.0.0 (2020-01-25)	85
35	Release v1.5.1 (2019-04-19)	87
36	Release v1.5.0 (2019-04-19)	89
37	Release v1.4.3 (2017-06-15)	91
38	Release v1.4.1 (2017-05-25)	93
39	Release v1.4.0 (2017-03-28)	95
40	Release v1.3.0 (2017-01-20)	97

41	Release v1.2.3 (2016-09-28)	99
42	Release v1.2.2 (2016-04-23)	101
43	Release v1.2.1 (2016-01-02)	103
44	Release v1.1.12 (2016-05-31)	105
45	Release v1.1.11 (2016-05-31)	107
46	Release v1.1.10 (2016-03-27)	109
47	Release v1.1.9 (2016-03-27)	111
48	Release v1.1.8 (2016-03-27)	113
49	Release v1.1.7 (2016-03-27)	115
50	Release v1.1.6 (2016-02-29)	117
51	Release v1.1.4 (2016-02-27)	119
52	Release v1.1.0 (2016-02-27)	121
53	Release v1.0.0 (2015-11-10)	123
54	Release v0.6.0 (2015-09-23)	125
55	Release v0.5.3 (2015-04-10)	127
56	Release v0.5.1 (2015-08-06)	129
57	Release v0.5.0 (2015-08-05)	131
58	Release v0.4.1 (2015-08-01)	133
	Index	135

Contents:

ABOUT

aiotapioca-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.

aiotapioca-wrapper is an asynchronous fork of the [tapioca-wrapper](#) library.



QUICKSTART

2.1 Using a aiotapioca 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 **aiotapioca-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 aiotapioca-**package** available.

2.2 What's aiotapioca?

aiotapioca 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 aiotapioca *here*.

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

2.3 Getting started

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

```
$ pip install aiotapioca-facebook
```

To better experience aiotapioca, 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.

2.4 TapiocaClient object

This is how you initialize your aiotapioca client:

```
from aiotapioca_facebook import Facebook

async with Facebook(access_token='{your_generated_access_token}') as api:
    ...
# or

api = Facebook(access_token='{your_generated_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
...
```

2.4.1 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')
```

2.4.2 Fetching data

To request the current user likes, its easy:

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

To print the returned data:

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

(continues on next page)

(continued from previous page)

```
'paging': {...}
}
```

2.4.3 Exploring data

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

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

2.4.4 Iterating over data

You can iterate over returned data:

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

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

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

2.5 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`:

2.5.1 Making requests

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

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

URL params

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

```
likes = await 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
await api.user_feed(id='me').post(
    data={'message': 'I love tapiocas!! S2'})
```

Please read [aiohttp](#) for more detailed information about how to use HTTP methods.

Multiple requests

To perform multiple requests asynchronously, you can use batch methods (`post_batch()/patch_batch()/put_batch()/delete_batch()`):

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

2.5.2 Accessing raw data

Use `data()` to return data contained in the `ProcessData` object.

```
>>> likes = await 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()
>>> [...]
```

2.5.3 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 = await api.user_likes(id='me').get()

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

This will keep fetching user likes until there are none left. Items passed to the `for` loop will be wrapped in `aiotapioca` 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:

```
async 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.
```

2.5.4 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 = await 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 aiotapioca object
```

2.5.5 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()
```

2.5.6 Opening any link in the browser

Whenever the data contained in a `aiotapioca` 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.

FEATURES

Here are some features *aiotapioca* supports. The wrapper you are using may support them or not, it will depend on the *aiotapioca-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 *aiotapioca-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!

3.1 TapiocaClient

The first object you get after you instantiate a *aiotapioca* wrapper is an instance of the `TapiocaClient` class. This class is capable of accessing the API endpoints of the wrapper and traversing response 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

3.1.1 Default URL params

Sometimes URLs templates need parameters that will be repeated across all API calls. For example, an 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/
```

3.1.2 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` parameter:

```
session = aiohttp.ClientSession()
async with MyWrapper(access_token='some_token', session=session) as cli:
    cli.resources() # http://www.someapi.com/123456/resources/
```

This allows us to perform some interesting operations without having to support them directly in `TapiocaClient` and instantiate it using the `async with` construct. 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="ilindrey", repo="aiotapioca-wrapper").get()
repo_data = response().data
```

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

3.2 TapiocaClientExecutor

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

3.2.1 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 attribute`.

```
cli = MyWrapper(access_token='some_token')

response = await cli.some_resource().get()

data = response.data()
response = response.response # return aiohttp.ClientResponse
status = response.status
```

3.2.2 HTTP calls

Executors have access to make HTTP calls using the current data it possesses as the URL. The `aiohttp` 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.

```
cli = MyWrapper()
response = await cli.some_resource().get(params={'myparam': 'paramvalue'})
response = await cli.some_resource().post(data={'datakey': 'keyvalue'})
response = await cli.some_resource().delete(data={'id': 123})
```

For perform multiple requests asynchronously, you can use batch methods as like a `post_batch()`, `patch_batch()`, `put_batch()`, `delete_batch()`. The data in the list must be passed to the data parameter in order to execute requests.

```
cli = MyWrapper()
response = await cli.some_resource().post_batch(data=[
    {'datakey': 'keyvalue1'},
    {'datakey': 'keyvalue2'},
])
```

3.2.3 Auth refreshing (*)

Some clients need to update its token once they have expired. If the client supports this feature, you might specify `refresh_token=True` in the adapter class, instantiate it passing `refresh_token=True` or make any HTTP call passing `refresh_auth=True` (both default to `False`). Note that if your adapter class or client instance has `refresh_token=True`, then you don't need to explicitly set it on HTTP calls.

```
class MyAPIAdapter(TapiocaAdapter):
    refresh_token=True
    ...

MyWrapper = generate_wrapper_from_adapter(MyAPIAdapter)

# or

cli = MyWrapper(refresh_token=True)
...

# or

cli = MyWrapper()
response = await cli.some_resource().post(refresh_token=True)
...
```

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

3.2.4 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 until 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.

```
cli = MyWrapper():

response = await cli.some_resource().get(params=...)

async for page in response().pages():
    print(page.data())
    print(page.response)
    ...

# or

async for page in response().pages(max_pages=2):
    ...

# or

async for page in response().pages(max_items=10):
    ...

# or

async for page in response().pages(max_pages=2, max_items=10):
    ...
```

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

3.2.5 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 opened in a browser.

```
cli = MyWrapper()
cli.some_resource.open_docs()
```

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

3.2.6 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()`

```
cli = MyWrapper()
response = await cli.some_resource().get()
response.data.url.open_in_browser()
```

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

3.3 Exceptions

AioTapioca built in exceptions will help you to beautifully 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.

3.4 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.

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.

4.1 Usage

4.1.1 Serialization

Data serialization is done in the background when `aiotapioca` 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.

```
>>> response = await 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.

4.1.2 Deserialization

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

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

4.1.3 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)
```

4.2 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 its full code:

```
class SimpleSerializer(BaseSerializer):

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

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

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

4.3 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.

4.3.1 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
```


4.3.2 Deserializing

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

```
from aiotapioca.serializers import BaseSerializer

class MyCustomSerializer(BaseSerializer):

    def to_stripped(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 = await cli.the_resource().get()

stripped_data = response.the_data().to_stripped()
```


EXCEPTIONS

5.1 Catching API errors

AioTapioca 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 overridden 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 aiotapioca exceptions. Example usage:

```
from aiotapioca.exceptions import TapiocaException

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

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

```
from aiotapioca.exceptions import TapiocaException

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

class ClientError

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

class ServerError

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

FLAVOURS

6.1 Available Flavours

6.1.1 aiotapioca-wrapper library

Facebook

<https://github.com/ilindrey/aiotapioca-facebook>

Yandex Metrika

<https://github.com/ilindrey/aiotapioca-yandex-metrika>

6.1.2 tapioca-wrapper library

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/parafernaliala/tapioca_braspag

Iugu

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

Instagram

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

Youtube

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

Asana

<https://github.com/henriquebastos/tapioca-asana>

Desk

<https://github.com/medeirosthiago/tapioca-desk>

Mailgun

<https://github.com/vintasoftware/tapioca-mailgun>

Discourse

<https://github.com/humrochagf/tapioca-discourse>

StatusPage

<https://github.com/olist/tapioca-statuspage>

Trello

<https://github.com/humrochagf/tapioca-trello>

6.2 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.

BUILDING A WRAPPER

7.1 Wrapping an API with AioTapioca

The easiest way to wrap an API using `aiotapioca` 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.

7.2 Adapter

`AioTapioca` 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 [aiotapioca-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)
```

Please refer to the [TapiocaAdapter class](#) document for more information on the available methods.

7.3 Features

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

7.3.1 Resource Mapping

The resource mapping is a very simple dictionary which will tell your aiotapioca 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 [aiotapioca-facebook's resource mapping](#).

AioTapioca uses [aiohttp](#) 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.

7.3.2 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`
- `PydanticAdapterMixin`

7.3.3 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

7.3.4 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

7.3.5 Serializers (optional)

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

```
from aiotapioca import TapiocaAdapter
from aiotapioca.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.

7.3.6 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 successful 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=True` (or pass `refresh_token=True` in HTTP calls) and `refresh_authentication` will be called automatically. You can also specify `refresh_token=True` in the adapter class.

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

(continues on next page)

(continued from previous page)

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

7.3.7 XMLAdapterMixin Configuration (only if required)

Additionally, the XMLAdapterMixin accepts configuration keyword arguments to be passed to the xmldict library during parsing and unparsing by prefixing the xmldict keyword with `xmldict_parse__` or `xmldict_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 xmldict [docs](#) and [source](#) for valid parameters.

Users should be able to construct dictionaries as defined by the xmldict 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['xmldict_unparse__full_document'] = False  
        ...
```

TAPIOCAADAPTER CLASS

`class TapiocaAdapter`

8.1 Attributes

api_root

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

resource_mapping

A dictionary containing a mapping of the resources. For example:

You can either set this attribute or use the `get_resource_mapping` method.

serializer_class

For more information about the `serializer_class` attribute, read the [serializers documentation](#).

refresh_token

If the HTTP method was called with `refresh_token=True`, then it will automatically call `refresh_authentication` method and retry the original request.

semaphore

For more information about the `semaphore` attribute, read the [Python's standard library](#). Default value **10**.

8.2 Methods

get_resource_mapping(*self*, *api_params*, ***kwargs*)

This method can be used instead of the `resource_mapping` attribute. Returns the `resource_mapping` attribute by default.

get_api_root(*self*, *api_params*, ***kwargs*)

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

You may also need to set different `api_root` to a specific resource. To do that you can access the `resource_name` inside `kwargs`.

```
def get_api_root(self, api_params, **kwargs):
    if kwargs.get('resource_name') == 'some_resource_name':
        return 'http://api.another.com/'
    else:
        return self.api_root
```

```
def get_api_root(self, api_params, **kwargs):
    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 aiotapioca 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 aiohttp.helpers import BasicAuth

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'] = BasicAuth(
            api_params.get('user'), api_params.get('password'))

        return params
```

process_response(*self*, *response*, **kwargs)

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*, ****kwargs**)

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

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

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 parameters 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*, ****kwargs**)

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 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* 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 refreshes 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.

retry_request(*self*, *exception*, *error_message*, *repeat_number*, ****kwargs**):

Conditions for repeating a request. If it returns *True*, the request will be repeated.

error_handling(*self*, *exception*, *error_message*, *repeat_number*, ****kwargs**):

Wrapper for throwing custom exceptions. When, for example, the server responds with 200, and errors are passed inside json.

CONTRIBUTORS

9.1 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)

RELEASE V4.2.0 (2022-10-19)

- Added linters, mypy and pre-commit hooks.
- Added CI for tests, pre-commit and release.
- Migrate to PDM.
- Optional support orjson and ujson when available.
- Optional dependencies xml and pydantic mixins.
- Replaced setup.py to pyproject.toml.

RELEASE V4.1.1 (2022-06-21)

- Added a py.typed marker for type hints.

RELEASE V4.0.2 (2022-06-17)

- Added to_dict_by_alias option in TapiocaAdapterPydanticMixin.

RELEASE V4.0.1 (2022-06-15)

- Added option to specify a session when generating a wrapper.
- Added `max_retries_requests` flag to limit the number of retries requests.
- Added ability to specify `classmethod` in parsers.
- Dump and load data in threads. Added more abstract methods to prepare parameters and data for requests or prepare native data.
- Drop support Python 3.6 and below.
- Implemented multiple use of `aiohttp.ClientSession`, outside the context manager.
- Rework of the library structure. Splitting Tapioca client structure into classes: `TapiocaClient`, `TapiocaClientResource`, `TapiocaClientExecutor` and `TapiocaClientResponse`.
- Reworked exception handling v.2.
- Minor fixes.

RELEASE V3.8.0 (2022-05-15)

- Added propagate_exception flag after retry_request call.
- Added data parsing mechanism. Parsers can be specified in resource_mapping.
- Reworked exception handling.

RELEASE V3.7.0 (2022-04-27)

- Removed debug option.
- Expanded the possibility of error handling. Added catch aiohttp exceptions.

RELEASE V3.6.0 (2022-04-22)

- Added context transfer to `get_request_kwargs` method.
- Peddling kwargs to `format_data_to_request` and `serialize_data` methods.
- Increased the debugging data output limit.
- Removed `api_params` argument from `get_request_kwargs` method.
- Removed `PydanticSerializer`.
- Added `PydanticAdapterMixin`.

RELEASE V3.5.0 (2022-04-12)

- migration to orjson

RELEASE V3.4.2 (2022-04-08)

- Fixed requirements.

RELEASE V3.4.1 (2022-04-08)

- Fixed requirements.

RELEASE V3.4.0 (2022-04-0)

- Using aiologger for debugging logs.
- Fix for recursion due to refresh_token flag.
- Added attribute semaphore to TapiocaAdapter.
- Added ability to pass Semaphore as a client or request parameter.
- Added get_resource_mapping method to TapiocaAdapter.
- Fixed an unnecessary request.
- Added serialisation from the pydantic model.
- Reworked flag debug.

RELEASE V3.3.1 (2022-03-24)

- Expanding debugging information.

RELEASE V3.3.0 (2022-03-24)

- The handling of the refresh token parameter was changed.
- Added refresh_token attribute to the TapiocaAdapter class.
- Removed refresh_token_by_default parameter in the tapioca classes.
- Parameters passing was changed in _wrap_in_tapioca and _wrap_in_tapioca_executor.
- Minor fixes.

RELEASE V3.2.4 (2022-03-23)

- Fixed “This instance has no response object” error in `_wrap_in_tapioca` and `_wrap_in_tapioca_executor` (empty response in property descendants and pages).

RELEASE V3.2.3 (2022-03-22)

- Returned pass request_method as param in get_request_kwargs.

RELEASE V3.2.2 (2022-03-22)

- Fixed fill resource template url.

RELEASE V3.2.1 (2022-03-22)

- Context transmission was extended.

RELEASE V3.2.0 (2022-03-22)

- Added `retry_request` and `error_handling` methods.
- Added context passed to different adapter methods.

RELEASE V3.1.1 (2022-03-21)

- Fixed debugging flag.

RELEASE V3.1.0 (2022-03-21)

- Added PydanticSerializer.

RELEASE V3.0.0 (2022-03-21)

- Implementing an asynchronous fork.

RELEASE V2.1.0 (2022-03-19)

- Make `TapiocaClient` and `TapiocaClientExecutor` pickle-able.

RELEASE V2.0.2 (2022-02-25)

- Updated deprecated collections import
- Adds support for python 3.10

RELEASE V2.0.1 (2020-01-25)

- Updates the list of supported versions in setup.py

RELEASE V2.0.0 (2020-01-25)

- Drops support for python 2.7 and 3.4
- Adds support for python 3.7 and 3.8

RELEASE V1.5.1 (2019-04-19)

- Adds a `resource_name` kwarg to the `get_api_root` method

RELEASE V1.5.0 (2019-04-19)

- Removes support for Python 3.3

RELEASE V1.4.3 (2017-06-15)

RELEASE V1.4.1 (2017-05-25)

RELEASE V1.4.0 (2017-03-28)

- Adds support to Session requests

RELEASE V1.3.0 (2017-01-20)

- `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`

RELEASE V1.2.3 (2016-09-28)

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

RELEASE V1.2.2 (2016-04-23)

RELEASE V1.2.1 (2016-01-02)

RELEASE V1.1.12 (2016-05-31)

RELEASE V1.1.11 (2016-05-31)

RELEASE V1.1.10 (2016-03-27)

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

RELEASE V1.1.9 (2016-03-27)

RELEASE V1.1.8 (2016-03-27)

RELEASE V1.1.7 (2016-03-27)

RELEASE V1.1.6 (2016-02-29)

RELEASE V1.1.4 (2016-02-27)

RELEASE V1.1.0 (2016-02-27)

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

RELEASE V1.0.0 (2015-11-10)

- 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

RELEASE V0.6.0 (2015-09-23)

- 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 improvements

RELEASE V0.5.3 (2015-04-10)

- Adding max_pages and max_items to pages method

RELEASE V0.5.1 (2015-08-06)

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

RELEASE V0.5.0 (2015-08-05)

- 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

RELEASE V0.4.1 (2015-08-01)

- 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

A

`api_root`, 31

C

`ClientError` (*built-in class*), 21

F

`format_data_to_request()`, 32

G

`get_api_root()`, 31

`get_iterator_list()`, 33

`get_iterator_next_request_kwargs()`, 33

`get_request_kwargs()`, 32

`get_resource_mapping()`, 31

I

`is_authentication_expired()`, 33

P

`process_response()`, 32

R

`refresh_token`, 31

`resource_mapping`, 31

`response_to_native()`, 32

S

`semaphore`, 31

`serializer_class`, 31

`ServerError` (*built-in class*), 21

`SimpleSerializer` (*built-in class*), 18

T

`TapiocaAdapter` (*built-in class*), 31

`TapiocaException` (*built-in class*), 21