chinup Documentation

Release 0.3.2

Aron Griffis

October 22, 2014

Contents

1 Installation **3**

2 Contents **5**

2.1 Quickstart 5
2.2 Advanced 7
2.3 Settings 10
2.4 API Reference 12

3 License **13**

4 Indices and tables **15**

Chinup is a high-performance Python client for interacting with the Facebook Graph API. It features automatic request batching, transparent etags support, and support for paged responses.

If you're new to chinup, take a look at the *quickstart*. The full API is detailed in the *API reference*.

CHAPTER 1

Installation

Install using `pip` from pypi. Chinup supports Python 2.7. Chinup depends on requests and URLObject which will both be installed automatically.

```
pip install chinup
```

CHAPTER 2

Contents

2.1 Quickstart

First, install chinup:

```
pip install chinup
```

Set your app token in chinup settings. Do this in your own application code, by importing the `chinup.settings` module:

```
>>> import chinup.settings
>>> chinup.settings.APP_TOKEN = 'NGAUy7KT'
```

Now make a request:

```
>>> from chinup import ChinupBar
>>> c = ChinupBar().get('facebook')
>>> c
<Chinup id=140416098416080 GET facebook data=None response=None >
```

At this point you have a request on the queue, but it hasn't actually fetched from Facebook. It will be fetched as soon as you access the data:

```
>>> c.data
{u'about': u'The Facebook Page celebrates how our friends inspire us, support us, and help us discove
 u'can_post': False,
 u'category': u'Product/service',
 u'checkins': 348,
 u'cover': {u'cover_id': u'10152883780951729',
  u'offset_x': 0,
  u'offset_y': 45,
  u'source': u'https://scontent-b.xx.fbcdn.net/hphotos-xfp1/t31.0-8/q71/s720x720/10497021_1015288378
 u'founded': u'February 4, 2004',
 u'has_added_app': False,
 u'id': u'20531316728',
 u'is_community_page': False,
 u'is_published': True,
 u'likes': 154837767,
 u'link': u'https://www.facebook.com/facebook',
 u'mission': u'Founded in 2004, Facebook\u2019s mission is to give people the power to share and make
 u'name': u'Facebook',
 u'parking': {u'lot': 0, u'street': 0, u'valet': 0},
 u'talking_about_count': 2796719,
 u'username': u'facebook',
 u'website': u'http://www.facebook.com',
 u'were_here_count': 0}
```

As a shortcut, you can use keyed access on the chinup directly, rather than through the `data` attribute:

```
>>> c['name']
u'Facebook'
```

App tokens have very limited functionality on the Graph API. Most of the time you'll need either a user token or a page token. You can pass that token to `ChinupBar`:

```
>>> ChinupBar(token='6Fq7Uy8J').get('me')['name']
u'Aron Griffis'
```

All of the examples above make a single request and immediately access the data. The full power of chinup is harnessed by instantiating a number of Chinups at once, before accessing their response data:

```
>>> chinups = [ChinupBar(token=t).get('me') for t in tokens]
>>> len(chinups)
40
>>> for c in chinups: print c['first_name']
Vincent
Suzanne
Aron
Amy
Andrew
Cristin
Abigail
Daniel
Adam
...
```

In this example, there's only a single batch request to Facebook, itself containing 40 individual requests. If `settings.DEBUG` is enabled, you can see the count like this:

```
>>> from chinup.lowlevel import batches
>>> len(batches)
1
>>> len(batches[0])
40
```

2.1.1 Django

If you're using chinup with Django, you can put your chinup settings in Django's settings.py by prefixing `CHINUP_` like this:

```
# django settings.py

CHINUP_APP_TOKEN = 'NGAUy7KT'
CHINUP_DEBUG = DEBUG
```

Additionally you can take advantage of chinup's etags caching by hooking in the Django cache:

```
CHINUP_CACHE = 'django.core.cache.cache'
```

2.1.2 django-allauth

If chinup can import django-allauth, then it adds the ability to instantiate `ChinupBar` with `user` rather than `token`, for example:

```
>>> user = User.objects.get(username='aron')
>>> ChinupBar(user=user).get('me')['name']
u'Aron Griffis'
```

You can defer the `User` fetch to chinup by passing a username or primary key:

```
>>> ChinupBar(user='aron').get('me')['name']
u'Aron Griffis'
```

2.2 Advanced

2.2.1 Paging

Chinup supports transparent or explicit paging of Facebook data. To access all the response data, paging transparently, iterate over the Chinup object:

```
friends = ChinupBar(token='6Fq7Uy8J').get('me/friends')
for friend in friends:
    print friend['name']
```

This will fetch additional pages as necessary to iterate over the entire list of friends. Listifying will also fetch all the pages:

```
friends = ChinupBar(token='6Fq7Uy8J').get('me/friends')
friends = list(friends)
```

Alternatively you can control paging explicitly by calling the `next_page` method. In that case, you should iterate on `data` to avoid automatic paging:

```
friends = ChinupBar(token='6Fq7Uy8J').get('me/friends')
while friends:
    for friend in friends.data:
        print friend['name']
    friends = friends.next_page()
```

2.2.2 ETags

The Facebook batch API supports ETags in requests and responses. See https://developers.facebook.com/docs/reference/ads-api/batch-requests/#etags

Chinup supports ETags transparently if you provide a suitable `settings.CACHE`, then chinup will automatically convert 304 responses to the previously cached 200 response.

2.2.3 Inter-request dependencies

Chinup does not yet support inter-request dependencies with JSONpath. This is on the radar though! See https://developers.facebook.com/docs/graph-api/making-multiple-requests/#operations

2.2.4 raise_exceptions=False

If chinup encounters an error retrieving a response from the Facebook Graph, the `Chinup` object will raise its own exception whenever your code attempts to access the response data. For large batch operations where you're expecting errors, you can avoid the exception by setting `raise_exceptions=False`:

```
chinups = [ChinupBar(user=u, raise_exceptions=False).get('me')
           for u in User.objects.all()]
for c in chinups:
    print "%s: %r" % (c.user, c.data or c.exception)
```

The above example uses `raise_exceptions=False` to handle users that don't have associated tokens, or maybe their tokens have expired. The `data` attribute for those chinups will be `None` and will not raise an exception when accessed.

2.2.5 Testing

Chinup provides a mixin for Python `unittest`. The mixin clears state prior to each test, and provides `assertBatches` to make sure your code changes don't adversely affect the batching of requests to Facebook. For example:

```
from django.test import TestCase
from chinup.testing import ChinupTestMixin

from app import something

class MyTestCase(ChinupTestMixin, TestCase):

    def test_something(self):
        something()

        # Calling something() should have resulted in two batches with
        # a total of 30 requests.
        self.assertBatches(2, 30)
```

2.2.6 Subclassing

While the `Chinup` and `ChinupBar` classes can be used out of the box, they're amenable to subclassing to support object and token lookup according to the specifics of your project. The most prominent example is the built-in support for django-allauth. If the allauth module can be imported, then `ChinupBar` will accept a `user` parameter rather than requiring a `token` parameter.

Here's another example, which layers on support for Facebook page tokens from a separate table. Be sure to set `ChinupBar.chinup_class` to your subclass, as shown below.

```
import chinup
from chinup.exceptions import ChinupError, MissingToken
from myapp.models import Page


class NoSuchPage(ChinupError):
    pass


class Chinup(chinup.Chinup):
```

```
def __init__(self, **kwargs):
    self.page = kwargs.pop('page', None)

    # Make sure there's only one token provider on this Chinup:
    # page or user, not both.
    assert not (self.page and kwargs.get('user'))

    super(Chinup, self).__init__(**kwargs)

@classmethod
def prepare_batch(cls, chinups):
    """
    Populates page tokens into chinups. This also immediately
    "completes" any chinups which require a token that isn't
    available, by setting chinup.exception.
    """
    cls._fetch_pages(chinups)
    cls._fetch_page_tokens(chinups)

    # Weed out any chinups that didn't pass token stage.
    chinups = [c for c in chinups if not c.completed]

    return super(Chinup, cls).prepare_batch(chinups)

@classmethod
def _fetch_pages(cls, chinups):
    """
    Replaces .page=PK with .page=OBJ for the chinups in the list.
    If the PK can't be found, then sets NoSuchPage to be raised
    when the chinup data is accessed.
    """
    chinups = [c for c in chinups if not c.completed and not c.token
               and isinstance(c.page, basestring)]
    if chinups:
        pages = Page.objects.filter(pk__in=set(c.page for c in chinups))
        pages = {page.pk: page for page in pages}

        for c in chinups:
            page = pages.get(c.page)
            if page:
                c.page = page
            else:
                c.exception = NoSuchPage("No page with pk=%r" % c.page)

@classmethod
def _fetch_page_tokens(cls, chinups):
    """
    Sets .token for the chinups in the list that have .page set.
    If a token isn't available for the given page, then sets
    MissingToken to be raised when the chinup data is accessed.
    """
    chinups = [c for c in chinups if not c.completed and not c.token
               and c.page]
    if chinups:
        page_tokens = PageToken.objects.filter(
            account__page_id__in=set(c.page.pk for c in chinups),
        )
        page_tokens = page_tokens.select_related('account')
```

```
            tokens = {pt.account.page_id: pt.token for pt in page_tokens}

            for c in chinups:
                token = tokens.get(c.page.pk)
                if token:
                    c.token = token
                else:
                    c.exception = MissingToken("No token for %r" % c.page)


class ChinupBar(chinup.ChinupBar):
    chinup_class = Chinup

    def __init__(self, **kwargs):
        self.page = kwargs.pop('page', None)

        # Make sure there's only one token provider on this ChinupBar:
        # page or user, not both.
        assert not (self.page and kwargs.get('user'))

        super(ChinupBar, self).__init__(**kwargs)

    def _get_chinup(self, **kwargs):
        return super(ChinupBar, self)._get_chinup(
            page=self.page,
            **kwargs)
```

2.3 Settings

Here's a list of the settings available in chinup and their default values. To override settings, import the module in your application and set them, for example:

```
import chinup.settings

chinup.settings.APP_TOKEN = 'NGAUy7KT'
chinup.settings.DEBUG = True
```

If you're using Django, you can set them in your Django `settings.py`:

```
CHINUP_APP_TOKEN = 'NGAUy7KT'
CHINUP_DEBUG = DEBUG  # reflect Django DEBUG setting into chinup
```

2.3.1 APP_TOKEN

Default: `None`

An app token is required to make requests with chinup. This is because chinup *always* makes batch requests, even for a single request, and Facebook's batch API requires an app token.

If you don't set `settings.APP_TOKEN` then you must pass your app token to `ChinupBar`, however this will become unwieldy quickly:

```
ChinupBar(app_token='NGAUy7KT', token='6Fq7Uy8J').get('me')
```

2.3.2 CACHE

Default: `None`

A cache is required to take advantage of etags in batch requests. This setting can either be a cache object, or a string dotted path to a module attribute. For example, using Django's default cache:

```
CHINUP_CACHE = 'django.core.cache.cache'
```

The cache object must support the two methods: `get_many` and `set_many`.

2.3.3 DEBUG

Default: `False`

Setting this to `True` causes chinup to track all the batches, similarly to Django's tracking of database queries. Then you can verify that batching is occurring as you expect:

```
>>> from chinup.lowlevel import batches
>>> len(batches)
3
>>> [len(b) for b in batches]
[5, 10, 1]
```

This shows that you've made three batch requests so far, containing five, ten, and one request respectively. You might say to yourself: "That last one looks suspicious. Can I tune my code to include that request in the prior batch for better performance, i.e. `[5, 11]`?"

2.3.4 ETAGS

Default: `True`

Chinup will cache individual responses within a batch by default, if `settings.CACHE` is also set. You can disable this by setting `ETAGS = False`.

2.3.5 DEBUG_HEADERS

Default: `False`

Chinup will omit response headers from the `repr` output for a `Chinup` by default, since they tend to be lengthy and uninteresting. To include these headers, set `DEBUG_HEADERS = True`.

2.3.6 DEBUG_REQUESTS

Default: `DEBUG`

Chinup always logs request info, the question is what logging level it will use. By default it's `logging.DEBUG` but this becomes `logging.INFO` if `DEBUG_REQUESTS` is `True`.

2.3.7 TESTING

Default: `False`

This has the same effect as setting `DEBUG`, meaning that it causes `chinup.lowlevel.batches` to be tracked. Normally you don't set this yourself, rather it's set by the provided `ChinupTestMixin`. Then you can use `assertBatches` to make sure that changes to your code don't cause an unwelcome change in the number of batches and requests.

2.4 API Reference

This is a **WORK IN PROGRESS**. We need to update all the docstrings to provide decent documentation here.

class `chinup.`**`Chinup`**(*queue*, *method*, *path*, *data*, *token=None*, *raise_exceptions=True*, *callback=None*, *prefetch_next_page=True*)

A single FB request/response. This shouldn't be instantiated directly, rather the caller should use a ChinupBar:

chinup = ChinupBar(token='XYZ').get('me')

This returns a chinup which is a lazy request. It's on the queue and will be processed when convenient. The chinup can be access as follows:

chinup.response = raw response from FB chinup.data = dict or list from FB, depending on endpoint chinup[key] = shortcut for chinup.data[key] key in chinup = shortcut for key in chinup.data

The preferred method for accessing a list response is to iterate or listify the chinup directly. This will automatically advance through paged data, whereas accessing chinup.data will not.

list(chinup) OR

for d in chinup: do something clever with d

`completed`

Returns False if this chinup remains to be synced, otherwise returns a truthy tuple of (response, exception).

`fetch_next_page`()

Prepare to load the next page by putting a chinup on the queue. This doesn't actually do anything, of course, until .data or similar is accessed.

`make_request_dict`()

Returns a dict suitable for a single request in a batch.

`next_page`()

Returns the chinup corresponding to the next page, or None if il n'y en a pas.

classmethod **`prepare_batch`**(*chinups*)

Returns a tuple of (chinups, requests) where requests is a list of dicts appropriate for a batch request.

class `chinup.`**`ChinupBar`**(*token=None*, *app_token=None*, *raise_exceptions=True*, *prefetch_next_page=True*)

`chinup_class`

alias of `Chinup`

`queue_class`

alias of `ChinupQueue`

CHAPTER 3

License

CHAPTER 4

Indices and tables

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

C

Chinup (class in chinup), 12
chinup_class (chinup.ChinupBar attribute), 12
ChinupBar (class in chinup), 12
completed (chinup.Chinup attribute), 12

F

fetch_next_page() (chinup.Chinup method), 12

M

make_request_dict() (chinup.Chinup method), 12

N

next_page() (chinup.Chinup method), 12

P

prepare_batch() (chinup.Chinup class method), 12

Q

queue_class (chinup.ChinupBar attribute), 12