fbchat

Release 2.0.0a4

Taehoon Kim; Moreels Pieter-Jan; Mads Marquart

Jun 07, 2020

CONTENTS

A powerful and efficient library to interact with Facebook's Messenger, using just your email and password.

This is *not* an official API, Facebook has that over here for chat bots. This library differs by using a normal Facebook account instead.

`fbchat` currently support:

- Sending many types of messages, with files, stickers, mentions, etc.
- Fetching all messages, threads and images in threads.
- Searching for messages and threads.
- Creating groups, setting the group emoji, changing nicknames, creating polls, etc.
- Listening for, an reacting to messages and other events in real-time.
- Type hints, and it has a modern codebase (e.g. only Python 3.5 and upwards).
- `async`/`await` (COMING).

Essentially, everything you need to make an amazing Facebook bot!

CHAPTER ONE

VERSION WARNING

`v2` is currently being developed at the `master` branch and it's highly unstable. If you want to view the old `v1`, go here.

Additionally, you can view the project's progress here.

CHAPTER TWO

CAVEATS

`fbchat` works by imitating what the browser does, and thereby tricking Facebook into thinking it's accessing the website normally.

However, there's a catch! **Using this library may not comply with Facebook's Terms Of Service!**, so be responsible Facebook citizens! We are not responsible if your account gets banned!

Additionally, **the APIs the library is calling is undocumented!** In theory, this means that your code could break tomorrow, without the slightest warning! If this happens to you, please report it, so that we can fix it as soon as possible!

With that said, let's get started!

CHAPTER THREE

INSTALLATION

```
$ pip install fbchat
```

If you don't have pip, this guide can guide you through the process.

You can also install directly from source, provided you have `pip>=19.0`:

```
$ pip install git+https://github.com/carpedm20/fbchat.git
```

CHAPTER FOUR

DOCUMENTATION OVERVIEW

4.1 Introduction

Welcome, this page will guide you through the basic concepts of using `fbchat`.

The hardest, and most error prone part is logging in, and managing your login session, so that is what we will look at first.

4.1.1 Logging In

Everything in `fbchat` starts with getting an instance of *`Session`*. Currently there are two ways of doing that, *`Session.login`* and *`Session.from_cookies`*.

The follow example will prompt you for you password, and use it to login:

```
import getpass
import fbchat
session = fbchat.Session.login("<email/phone number>", getpass.getpass())
# If your account requires a two factor authentication code:
session = fbchat.Session.login(
    "<your email/phone number>",
    getpass.getpass(),
    lambda: getpass.getpass("2FA code"),
)
```

However, **this is not something you should do often!** Logging in/out all the time *will* get your Facebook account locked!

Instead, you should start by using *`Session.login`*, and then store the cookies with *`Session.get_cookies`*, so that they can be used instead the next time your application starts.

Usability-wise, this is also better, since you won't have to re-type your password every time you want to login.

The following, quite lengthy, yet very import example, illustrates a way to do this:

```
# TODO: Consider adding Session.from_file and Session.to_file,
# which would make this example a lot easier!

import atexit
import json
import getpass
import fbchat
```

(continues on next page)

(continued from previous page)

```python
def load_cookies(filename):
    try:
        # Load cookies from file
        with open(filename) as f:
            return json.load(f)
    except FileNotFoundError:
        return  # No cookies yet


def save_cookies(filename, cookies):
    with open(filename, "w") as f:
        json.dump(cookies, f)


def load_session(cookies):
    if not cookies:
        return
    try:
        return fbchat.Session.from_cookies(cookies)
    except fbchat.FacebookError:
        return  # Failed loading from cookies


cookies = load_cookies("session.json")
session = load_session(cookies)
if not session:
    # Session could not be loaded, login instead!
    session = fbchat.Session.login("<email>", getpass.getpass())

# Save session cookies to file when the program exits
atexit.register(lambda: save_cookies("session.json", session.get_cookies()))

# Do stuff with session here
```

Assuming you have successfully completed the above, congratulations! Using `fbchat` should be mostly trouble free from now on!

4.1.2 Understanding Thread Ids

At the core of any thread is its unique identifier, its ID.

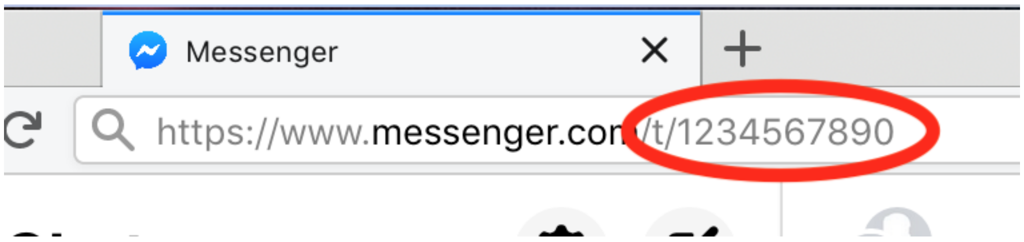
A thread basically just means "something I can chat with", but more precisely, it can refer to a few things: - A Messenger group thread (*Group*) - The conversation between you and a single Facebook user (*User*) - The conversation between you and a Facebook Page (*Page*)

You can get your own user ID from *Session.user* with `session.user.id`.

Getting the ID of a specific group thread is fairly trivial, you only need to login to https://www.messenger.com/, click on the group you want to find the ID of, and then read the id from the address bar. The URL will look something like this: `https://www.messenger.com/t/1234567890`, where `1234567890` would be the ID of the group.

The same method can be applied to some user accounts, though if they have set a custom URL, then you will have to use a different method.

An image to illustrate the process is shown below:



Once you have an ID, you can use it to create a *Group* or a *User* instance, which will allow you to do all sorts of things. To do this, you need a valid, logged in session:

```
group = fbchat.Group(session=session, id="<The id you found>")
# Or for user threads
user = fbchat.User(session=session, id="<The id you found>")
```

Just like threads, every message, poll, plan, attachment, action etc. you send or do on Facebook has a unique ID.

Below is an example of using such a message ID to get a *Message* instance:

```
# Provide the thread the message was created in, and it's ID
message = fbchat.Message(thread=user, id="<The message id>")
```

4.1.3 Fetching Information

Managing these ids yourself quickly becomes very cumbersome! Luckily, there are other, easier ways of getting *Group*/*User* instances.

You would start by creating a *Client* instance, which is basically just a helper on top of *Session*, that will allow you to do various things:

```
client = fbchat.Client(session=session)
```

Now, you could search for threads using *Client.search_for_threads*, or fetch a list of them using *Client.fetch_threads*:

```
# Fetch the 5 most likely search results
# Uses Facebook's search functions, you don't have to specify the whole name, first␣
↪names will usually be enough
threads = list(client.search_for_threads("<name of the thread to search for>",␣
↪limit=5))
# Fetch the 5 most recent threads in your account
threads = list(client.fetch_threads(limit=5))
```

Note the `list` statements; this is because the methods actually return generators. If you don't know what that means, don't worry, it is just something you can use to make your code faster later.

The examples above will actually fetch *UserData*/*GroupData*, which are subclasses of *User*/*Group*. These model have extra properties, so you could for example print the names and ids of the fetched threads like this:

```
for thread in threads:
    print(f"{thread.id}: {thread.name}")
```

Once you have a thread, you can use that to fetch the messages therein:

```
for message in thread.fetch_messages(limit=20):
    print(message.text)
```

4.1.4 Interacting with Threads

Once you have a *User*/*Group* instance, you can do things on them as described in *ThreadABC*, since they are subclasses of that.

Some functionality, like adding users to a *Group*, or blocking a *User*, logically only works the relevant threads, so see the full API documentation for that.

With that out of the way, let's see some examples!

The simplest way of interacting with a thread is by sending a message:

```
# Send a message to the user
message = user.send_text("test message")
```

There are many types of messages you can send, see the full API documentation for more.

Notice how we held on to the sent message? The return type i a *Message* instance, so you can interact with it afterwards:

```
# React to the message with the  emoji
message.react("")
```

Besides sending messages, you can also interact with threads in other ways. An example is to change the thread color:

```
# Will change the thread color to the default blue
thread.set_color("#0084ff")
```

4.1.5 Listening & Events

Now, we are finally at the point we have all been waiting for: Creating an automatic Facebook bot!

To get started, you create the functions you want to call on certain events:

```
def my_function(event: fbchat.MessageEvent):
    print(f"Message from {event.author.id}: {event.message.text}")
```

Then you create a *fbchat.Listener* object:

```
listener = fbchat.Listener(session=session, chat_on=False, foreground=False)
```

Which you can then use to receive events, and send them to your functions:

```
for event in listener.listen():
    if isinstance(event, fbchat.MessageEvent):
        my_function(event)
```

View the *Examples* to see some more examples illustrating the event system.

4.2 Examples

These are a few examples on how to use `fbchat`. Remember to swap out `<email>` and `<password>` for your email and password

4.2.1 Basic example

This will show basic usage of `fbchat`

```
import fbchat

# Log the user in
session = fbchat.Session.login("<email>", "<password>")

print("Own id: {}".format(session.user.id))

# Send a message to yourself
session.user.send_text("Hi me!")

# Log the user out
session.logout()
```

4.2.2 Interacting with Threads

This will interact with the thread in every way `fbchat` supports

```
import fbchat
import requests

session = fbchat.Session.login("<email>", "<password>")

client = fbchat.Client(session)

thread = session.user
# thread = fbchat.User(session=session, id="0987654321")
# thread = fbchat.Group(session=session, id="1234567890")

# Will send a message to the thread
thread.send_text("<message>")

# Will send the default `like` emoji
thread.send_sticker(fbchat.EmojiSize.LARGE.value)

# Will send the emoji ``
thread.send_emoji("", size=fbchat.EmojiSize.LARGE)

# Will send the sticker with ID `767334476626295`
thread.send_sticker("767334476626295")

# Will send a message with a mention
thread.send_text(
    text="This is a @mention",
    mentions=[fbchat.Mention(thread.id, offset=10, length=8)],
)
```

(continues on next page)

(continued from previous page)

```
# Will send the image located at `<image path>`
with open("<image path>", "rb") as f:
    files = client.upload([("image_name.png", f, "image/png")])
thread.send_text(text="This is a local image", files=files)

# Will download the image at the URL `<image url>`, and then send it
r = requests.get("<image url>")
files = client.upload([("image_name.png", r.content, "image/png")])
thread.send_files(files)  # Alternative to .send_text


# Only do these actions if the thread is a group
if isinstance(thread, fbchat.Group):
    # Will remove the user with ID `<user id>` from the group
    thread.remove_participant("<user id>")
    # Will add the users with IDs `<1st user id>`, `<2nd user id>` and `<3th user id>
↪` to the group
    thread.add_participants(["<1st user id>", "<2nd user id>", "<3rd user id>"])
    # Will change the title of the group to `<title>`
    thread.set_title("<title>")


# Will change the nickname of the user `<user id>` to `<new nickname>`
thread.set_nickname(fbchat.User(session=session, id="<user id>"), "<new nickname>")

# Will set the typing status of the thread
thread.start_typing()

# Will change the thread color to #0084ff
thread.set_color("#0084ff")

# Will change the thread emoji to ``
thread.set_emoji("")

message = fbchat.Message(thread=thread, id="<message id>")

# Will react to a message with a  emoji
message.react("")
```

4.2.3 Fetching Information

This will show the different ways of fetching information about users and threads

```
import fbchat

session = fbchat.Session.login("<email>", "<password>")

client = fbchat.Client(session=session)

# Fetches a list of all users you're currently chatting with, as `User` objects
users = client.fetch_all_users()

print("users' IDs: {}".format([user.id for user in users]))
print("users' names: {}".format([user.name for user in users]))
```

(continues on next page)

(continued from previous page)

```
# If we have a user id, we can use `fetch_user_info` to fetch a `User` object
user = client.fetch_user_info("<user id>")["<user id>"]
# We can also query both mutiple users together, which returns list of `User` objects
users = client.fetch_user_info("<1st user id>", "<2nd user id>", "<3rd user id>")

print("user's name: {}".format(user.name))
print("users' names: {}".format([users[k].name for k in users]))


# `search_for_users` searches for the user and gives us a list of the results,
# and then we just take the first one, aka. the most likely one:
user = client.search_for_users("<name of user>")[0]

print("user ID: {}".format(user.id))
print("user's name: {}".format(user.name))
print("user's photo: {}".format(user.photo))
print("Is user client's friend: {}".format(user.is_friend))


# Fetches a list of the 20 top threads you're currently chatting with
threads = client.fetch_thread_list()
# Fetches the next 10 threads
threads += client.fetch_thread_list(offset=20, limit=10)

print("Threads: {}".format(threads))


# If we have a thread id, we can use `fetch_thread_info` to fetch a `Thread` object
thread = client.fetch_thread_info("<thread id>")["<thread id>"]
print("thread's name: {}".format(thread.name))


# Gets the last 10 messages sent to the thread
messages = thread.fetch_messages(limit=10)
# Since the message come in reversed order, reverse them
messages.reverse()

# Prints the content of all the messages
for message in messages:
    print(message.text)


# `search_for_threads` searches works like `search_for_users`, but gives us a list of␣
↪threads instead
thread = client.search_for_threads("<name of thread>")[0]
print("thread's name: {}".format(thread.name))


# Here should be an example of `getUnread`


# Print image url for up to 20 last images from thread.
images = list(thread.fetch_images(limit=20))
for image in images:
    if isinstance(image, fbchat.ImageAttachment):
```

(continues on next page)

(continued from previous page)

```
        url = client.fetch_image_url(image.id)
        print(url)
```

4.2.4 Echobot

This will reply to any message with the same message

```
import fbchat

session = fbchat.Session.login("<email>", "<password>")
listener = fbchat.Listener(session=session, chat_on=False, foreground=False)

for event in listener.listen():
    if isinstance(event, fbchat.MessageEvent):
        print(f"{event.message.text} from {event.author.id} in {event.thread.id}")
        # If you're not the author, echo
        if event.author.id != session.user.id:
            event.thread.send_text(event.message.text)
```

4.2.5 Remove Bot

This will remove a user from a group if they write the message `Remove me!`

```
import fbchat


def on_message(event):
    # We can only kick people from group chats, so no need to try if it's a user chat
    if not isinstance(event.thread, fbchat.Group):
        return
    if event.message.text == "Remove me!":
        print(f"{event.author.id} will be removed from {event.thread.id}")
        event.thread.remove_participant(event.author.id)


session = fbchat.Session.login("<email>", "<password>")
listener = fbchat.Listener(session=session, chat_on=False, foreground=False)
for event in listener.listen():
    if isinstance(event, fbchat.MessageEvent):
        on_message(event)
```

4.2.6 “Prevent changes”-Bot

This will prevent chat color, emoji, nicknames and chat name from being changed. It will also prevent people from being added and removed

```
# This example uses the `blinker` library to dispatch events. See echobot.py for how
# this could be done differenly. The decision is entirely up to you!
import fbchat
import blinker

```

(continues on next page)

(continued from previous page)

```
# Change this to your group id
old_thread_id = "1234567890"

# Change these to match your liking
old_color = "#0084ff"
old_emoji = ""
old_title = "Old group chat name"
old_nicknames = {
    "12345678901": "User nr. 1's nickname",
    "12345678902": "User nr. 2's nickname",
    "12345678903": "User nr. 3's nickname",
    "12345678904": "User nr. 4's nickname",
}

# Create a blinker signal
events = blinker.Signal()

# Register various event handlers on the signal
@events.connect_via(fbchat.ColorSet)
def on_color_set(sender, event: fbchat.ColorSet):
    if old_thread_id != event.thread.id:
        return
    if old_color != event.color:
        print(f"{event.author.id} changed the thread color. It will be changed back")
        event.thread.set_color(old_color)


@events.connect_via(fbchat.EmojiSet)
def on_emoji_set(sender, event: fbchat.EmojiSet):
    if old_thread_id != event.thread.id:
        return
    if old_emoji != event.emoji:
        print(f"{event.author.id} changed the thread emoji. It will be changed back")
        event.thread.set_emoji(old_emoji)


@events.connect_via(fbchat.TitleSet)
def on_title_set(sender, event: fbchat.TitleSet):
    if old_thread_id != event.thread.id:
        return
    if old_title != event.title:
        print(f"{event.author.id} changed the thread title. It will be changed back")
        event.thread.set_title(old_title)


@events.connect_via(fbchat.NicknameSet)
def on_nickname_set(sender, event: fbchat.NicknameSet):
    if old_thread_id != event.thread.id:
        return
    old_nickname = old_nicknames.get(event.subject.id)
    if old_nickname != event.nickname:
        print(
            f"{event.author.id} changed {event.subject.id}'s' nickname."
            " It will be changed back"
        )
        event.thread.set_nickname(event.subject.id, old_nickname)
```

(continues on next page)

(continued from previous page)

```python
@events.connect_via(fbchat.PeopleAdded)
def on_people_added(sender, event: fbchat.PeopleAdded):
    if old_thread_id != event.thread.id:
        return
    if event.author.id != session.user.id:
        print(f"{', '.join(x.id for x in event.added)} got added. They will be removed
↪")
        for added in event.added:
            event.thread.remove_participant(added.id)


@events.connect_via(fbchat.PersonRemoved)
def on_person_removed(sender, event: fbchat.PersonRemoved):
    if old_thread_id != event.thread.id:
        return
    # No point in trying to add ourself
    if event.removed.id == session.user.id:
        return
    if event.author.id != session.user.id:
        print(f"{event.removed.id} got removed. They will be re-added")
        event.thread.add_participants([event.removed.id])


# Login, and start listening for events
session = fbchat.Session.login("<email>", "<password>")
listener = fbchat.Listener(session=session, chat_on=False, foreground=False)

for event in listener.listen():
    # Dispatch the event to the subscribed handlers
    events.send(type(event), event=event)
```

4.3 Frequently Asked Questions

4.3.1 The new version broke my application

`fbchat` follows Scemantic Versioning quite rigorously!

That means that breaking changes can *only* occur in major versions (e.g. `v1.9.6` -> `v2.0.0`).

If you find that something breaks, and you didn't update to a new major version, then it is a bug, and we would be grateful if you reported it!

In case you're stuck with an old codebase, you can downgrade to a previous version of `fbchat`, e.g. version `1.9.6`:

```
$ pip install fbchat==1.9.6
```

4.3.2 Will you be supporting creating posts/events/pages and so on?

We won't be focusing on anything else than chat-related things. This library is called `fbCHAT`, after all!

4.4 Full API

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

4.4.1 Session

class fbchat.**Session**(*, *user_id*, *fb_dtsg*, *revision*, *session=NOTHING*, *counter=0*, *client_id=NOTHING*)

Stores and manages state required for most Facebook requests.

This is the main class, which is used to login to Facebook.

property user

The logged in user.

classmethod login(*email*, *password*, *on_2fa_callback=None*)

Login the user, using `email` and `password`.

Parameters

- **email** (`str`) – Facebook `email`, `id` or `phone number`
- **password** (`str`) – Facebook account password
- **on_2fa_callback** (`Optional`[`Callable`[[], `int`]]) – Function that will be called, in case a two factor authentication code is needed. This should return the requested code.

  Tested using SMS and authentication applications. If you have both enabled, you might not receive an SMS code, and you'll have to use the authentication application.

  Note: Facebook limits the amount of codes they will give you, so if you don't receive a code, be patient, and try again later!

Example

```
>>> import fbchat
>>> import getpass
>>> session = fbchat.Session.login(
...     input("Email: "),
...     getpass.getpass(),
...     on_2fa_callback=lambda: input("2FA Code: ")
... )
Email: abc@gmail.com
Password: ****
2FA Code: 123456
>>> session.user.id
"1234"
```

is_logged_in()

Send a request to Facebook to check the login status.

Return type `bool`

Returns Whether the user is still logged in

Example

```
>>> assert session.is_logged_in()
```

logout()

Safely log out the user.

The session object must not be used after this action has been performed!

Example

```
>>> session.logout()
```

Return type None

get_cookies()

Retrieve session cookies, that can later be used in *from_cookies*.

Return type Mapping[str, str]

Returns A dictionary containing session cookies

Example

```
>>> cookies = session.get_cookies()
```

classmethod from_cookies(*cookies*)

Load a session from session cookies.

Parameters **cookies** (Mapping[str, str]) – A dictionary containing session cookies

Example

```
>>> cookies = session.get_cookies()
>>> # Store cookies somewhere, and then subsequently
>>> session = fbchat.Session.from_cookies(cookies)
```

4.4.2 Client

class fbchat.**Client**(**, session*)

A client for Facebook Messenger.

This contains methods that are generally needed to interact with Facebook.

Example

Create a new client instance.

```
>>> client = fbchat.Client(session=session)
```

session
The session to use when making requests.

fetch_users()
Fetch users the client is currently chatting with.

This is very close to your friend list, with the follow differences:

It differs by including users that you're not friends with, but have chatted with before, and by including accounts that are "Messenger Only".

But does not include deactivated, deleted or memorialized users (logically, since you can't chat with those).

The order these are returned is arbitrary.

Example

Get the name of an arbitrary user that you're currently chatting with.

```
>>> users = client.fetch_users()
>>> users[0].name
"A user"
```

Return type Sequence[UserData]

search_for_users(*name*, *limit*)
Find and get users by their name.

The returned users are ordered by relevance.

Parameters

- **name** (str) – Name of the user
- **limit** (int) – The max. amount of users to fetch

Example

Get the full name of the first found user.

```
>>> (user,) = client.search_for_users("user", limit=1)
>>> user.name
"A user"
```

Return type Iterable[UserData]

search_for_pages(*name*, *limit*)
Find and get pages by their name.

The returned pages are ordered by relevance.

Parameters

- **name** (str) – Name of the page
- **limit** (int) – The max. amount of pages to fetch

Example

Get the full name of the first found page.

```
>>> (page,) = client.search_for_pages("page", limit=1)
>>> page.name
"A page"
```

Return type Iterable[PageData]

search_for_groups (*name*, *limit*)
Find and get group threads by their name.

The returned groups are ordered by relevance.

Parameters

- **name** (str) – Name of the group thread
- **limit** (int) – The max. amount of groups to fetch

Example

Get the full name of the first found group.

```
>>> (group,) = client.search_for_groups("group", limit=1)
>>> group.name
"A group"
```

Return type Iterable[GroupData]

search_for_threads (*name*, *limit*)
Find and get threads by their name.

The returned threads are ordered by relevance.

Parameters

- **name** (str) – Name of the thread
- **limit** (int) – The max. amount of threads to fetch

Example

Search for a user, and get the full name of the first found result.

```
>>> (user,) = client.search_for_threads("user", limit=1)
>>> assert isinstance(user, fbchat.User)
>>> user.name
"A user"
```

Return type Iterable[ThreadABC]

search_messages(*query*, *limit*)

Search for messages in all threads.

Intended to be used alongside *ThreadABC.search_messages*.

Warning! If someone send a message to a thread that matches the query, while we're searching, some snippets will get returned twice, and some will be lost.

This is fundamentally not fixable, it's just how the endpoint is implemented.

Parameters

- **query** (str) – Text to search for
- **limit** (Optional[int]) – Max. number of items to retrieve. If None, all will be retrieved

Example

Search for messages, and print the amount of snippets in each thread.

```
>>> for thread, count in client.search_messages("abc", limit=3):
...     print(f"{thread.id} matched the search {count} time(s)")
...
1234 matched the search 2 time(s)
2345 matched the search 1 time(s)
3456 matched the search 100 time(s)
```

Return type Iterable[Tuple[ThreadABC, int]]

Returns Iterable with tuples of threads, and the total amount of matches.

fetch_thread_info(*ids*)

Fetch threads' info from IDs, unordered.

Warning: Sends two requests if users or pages are present, to fetch all available info!

Parameters **ids** (Iterable[str]) – Thread ids to query

Example

Get data about the user with id "4".

```
>>> (user,) = client.fetch_thread_info(["4"])
>>> user.name
"Mark Zuckerberg"
```

Return type Iterable[ThreadABC]

fetch_threads(*limit*, *location=<ThreadLocation.INBOX: 'INBOX'>*)

Fetch the client's thread list.

The returned threads are ordered by last active first.

Parameters

- **limit** (Optional[int]) – Max. number of threads to retrieve. If None, all threads will be retrieved.
- **location** (ThreadLocation) – INBOX, PENDING, ARCHIVED or OTHER

Example

Fetch the last three threads that the user chatted with.

```
>>> for thread in client.fetch_threads(limit=3):
...     print(f"{thread.id}: {thread.name}")
...
1234: A user
2345: A group
3456: A page
```

Return type Iterable[ThreadABC]

fetch_unread()
Fetch unread threads.

Warning: This is not finished, and the API may change at any point!

Return type Sequence[ThreadABC]

fetch_unseen()
Fetch unseen / new threads.

Warning: This is not finished, and the API may change at any point!

Return type Sequence[ThreadABC]

fetch_image_url(*image_id*)
Fetch URL to download the original image from an image attachment ID.

Parameters **image_id** (str) – The image you want to fetch

Example

```
>>> client.fetch_image_url("1234")
"https://scontent-arn1-1.xx.fbcdn.net/v/t1.123-4/1_23_45_n.png?..."
```

Return type str

Returns An URL where you can download the original image

get_phone_numbers()
Fetch the user's phone numbers.

Return type Sequence[str]

get_emails()

Fetch the user's emails.

Return type Sequence[str]

upload(*files*, *voice_clip=False*)

Upload files to Facebook.

`Distribution files` should be a list of files that requests can upload, see requests.request.

Example

```
>>> with open("file.txt", "rb") as f:
...     (file,) = client.upload([("file.txt", f, "text/plain")])
...
>>> file
("1234", "text/plain")
```

Return type Sequence[Tuple[str, str]]

Returns Tuples with a file's ID and mimetype. This result can be passed straight on to *ThreadABC.send_files*, or used in *Group.set_image*.

mark_as_delivered(*message*)

Mark a message as delivered.

Warning: This is not finished, and the API may change at any point!

Parameters **message** (Message) – The message to set as delivered

mark_as_read(*threads*, *at*)

Mark threads as read.

All messages inside the specified threads will be marked as read.

Parameters

- **threads** (Iterable[ThreadABC]) – Threads to set as read
- **at** (datetime) – Timestamp to signal the read cursor at

mark_as_unread(*threads*, *at*)

Mark threads as unread.

All messages inside the specified threads will be marked as unread.

Parameters

- **threads** (Iterable[ThreadABC]) – Threads to set as unread
- **at** (datetime) – Timestamp to signal the read cursor at

move_threads(*location*, *threads*)

Move threads to specified location.

Parameters

- **location** (ThreadLocation) – INBOX, PENDING, ARCHIVED or OTHER
- **threads** (Iterable[ThreadABC]) – Threads to move

delete_threads (*threads*)
Bulk delete threads.

Parameters **threads** (Iterable[ThreadABC]) – Threads to delete

Example

```
>>> group = fbchat.Group(session=session, id="1234")
>>> client.delete_threads([group])
```

delete_messages (*messages*)
Bulk delete specified messages.

Parameters **messages** (Iterable[Message]) – Messages to delete

Example

```
>>> message1 = fbchat.Message(thread=thread, id="1234")
>>> message2 = fbchat.Message(thread=thread, id="2345")
>>> client.delete_threads([message1, message2])
```

4.4.3 Threads

class fbchat.**ThreadABC**
Implemented by thread-like classes.

This is private to implement.

abstract property session
The session to use when making requests.

Return type Session

abstract property id
The unique identifier of the thread.

Return type str

wave (*first=True*)
Wave hello to the thread.

Parameters **first** (bool) – Whether to wave first or wave back

Example

Wave back to the thread.

```
>>> thread.wave(False)
```

Return type str

send_text (*text*, *mentions=None*, *files=None*, *reply_to_id=None*)
Send a message to the thread.

Parameters

- **text** (str) – Text to send
- **mentions** (Optional[Iterable[Mention]]) – Optional mentions
- **files** (Optional[Iterable[Tuple[str, str]]]) – Optional tuples, each containing an uploaded file's ID and mimetype. See *ThreadABC.send_files* for an example.
- **reply_to_id** (Optional[str]) – Optional message to reply to

Example

```
>>> mention = fbchat.Mention(thread_id="1234", offset=5, length=2)
>>> thread.send_text("A message", mentions=[mention])
```

Return type str

Returns The sent message

send_emoji (*emoji*, *size*)
Send an emoji to the thread.

Parameters

- **emoji** (str) – The emoji to send
- **size** (EmojiSize) – The size of the emoji

Example

```
>>> thread.send_emoji("", size=fbchat.EmojiSize.LARGE)
```

Return type str

Returns The sent message

send_sticker (*sticker_id*)
Send a sticker to the thread.

Parameters **sticker_id** (str) – ID of the sticker to send

Example

Send a sticker with the id "1889713947839631"

```
>>> thread.send_sticker("1889713947839631")
```

Return type str

Returns The sent message

send_location (*latitude*, *longitude*)
Send a given location to a thread as the user's current location.

Parameters

- **latitude** (float) – The location latitude
- **longitude** (float) – The location longitude

Example

Send a location in London, United Kingdom.

```
>>> thread.send_location(51.5287718, -0.2416815)
```

send_pinned_location (*latitude*, *longitude*)
Send a given location to a thread as a pinned location.

Parameters

- **latitude** (float) – The location latitude
- **longitude** (float) – The location longitude

Example

Send a pinned location in Beijing, China.

```
>>> thread.send_pinned_location(39.9390731, 116.117273)
```

send_files (*files*)
Send files from file IDs to a thread.

`Distribution files` should be a list of tuples, with a file's ID and mimetype.

Example

Upload and send a video to a thread.

```
>>> with open("video.mp4", "rb") as f:
...     files = client.upload([("video.mp4", f, "video/mp4")])
>>>
>>> thread.send_files(files)
```

search_messages (*query*, *limit*)
Find and get message IDs by query.

Warning! If someone send a message to the thread that matches the query, while we're searching, some snippets will get returned twice.

This is fundamentally not fixable, it's just how the endpoint is implemented.

The returned message snippets are ordered by last sent first.

Parameters

- **query** (str) – Text to search for
- **limit** (int) – Max. number of message snippets to retrieve

Example

Fetch the latest message in the thread that matches the query.

```
>>> (message,) = thread.search_messages("abc", limit=1)
>>> message.text
"Some text and abc"
```

Return type `Iterable[MessageSnippet]`

`fetch_messages`(*limit*)
Fetch messages in a thread.

The returned messages are ordered by last sent first.

Parameters **`limit`** (`Optional[int]`) – Max. number of threads to retrieve. If `None`, all threads will be retrieved.

Example

```
>>> for message in thread.fetch_messages(limit=5)
...     print(message.text)
...
A message
Another message
None
A fourth message
```

Return type `Iterable[Message]`

`fetch_images`(*limit*)
Fetch images/videos posted in the thread.

The returned images are ordered by last sent first.

Parameters **`limit`** (`Optional[int]`) – Max. number of images to retrieve. If `None`, all images will be retrieved.

Example

```
>>> for image in thread.fetch_messages(limit=3)
...     print(image.id)
...
1234
2345
```

Return type `Iterable[Attachment]`

`set_nickname`(*user_id, nickname*)
Change the nickname of a user in the thread.

Parameters

- **`user_id`** (`str`) – User that will have their nickname changed

- **nickname** (str) – New nickname

Example

```
>>> thread.set_nickname("1234", "A nickname")
```

set_color (*color*)

Change thread color.

The new color must be one of the following:

```
"#0084ff", "#44bec7", "#ffc300", "#fa3c4c", "#d696bb", "#6699cc",
"#13cf13", "#ff7e29", "#e68585", "#7646ff", "#20cef5", "#67b868",
"#d4a88c", "#ff5ca1", "#a695c7", "#ff7ca8", "#1adb5b", "#f01d6a",
"#ff9c19" or "#0edcde".
```

This list is subject to change in the future!

The default when creating a new thread is `"#0084ff"`.

Parameters **color** (str) – New thread color

Example

Set the thread color to "Coral Pink".

```
>>> thread.set_color("#e68585")
```

set_emoji (*emoji*)

Change thread emoji.

Parameters **emoji** (Optional[str]) – New thread emoji. If None, will be set to the default "LIKE" icon

Example

Set the thread emoji to "".

```
>>> thread.set_emoji("")
```

forward_attachment (*attachment_id*)

Forward an attachment.

Parameters **attachment_id** (str) – Attachment ID to forward

Example

```
>>> thread.forward_attachment("1234")
```

start_typing ()

Set the current user to start typing in the thread.

Example

```
>>> thread.start_typing()
```

stop_typing()

Set the current user to stop typing in the thread.

Example

```
>>> thread.stop_typing()
```

create_plan(*name*, *at*, *location_name=None*, *location_id=None*)

Create a new plan.

# TODO: Arguments

Parameters

- **name** (str) – Name of the new plan
- **at** (datetime) – When the plan is for

Example

```
>>> thread.create_plan(...)
```

create_poll(*question*, *options*)

Create poll in a thread.

Parameters

- **question** (str) – The question
- **options** (Mapping[str, bool]) – Options and whether you want to select the option

Example

```
>>> thread.create_poll("Test poll", {"Option 1": True, "Option 2": False})
```

mute(*duration=None*)

Mute the thread.

Parameters **duration** (Optional[timedelta]) – Time to mute, use `None` to mute forever

Example

```
>>> import datetime
>>> thread.mute(datetime.timedelta(days=2))
```

unmute()
Unmute the thread.

Example

```
>>> thread.unmute()
```

mute_reactions()
Mute thread reactions.

unmute_reactions()
Unmute thread reactions.

mute_mentions()
Mute thread mentions.

unmute_mentions()
Unmute thread mentions.

mark_as_spam()
Mark the thread as spam, and delete it.

delete()
Delete the thread.

If you want to delete multiple threads, please use *Client.delete_threads*.

Example

```
>>> message.delete()
```

class fbchat.**Thread**(*, *session*, *id*)
Represents a Facebook thread, where the actual type is unknown.

Implements parts of *ThreadABC*, call the method to figure out if your use case is supported. Otherwise, you'll have to use an *User*/*Group*/*Page* object.

Note: This list may change in minor versions!

session
The session to use when making requests.

id
The unique identifier of the thread.

class fbchat.**Page**(*, *session*, *id*)
Represents a Facebook page. Implements *ThreadABC*.

Example

```
>>> page = fbchat.Page(session=session, id="1234")
```

session
The session to use when making requests.

id
The unique identifier of the page.

class fbchat.**User**(***, *session*, *id*)
Represents a Facebook user. Implements *ThreadABC*.

Example

```
>>> user = fbchat.User(session=session, id="1234")
```

session
The session to use when making requests.

id
The user's unique identifier.

confirm_friend_request()
Confirm a friend request, adding the user to your friend list.

Example

```
>>> user.confirm_friend_request()
```

remove_friend()
Remove the user from the client's friend list.

Example

```
>>> user.remove_friend()
```

block()
Block messages from the user.

Example

```
>>> user.block()
```

unblock()
Unblock a previously blocked user.

Example

```
>>> user.unblock()
```

class fbchat.**Group**(**, session, id*)

Represents a Facebook group. Implements *ThreadABC*.

Example

```
>>> group = fbchat.Group(session=session, id="1234")
```

session

The session to use when making requests.

id

The group's unique identifier.

add_participants(*user_ids*)

Add users to the group.

Parameters **user_ids** (Iterable[str]) – One or more user IDs to add

Example

```
>>> group.add_participants(["1234", "2345"])
```

remove_participant(*user_id*)

Remove user from the group.

Parameters **user_id** (str) – User ID to remove

Example

```
>>> group.remove_participant("1234")
```

add_admins(*user_ids*)

Set specified users as group admins.

Parameters **user_ids** (Iterable[str]) – One or more user IDs to set admin

Example

```
>>> group.add_admins(["1234", "2345"])
```

remove_admins(*user_ids*)

Remove admin status from specified users.

Parameters **user_ids** (Iterable[str]) – One or more user IDs to remove admin

Example

```
>>> group.remove_admins(["1234", "2345"])
```

set_title(*title*)
Change title of the group.

Parameters **title** (str) – New title

Example

```
>>> group.set_title("Abc")
```

set_image(*image_id*)
Change the group image from an image id.

Parameters **image_id** (str) – ID of uploaded image

Example

Upload an image, and use it as the group image.

```
>>> with open("image.png", "rb") as f:
...     (file,) = client.upload([("image.png", f, "image/png")])
...
>>> group.set_image(file[0])
```

set_approval_mode(*require_admin_approval*)
Change the group's approval mode.

Parameters **require_admin_approval** (bool) – True or False

Example

```
>>> group.set_approval_mode(False)
```

accept_users(*user_ids*)
Accept users to the group from the group's approval.

Parameters **user_ids** (Iterable[str]) – One or more user IDs to accept

Example

```
>>> group.accept_users(["1234", "2345"])
```

deny_users(*user_ids*)
Deny users from joining the group.

Parameters **user_ids** (Iterable[str]) – One or more user IDs to deny

Example

```
>>> group.deny_users(["1234", "2345"])
```

4.4.4 Thread Data

class fbchat.**PageData**(**, session, id, photo, name, last_active=None, message_count=None, plan=None, url=None, city=None, likes=None, sub_title=None, category=None*)

Represents data about a Facebook page.

Inherits `Page`, and implements `ThreadABC`.

photo
The page's picture

name
The name of the page

last_active
When the thread was last active / when the last message was sent

message_count
Number of messages in the thread

plan
Set `Plan`

url
The page's custom URL

city
The name of the page's location city

likes
Amount of likes the page has

sub_title
Some extra information about the page

category
The page's category

class fbchat.**UserData**(**, session, id, photo, name, is_friend, first_name, last_name=None, last_active=None, message_count=None, plan=None, url=None, gender=None, affinity=None, nickname=None, own_nickname=None, color=None, emoji=None*)

Represents data about a Facebook user.

Inherits `User`, and implements `ThreadABC`.

photo
The user's picture

name
The name of the user

is_friend
Whether the user and the client are friends

first_name
The users first name

last_name
The users last name

last_active
When the thread was last active / when the last message was sent

message_count
Number of messages in the thread

plan
Set *Plan*

url
The profile URL. `None` for Messenger-only users

gender
The user's gender

affinity
From 0 to 1. How close the client is to the user

nickname
The user's nickname

own_nickname
The clients nickname, as seen by the user

color
The message color

emoji
The default emoji

class fbchat.**GroupData**(**, session, id, photo=None, name=None, last_active=None, message_count=None, plan=None, participants=NOTHING, nicknames=NOTHING, color=None, emoji=None, admins=NOTHING, approval_mode=None, approval_requests=NOTHING, join_link=None*)
Represents data about a Facebook group.

Inherits *Group*, and implements *ThreadABC*.

photo
The group's picture

name
The name of the group

last_active
When the group was last active / when the last message was sent

message_count
Number of messages in the group

plan
Set *Plan*

participants
The group thread's participant user ids

nicknames
A dictionary, containing user nicknames mapped to their IDs

color
The groups's message color

emoji
The groups's default emoji

4.4.5 Messages

class fbchat.**Message**(**, thread, id*)
Represents a Facebook message.

Example

```
>>> thread = fbchat.User(session=session, id="1234")
>>> message = fbchat.Message(thread=thread, id="mid.$XYZ")
```

thread
The thread that this message belongs to.

id
The message ID.

property session
The session to use when making requests.

delete()
Delete the message (removes it only for the user).

If you want to delete multiple messages, please use *Client.delete_messages*.

Example

```
>>> message.delete()
```

unsend()
Unsend the message (removes it for everyone).

The message must to be sent by you, and less than 10 minutes ago.

Example

```
>>> message.unsend()
```

react(*reaction*)
React to the message, or removes reaction.

Currently, you can use "", "", "", "", "", "" or "". It should be possible to add support for more, but we haven't figured that out yet.

Parameters **reaction** (Optional[str]) – Reaction emoji to use, or if None, removes reaction.

Example

```
>>> message.react("")
```

fetch()
Fetch fresh *MessageData* object.

Example

```
>>> message = message.fetch()
>>> message.text
"The message text"
```

Return type MessageData

static **format_mentions**(*text*, **args*, ***kwargs*)
Like `str.format`, but takes tuples with a thread id and text instead.

Return a tuple, with the formatted string and relevant mentions.

```
>>> Message.format_mentions("Hey {!r}! My name is {}", ("1234", "Peter"), (
↪"4321", "Michael"))
("Hey 'Peter'! My name is Michael", [Mention(thread_id=1234, offset=4,␣
↪length=7), Mention(thread_id=4321, offset=24, length=7)])
```

```
>>> Message.format_mentions("Hey {p}! My name is {}", ("1234", "Michael"), p=(
↪"4321", "Peter"))
('Hey Peter! My name is Michael', [Mention(thread_id=4321, offset=4,␣
↪length=5), Mention(thread_id=1234, offset=22, length=7)])
```

class fbchat.**Mention**(***, *thread_id*, *offset*, *length*)
Represents a `@mention`.

```
>>> fbchat.Mention(thread_id="1234", offset=5, length=2)
Mention(thread_id="1234", offset=5, length=2)
```

thread_id
The thread ID the mention is pointing at

offset
The character where the mention starts

length
The length of the mention

class fbchat.**EmojiSize**(*Enum*)
Used to specify the size of a sent emoji.

LARGE = '369239383222810'

MEDIUM = '369239343222814'

SMALL = '369239263222822'

class fbchat.**MessageData**(**, thread, id, author, created_at, text=None, mentions=NOTHING, emoji_size=None, is_read=None, read_by=NOTHING, reactions=NOTHING, sticker=None, attachments=NOTHING, quick_replies=NOTHING, unsent=False, reply_to_id=None, replied_to=None, forwarded=False*)

Represents data in a Facebook message.

Inherits *Message*.

author
ID of the sender

created_at
When the message was sent

text
The actual message

mentions
A list of *Mention* objects

emoji_size
Size of a sent emoji

is_read
Whether the message is read

read_by
People IDs who read the message, only works with *ThreadABC.fetch_messages*

reactions
A dictionary with user's IDs as keys, and their reaction as values

sticker
A *Sticker*

attachments
A list of attachments

quick_replies
A list of *QuickReply*

unsent
Whether the message is unsent (deleted for everyone)

reply_to_id
Message ID you want to reply to

replied_to
Replied message

forwarded
Whether the message was forwarded

4.4.6 Exceptions

exception `fbchat.`**`FacebookError`**(*message*)

Base class for all custom exceptions raised by `fbchat`.

All exceptions in the module inherit this.

`message`

A message describing the error

exception `fbchat.`**`HTTPError`**(*message*, *status_code=None*)

Base class for errors with the HTTP(s) connection to Facebook.

`status_code`

The returned HTTP status code, if relevant

exception `fbchat.`**`ParseError`**(*message*, *data*)

Raised when we fail parsing a response from Facebook.

This may contain sensitive data, so should not be logged to file.

`data`

The data that triggered the error.

The format of this cannot be relied on, it's only for debugging purposes.

exception `fbchat.`**`NotLoggedIn`**(*message*)

Raised by Facebook if the client has been logged out.

exception `fbchat.`**`ExternalError`**(*message*, *description*, *code=None*)

Base class for errors that Facebook return.

`description`

The error message that Facebook returned (Possibly in the user's own language)

`code`

The error code that Facebook returned

exception `fbchat.`**`GraphQLError`**(*message*, *description*, *code=None*, *debug_info=None*)

Raised by Facebook if there was an error in the GraphQL query.

`debug_info`

Query debug information

exception `fbchat.`**`InvalidParameters`**(*message*, *description*, *code=None*)

Raised by Facebook if:

- Some function supplied invalid parameters.
- Some content is not found.
- Some content is no longer available.

exception `fbchat.`**`PleaseRefresh`**(*message*, *description*, *code=1357004*)

Raised by Facebook if the client has been inactive for too long.

This error usually happens after 1-2 days of inactivity.

4.4.7 Attachments

class `fbchat.`**`Attachment`**(*, *id=None*)

Represents a Facebook attachment.

`id`

The attachment ID

class `fbchat.`**`ShareAttachment`**(*, *id=None*, *author=None*, *url=None*, *original_url=None*, *title=None*, *description=None*, *source=None*, *image=None*, *original_image_url=None*, *attachments=NOTHING*)

Represents a shared item (e.g. URL) attachment.

`author`

ID of the author of the shared post

`url`

Target URL

`original_url`

Original URL if Facebook redirects the URL

`title`

Title of the attachment

`description`

Description of the attachment

`source`

Name of the source

`image`

The attached image

`original_image_url`

URL of the original image if Facebook uses `safe_image`

`attachments`

List of additional attachments

class `fbchat.`**`Sticker`**(*, *id=None*, *pack=None*, *is_animated=False*, *medium_sprite_image=None*, *large_sprite_image=None*, *frames_per_row=None*, *frames_per_col=None*, *frame_count=None*, *frame_rate=None*, *image=None*, *label=None*)

Represents a Facebook sticker that has been sent to a thread as an attachment.

`pack`

The sticker-pack's ID

`is_animated`

Whether the sticker is animated

`medium_sprite_image`

URL to a medium spritemap

`large_sprite_image`

URL to a large spritemap

`frames_per_row`

The amount of frames present in the spritemap pr. row

`frames_per_col`

The amount of frames present in the spritemap pr. column

frame_count
The total amount of frames in the spritemap

frame_rate
The frame rate the spritemap is intended to be played in

image
The sticker's image

label
The sticker's label/name

class fbchat.**LocationAttachment**(*, *id=None*, *latitude=None*, *longitude=None*, *image=None*, *url=None*, *address=None*)
Represents a user location.

Latitude and longitude OR address is provided by Facebook.

latitude
Latitude of the location

longitude
Longitude of the location

image
Image showing the map of the location

url
URL to Bing maps with the location

class fbchat.**LiveLocationAttachment**(*, *id=None*, *latitude=None*, *longitude=None*, *image=None*, *url=None*, *address=None*, *name=None*, *expires_at=None*, *is_expired=None*)
Represents a live user location.

name
Name of the location

expires_at
When live location expires

is_expired
True if live location is expired

class fbchat.**FileAttachment**(*, *id=None*, *url=None*, *size=None*, *name=None*, *is_malicious=None*)
Represents a file that has been sent as a Facebook attachment.

url
URL where you can download the file

size
Size of the file in bytes

name
Name of the file

is_malicious
Whether Facebook determines that this file may be harmful

class fbchat.**AudioAttachment**(*, *id=None*, *filename=None*, *url=None*, *duration=None*, *audio_type=None*)
Represents an audio file that has been sent as a Facebook attachment.

filename
Name of the file

url
URL of the audio file

duration
Duration of the audio clip

audio_type
Audio type

class fbchat.**ImageAttachment**(***, *id=None*, *original_extension=None*, *width=None*, *height=None*, *is_animated=None*, *previews=NOTHING*)
Represents an image that has been sent as a Facebook attachment.

To retrieve the full image URL, use: `Client.fetch_image_url`, and pass it the id of the image attachment.

original_extension
The extension of the original image (e.g. `png`)

width
Width of original image

height
Height of original image

is_animated
Whether the image is animated

previews
A set, containing variously sized / various types of previews of the image

class fbchat.**VideoAttachment**(***, *id=None*, *size=None*, *width=None*, *height=None*, *duration=None*, *preview_url=None*, *previews=NOTHING*)
Represents a video that has been sent as a Facebook attachment.

size
Size of the original video in bytes

width
Width of original video

height
Height of original video

duration
Length of video

preview_url
URL to very compressed preview video

previews
A set, containing variously sized previews of the video

class fbchat.**ImageAttachment**(***, *id=None*, *original_extension=None*, *width=None*, *height=None*, *is_animated=None*, *previews=NOTHING*)
Represents an image that has been sent as a Facebook attachment.

To retrieve the full image URL, use: `Client.fetch_image_url`, and pass it the id of the image attachment.

original_extension
The extension of the original image (e.g. `png`)

width
Width of original image

height
Height of original image

is_animated
Whether the image is animated

previews
A set, containing variously sized / various types of previews of the image

4.4.8 Events

class fbchat.**Listener**(**, session, chat_on, foreground, mqtt=NOTHING, sync_token=None, sequence_id=None, tmp_events=NOTHING*)
Listen to incoming Facebook events.

Initialize a connection to the Facebook MQTT service.

Parameters

- **session** (`Session`) – The session to use when making requests.
- **chat_on** (`bool`) – Whether . . .
- **foreground** (`bool`) – Whether . . .

Example

```
>>> listener = fbchat.Listener(session, chat_on=True, foreground=True)
```

listen()
Run the listening loop continually.

This is a blocking call, that will yield events as they arrive.

This will automatically reconnect on errors, except if the errors are one of *`PleaseRefresh`* or *`NotLoggedIn`*.

Example

Print events continually.

```
>>> for event in listener.listen():
...     print(event)
```

Return type `Iterable[Event]`

disconnect()
Disconnect the MQTT listener.

Can be called while listening, which will stop the listening loop.

The *`Listener`* object should not be used after this is called!

Example

Stop the listener when receiving a message with the text "/stop"

```
>>> for event in listener.listen():
...     if isinstance(event, fbchat.MessageEvent):
...         if event.message.text == "/stop":
...             listener.disconnect()  # Almost the same "break"
```

Return type `None`

`set_foreground`(*value*)
Set the `foreground` value while listening.

Return type `None`

`set_chat_on`(*value*)
Set the `chat_on` value while listening.

Return type `None`

4.4.9 Miscellaneous

`class fbchat.`**`ThreadLocation`**(*Enum*)
Used to specify where a thread is located (inbox, pending, archived, other).

`INBOX = 'INBOX'`

`PENDING = 'PENDING'`

`ARCHIVED = 'ARCHIVED'`

`OTHER = 'OTHER'`

`class fbchat.`**`ActiveStatus`**(**, active, last_active=None, in_game=None*)

`active`
Whether the user is active now

`last_active`
When the user was last active

`in_game`
Whether the user is playing Messenger game now

`class fbchat.`**`QuickReply`**(**, payload=None, external_payload=None, data=None, is_response=False*)
Represents a quick reply.

`payload`
Payload of the quick reply

`external_payload`
External payload for responses

`data`
Additional data

`is_response`
Whether it's a response for a quick reply

class fbchat.**QuickReplyText**(**, payload=None, external_payload=None, data=None, is_response=False, title=None, image_url=None*)

Represents a text quick reply.

title
Title of the quick reply

image_url
URL of the quick reply image

class fbchat.**QuickReplyLocation**(**, payload=None, external_payload=None, data=None, is_response=False*)

Represents a location quick reply (Doesn't work on mobile).

class fbchat.**QuickReplyPhoneNumber**(**, payload=None, external_payload=None, data=None, is_response=False, image_url=None*)

Represents a phone number quick reply (Doesn't work on mobile).

image_url
URL of the quick reply image

class fbchat.**QuickReplyEmail**(**, payload=None, external_payload=None, data=None, is_response=False, image_url=None*)

Represents an email quick reply (Doesn't work on mobile).

image_url
URL of the quick reply image

class fbchat.**Poll**(**, session, id, question, options, options_count*)

Represents a poll.

session
ID of the poll

id
ID of the poll

question
The poll's question

options
The poll's top few options. The full list can be fetched with `fetch_options`

options_count
Options count

fetch_options()
Fetch all `PollOption` objects on the poll.

The result is ordered with options with the most votes first.

Example

```
>>> options = poll.fetch_options()
>>> options[0].text
"An option"
```

Return type Sequence[PollOption]

set_votes(*option_ids, new_options=None*)
Update the user's poll vote.

Parameters

- **option_ids** (Iterable[str]) – Option ids to vote for / keep voting for
- **new_options** (Optional[Iterable[str]]) – New options to add

Example

```
>>> options = poll.fetch_options()
>>> # Add option
>>> poll.set_votes([o.id for o in options], new_options=["New option"])
>>> # Remove vote from option
>>> poll.set_votes([o.id for o in options if o.text != "Option 1"])
```

class fbchat.**PollOption**(**, id, text, vote, voters, votes_count*)

Represents a poll option.

id

ID of the poll option

text

Text of the poll option

vote

Whether vote when creating or client voted

voters

ID of the users who voted for this poll option

votes_count

Votes count

class fbchat.**Plan**(**, session, id*)

Base model for plans.

Example

```
>>> plan = fbchat.Plan(session=session, id="1234")
```

session

The session to use when making requests.

id

The plan's unique identifier.

fetch()

Fetch fresh *PlanData* object.

Example

```
>>> plan = plan.fetch()
>>> plan.title
"A plan"
```

Return type `PlanData`

edit (*name*, *at*, *location_name=None*, *location_id=None*)
Edit the plan.

# TODO: Arguments

delete ()
Delete the plan.

Example

```
>>> plan.delete()
```

participate ()
Set yourself as GOING/participating to the plan.

Example

```
>>> plan.participate()
```

decline ()
Set yourself as having DECLINED the plan.

Example

```
>>> plan.decline()
```

class fbchat.**PlanData** (**, session, id, time, title, location=None, location_id=None, author_id=None, guests=None*)
Represents data about a plan.

time
Plan time, only precise down to the minute

title
Plan title

location
Plan location name

location_id
Plan location ID

author_id
ID of the plan creator

guests
User ids mapped to their *GuestStatus*

property going

List of the *User* IDs who will take part in the plan.

Return type Sequence[str]

property declined

List of the *User* IDs who won't take part in the plan.

Return type Sequence[str]

property invited

List of the *User* IDs who are invited to the plan.

Return type Sequence[str]

class fbchat.**GuestStatus**(*Enum*)

An enumeration.

INVITED = 1

GOING = 2

DECLINED = 3

PYTHON MODULE INDEX

INDEX