Mastodon.py Documentation

Release 1.0.7

Lorenz Diener

May 19, 2017

Contents

```
from mastodon import Mastodon

# Register app - only once!
'''
Mastodon.create_app(
     'pytooterapp',
      to_file = 'pytooter_clientcred.secret'
)
'''

# Log in - either every time, or use persisted
'''
mastodon = Mastodon(client_id = 'pytooter_clientcred.secret')
mastodon.log_in(
    'my_login_email@example.com',
    'incrediblygoodpassword',
    to_file = 'pytooter_usercred.secret'
)
'''

# Create actual instance
mastodon = Mastodon(
    client_id = 'pytooter_clientcred.secret',
    access_token = 'pytooter_usercred.secret'
)
mastodon.toot('Tooting from python!')
```

Mastodon is an ostatus based twitter-like federated social network node. It has an API that allows you to interact with its every aspect. This is a simple python wrapper for that api, provided as a single python module. By default, it talks to the Mastodon flagship instance, but it can be set to talk to any node running Mastodon.

CHAPTER 1

A note about rate limits

Mastodons API rate limits per IP. By default, the limit is 150 requests per 5 minute time slot. This can differ from instance to instance and is subject to change. Mastodon.py has three modes for dealing with rate limiting that you can pass to the constructor, “throw”, “wait” and “pace”, “wait” being the default.

In “throw” mode, Mastodon.py makes no attempt to stick to rate limits. When a request hits the rate limit, it simply throws a MastodonRateLimitError. This is for applications that need to handle all rate limiting themselves (i.e. interactive apps), or applications wanting to use Mastodon.py in a multi-threaded context (“wait” and “pace” modes are not thread safe).

In “wait” mode, once a request hits the rate limit, Mastodon.py will wait until the rate limit resets and then try again, until the request succeeds or an error is encountered. This mode is for applications that would rather just not worry about rate limits much, don’t poll the api all that often, and are okay with a call sometimes just taking a while.

In “pace” mode, Mastodon.py will delay each new request after the first one such that, if requests were to continue at the same rate, only a certain fraction (set in the constructor as ratelimit_pacefactor) of the rate limit will be used up. The fraction can be (and by default, is) greater than one. If the rate limit is hit, “pace” behaves like “wait”. This mode is probably the most advanced one and allows you to just poll in a loop without ever sleeping at all yourself. It is for applications that would rather just pretend there is no such thing as a rate limit and are fine with sometimes not being very interactive.

CHAPTER 2

A note about IDs

Mastodons API uses IDs in several places: User IDs, Toot IDs, ...

While debugging, it might be tempting to copy-paste in IDs from the web interface into your code. This will not work, as the IDs on the web interface and in the URLs are not the same as the IDs used internally in the API, so don't do that.

CHAPTER 3

Return values

Unless otherwise specified, all data is returned as python dictionaries, matching the JSON format used by the API.

User dicts

```
mastodon.account(<numerical id>)
# Returns the following dictionary:
{
    'id': # Same as <numerical id>
    'username': # The username (what you @ them with)
    'acct': # The user's account name as username@domain (@domain omitted for local␣
↪users)
    'display_name': # The user's display name
    'locked': # Denotes whether the account can be followed without a follow request
    'following_count': # How many people they follow
    'followers_count': # How many followers they have
    'statuses_count': # How many statuses they have
    'note': # Their bio
    'url': # Their URL; usually 'https://mastodon.social/users/<acct>'
    'avatar': # URL for their avatar
    'header': # URL for their header image
}
```

Toot dicts

```
mastodon.toot("Hello from Python")
# Returns the following dictionary:
{
    'id': # Numerical id of this toot
    'uri': # Descriptor for the toot
        # EG 'tag:mastodon.social,2016-11-25:objectId=<id>:objectType=Status'
```

```
    'url': # URL of the toot
    'account': # Account dict for the account which posted the status
    'in_reply_to_id': # Numerical id of the toot this toot is in response to
    'in_reply_to_account_id': # Numerical id of the account this toot is in response␣
↪to
    'reblog': # Denotes whether the toot is a reblog
    'content': # Content of the toot, as HTML: '<p>Hello from Python</p>'
    'created_at': # Creation time
    'reblogs_count': # Number of reblogs
    'favourites_count': # Number of favourites
    'reblogged': # Denotes whether the logged in user has boosted this toot
    'favourited': # Denotes whether the logged in user has favourited this toot
    'sensitive': # Denotes whether media attachments to the toot are marked sensitive
    'spoiler_text': # Warning text that should be displayed before the toot content
    'visibility': # Toot visibility ('public', 'unlisted', 'private', or 'direct')
    'mentions': # A list of account dicts mentioned in the toot
    'media_attachments': # list of media dicts of attached files. Only present
                         # when there are attached files.
    'tags': # A list of hashtag dicts used in the toot
    'application': # Application dict for the client used to post the toot
}
```

Relationship dicts

```
mastodon.account_follow(<numerical id>)
# Returns the following dictionary:
{
    'id': # Numerical id (same one as <numerical id>)
    'following': # Boolean denoting whether you follow them
    'followed_by': # Boolean denoting whether they follow you back
    'blocking': # Boolean denoting whether you are blocking them
    'muting': # Boolean denoting whether you are muting them
    'requested': # Boolean denoting whether you have sent them a follow request
}
```

Notification dicts

```
mastodon.notifications()[0]
# Returns the following dictionary:
{
    'id': # id of the notification.
    'type': # "mention", "reblog", "favourite" or "follow".
    'created_at': # The time the notification was created.
    'account': # User dict of the user from whom the notification originates.
    'status': # In case of "mention", the mentioning status.
              # In case of reblog / favourite, the reblogged / favourited status.
}
```

Context dicts

```
mastodon.status_context(<numerical id>)
# Returns the following dictionary:
{
    'ancestors': # A list of toot dicts
    'descendants': # A list of toot dicts
}
```

Media dicts

```
mastodon.media_post("image.jpg", "image/jpeg")
# Returns the following dictionary:
{
    'id': # The ID of the attachment.
    'type': # Media type, EG 'image'
    'url': # The URL for the image in the local cache
    'remote_url': # The remote URL for the media (if the image is from a remote␣
↪instance)
    'preview_url': # The URL for the media preview
    'text_url': # The display text for the media (what shows up in toots)
}
```

Card dicts

```
mastodon.status_card(<numerical id>):
# Returns the folowing dictionary
{
    'url': # The URL of the card.
    'title': # The title of the card.
    'description': # The description of the card.
    'image': # (optional) The image associated with the card.
}
```

CHAPTER 4

App registration and user authentication

Before you can use the mastodon API, you have to register your application (which gets you a client key and client secret) and then log in (which gets you an access token). These functions allow you to do those things. For convenience, once you have a client id, secret and access token, you can simply pass them to the constructor of the class, too!

Note that while it is perfectly reasonable to log back in whenever your app starts, registering a new application on every startup is not, so don't do that - instead, register an application once, and then persist your client id and secret. Convenience methods for this are provided.

static `Mastodon.`**`create_app`**(*client_name, scopes=['read', 'write', 'follow'], redirect_uris=None, website=None, to_file=None, api_base_url='https://mastodon.social', request_timeout=300*)

Create a new app with given client_name and scopes (read, write, follow)

Specify redirect_uris if you want users to be redirected to a certain page after authenticating. Specify to_file to persist your apps info to a file so you can use them in the constructor. Specify api_base_url if you want to register an app on an instance different from the flagship one.

Presently, app registration is open by default, but this is not guaranteed to be the case for all future mastodon instances or even the flagship instance in the future.

Returns client_id and client_secret.

`Mastodon.`**`__init__`**(*client_id*, *client_secret=None*, *access_token=None*, *api_base_url='https://mastodon.social'*, *debug_requests=False*, *ratelimit_method='wait'*, *ratelimit_pacefactor=1.1*, *request_timeout=300*)

Create a new API wrapper instance based on the given client_secret and client_id. If you give a client_id and it is not a file, you must also give a secret.

You can also specify an access_token, directly or as a file (as written by log_in).

Mastodon.py can try to respect rate limits in several ways, controlled by ratelimit_method. "throw" makes functions throw a MastodonRatelimitError when the rate limit is hit. "wait" mode will, once the limit is hit, wait and retry the request as soon as the rate limit resets, until it succeeds. "pace" works like throw, but tries to wait in between calls so that the limit is generally not hit (How hard it tries to not hit the rate limit can be controlled by ratelimit_pacefactor). The default setting is "wait". Note that even in "wait" and "pace" mode, requests can still fail due to network or other problems! Also note that "pace" and "wait" are NOT thread safe.

Specify api_base_url if you wish to talk to an instance other than the flagship one. If a file is given as client_id, read client ID and secret from that file.

By default, a timeout of 300 seconds is used for all requests. If you wish to change this, pass the desired timeout (in seconds) as request_timeout.

`Mastodon.`**`log_in`**(*username=None, password=None, code=None, redirect_uri='urn:ietf:wg:oauth:2.0:oob', refresh_token=None, scopes=['read', 'write', 'follow'], to_file=None*)

Your username is the e-mail you use to log in into mastodon.

Can persist access token to file, to be used in the constructor.

Supports refresh_token but Mastodon.social doesn't implement it at the moment.

Handles password, authorization_code, and refresh_token authentication.

Will throw a MastodonIllegalArgumentError if username / password are wrong, scopes are not valid or granted scopes differ from requested.

For OAuth2 documentation, compare https://github.com/doorkeeper-gem/doorkeeper/wiki/Interacting-as-an-OAuth-client-with-Doorkeeper

Returns the access token.

`Mastodon.`**`auth_request_url`**(*client_id=None, redirect_uris='urn:ietf:wg:oauth:2.0:oob', scopes=['read', 'write', 'follow']*)

Returns the url that a client needs to request the grant from the server.

CHAPTER 5

Reading data: Instance

This function allows you to fetch information associated with the current instance.

`Mastodon.`**`instance`**`()`

Retrieve basic information about the instance, including the URI and administrative contact email.

Returns a dict.

CHAPTER 6

Reading data: Timelines

This function allows you to access the timelines a logged in user could see, as well as hashtag timelines and the public timeline.

Mastodon.**timeline**(*timeline='home'*, *max_id=None*, *since_id=None*, *limit=None*)
Fetch statuses, most recent ones first. Timeline can be home, local, public, or tag/hashtag. See the following functions documentation for what those do.

The default timeline is the "home" timeline.

Returns a list of toot dicts.

Mastodon.**timeline_home**(*max_id=None*, *since_id=None*, *limit=None*)
Fetch the authenticated users home timeline (i.e. followed users and self).

Returns a list of toot dicts.

Mastodon.**timeline_local**(*max_id=None*, *since_id=None*, *limit=None*)
Fetches the local / instance-wide timeline, not including replies.

Returns a list of toot dicts.

Mastodon.**timeline_public**(*max_id=None*, *since_id=None*, *limit=None*)
Fetches the public / visible-network timeline, not including replies.

Returns a list of toot dicts.

Mastodon.**timeline_hashtag**(*hashtag*, *max_id=None*, *since_id=None*, *limit=None*)
Fetch a timeline of toots with a given hashtag.

Returns a list of toot dicts.

CHAPTER 7

Reading data: Statuses

These functions allow you to get information about single statuses.

`Mastodon.`**`status`**(*id*)
Fetch information about a single toot.

Returns a toot dict.

`Mastodon.`**`status_context`**(*id*)
Fetch information about ancestors and descendants of a toot.

Returns a context dict.

`Mastodon.`**`status_reblogged_by`**(*id*)
Fetch a list of users that have reblogged a status.

Returns a list of user dicts.

`Mastodon.`**`status_favourited_by`**(*id*)
Fetch a list of users that have favourited a status.

Returns a list of user dicts.

`Mastodon.`**`status_card`**(*id*)
Fetch a card associated with a status.

Returns a card dict.

CHAPTER 8

Reading data: Notifications

This function allows you to get information about a users notifications.

Mastodon.**notifications**(*id=None*, *max_id=None*, *since_id=None*, *limit=None*)
Fetch notifications (mentions, favourites, reblogs, follows) for the authenticated user.

Can be passed an id to fetch a single notification.

Returns a list of notification dicts.

CHAPTER 9

Reading data: Accounts

These functions allow you to get information about accounts and their relationships.

Mastodon.**account**(*id*)
Fetch account information by user id.

Returns a user dict.

Mastodon.**account_verify_credentials**()
Fetch authenticated user's account information.

Returns a user dict.

Mastodon.**account_statuses**(*id*, *max_id=None*, *since_id=None*, *limit=None*)
Fetch statuses by user id. Same options as timeline are permitted.

Returns a list of toot dicts.

Mastodon.**account_following**(*id*, *max_id=None*, *since_id=None*, *limit=None*)
Fetch users the given user is following.

Returns a list of user dicts.

Mastodon.**account_followers**(*id*, *max_id=None*, *since_id=None*, *limit=None*)
Fetch users the given user is followed by.

Returns a list of user dicts.

Mastodon.**account_relationships**(*id*)
Fetch relationships (following, followed_by, blocking) of the logged in user to a given account. id can be a list.

Returns a list of relationship dicts.

Mastodon.**account_search**(*q*, *limit=None*)
Fetch matching accounts. Will lookup an account remotely if the search term is in the username@domain format and not yet in the database.

Returns a list of user dicts.

CHAPTER 10

Reading data: Follows

Mastodon.**follows**(*uri*)

Follow a remote user by uri (username@domain).

Returns a user dict.

CHAPTER 11

Reading data: Searching

This function allows you to search for content.

`Mastodon.`**`search`**(*q*, *resolve=False*)

Fetch matching hashtags, accounts and statuses. Will search federated instances if resolve is True.

Returns a dict of lists.

CHAPTER 12

Reading data: Mutes and blocks

These functions allow you to get information about accounts that are muted or blocked by the logged in user.

`Mastodon.`**`mutes`**(*max_id=None*, *since_id=None*, *limit=None*)
Fetch a list of users muted by the authenticated user.

Returns a list of user dicts.

`Mastodon.`**`blocks`**(*max_id=None*, *since_id=None*, *limit=None*)
Fetch a list of users blocked by the authenticated user.

Returns a list of user dicts.

CHAPTER 13

Reading data: Reports

These functions allow you to retrieve information about reports filed by the authenticated user, and file a report against a user.

`Mastodon.`**`reports`**`()`
Fetch a list of reports made by the authenticated user.

Returns a list of report dicts.

`Mastodon.`**`report`**`(`*account_id*, *status_ids*, *comment*`)`
Report a user to the admin.

Accepts a list of toot IDs associated with the report, and a comment.

Returns a report dict.

CHAPTER 14

Reading data: Favourites

This function allows you to get information about statuses favourited by the authenticated user.

`Mastodon.`**`favourites`**(*max_id=None*, *since_id=None*, *limit=None*)

Fetch the authenticated user's favourited statuses.

Returns a list of toot dicts.

CHAPTER 15

Reading data: Follow requests

This function allows you to get a list of pending incoming follow requests for the authenticated user.

`Mastodon.`**`follow_requests`**(*max_id=None*, *since_id=None*, *limit=None*)

Fetch the authenticated user's incoming follow requests.

Returns a list of user dicts.

CHAPTER 16

Writing data: Statuses

These functions allow you to post statuses to Mastodon and to interact with already posted statuses.

Mastodon.**status_post**(*status*, *in_reply_to_id=None*, *media_ids=None*, *sensitive=False*, *visibility=''*, *spoiler_text=None*)

Post a status. Can optionally be in reply to another status and contain up to four pieces of media (Uploaded via media_post()). media_ids can also be the media dicts returned by media_post - they are unpacked automatically.

The 'sensitive' boolean decides whether or not media attached to the post should be marked as sensitive, which hides it by default on the Mastodon web front-end.

The visibility parameter is a string value and matches the visibility option on the /api/v1/status POST API endpoint. It accepts any of: 'private' - post will be visible only to followers 'unlisted' - post will be public but not appear on the public timeline 'public' - post will be public

If not passed in, visibility defaults to match the current account's privacy setting (private if the account is locked, public otherwise).

The spoiler_text parameter is a string to be shown as a warning before the text of the status. If no text is passed in, no warning will be displayed.

Returns a toot dict with the new status.

Mastodon.**toot**(*status*)

Synonym for status_post that only takes the status text as input.

Returns a toot dict with the new status.

Mastodon.**status_reblog**(*id*)

Reblog a status.

Returns a toot with with a new status that wraps around the reblogged one.

Mastodon.**status_unreblog**(*id*)

Un-reblog a status.

Returns a toot dict with the status that used to be reblogged.

Mastodon.**status_favourite**(*id*)

Favourite a status.

Returns a toot dict with the favourited status.

Mastodon.**status_unfavourite**(*id*)
Un-favourite a status.

Returns a toot dict with the un-favourited status.

Mastodon.**status_delete**(*id*)
Delete a status

Returns an empty dict for good measure.

CHAPTER 17

Writing data: Accounts

These functions allow you to interact with other accounts: To (un)follow and (un)block.

Mastodon.**account_follow**(*id*)
Follow a user.

Returns a relationship dict containing the updated relationship to the user.

Mastodon.**follows**(*uri*)
Follow a remote user by uri (username@domain).

Returns a user dict.

Mastodon.**account_unfollow**(*id*)
Unfollow a user.

Returns a relationship dict containing the updated relationship to the user.

Mastodon.**account_block**(*id*)
Block a user.

Returns a relationship dict containing the updated relationship to the user.

Mastodon.**account_unblock**(*id*)
Unblock a user.

Returns a relationship dict containing the updated relationship to the user.

Mastodon.**account_mute**(*id*)
Mute a user.

Returns a relationship dict containing the updated relationship to the user.

Mastodon.**account_unmute**(*id*)
Unmute a user.

Returns a relationship dict containing the updated relationship to the user.

Mastodon.**account_update_credentials**(*display_name=None*, *note=None*, *avatar=None*, *header=None*)
Update the profile for the currently authenticated user.

'note' is the user's bio.

'avatar' and 'header' are images encoded in base64, prepended by a content-type (for example: '[...]')

CHAPTER 18

Writing data: Follow requests

These functions allow you to accept or reject incoming follow requests.

`Mastodon.`**`follow_request_authorize`**`(`*id*`)`
: Accept an incoming follow request.

  Returns an empty dict.

`Mastodon.`**`follow_request_reject`**`(`*id*`)`
: Reject an incoming follow request.

  Returns an empty dict.

CHAPTER 19

Writing data: Media

This function allows you to upload media to Mastodon. The returned media IDs (Up to 4 at the same time) can then be used with post_status to attach media to statuses.

Mastodon.**media_post**(*media_file*, *mime_type=None*)

Post an image. media_file can either be image data or a file name. If image data is passed directly, the mime type has to be specified manually, otherwise, it is determined from the file name.

Throws a MastodonIllegalArgumentError if the mime type of the passed data or file can not be determined properly.

Returns a media dict. This contains the id that can be used in status_post to attach the media file to a toot.

CHAPTER 20

Streaming

These functions allow access to the streaming API.

`Mastodon.`**`user_stream`**(*listener*)
: Streams events that are relevant to the authorized user, i.e. home timeline and notifications. 'listener' should be a subclass of StreamListener.

  This method blocks forever, calling callbacks on 'listener' for incoming events.

`Mastodon.`**`public_stream`**(*listener*)
: Streams public events. 'listener' should be a subclass of StreamListener.

  This method blocks forever, calling callbacks on 'listener' for incoming events.

`Mastodon.`**`hashtag_stream`**(*tag*, *listener*)
: Returns all public statuses for the hashtag 'tag'. 'listener' should be a subclass of StreamListener.

  This method blocks forever, calling callbacks on 'listener' for incoming events.

Python Module Index

m

Index

Symbols

A

B

C

F

H

I

L

M

N

P

R

S

T

U