pytwitcherapi Documentation

Release 0.9.3

David Zuber

Aug 27, 2017

Contents

This project is not developed anymore!

Python API for interacting with twitch.tv.

CHAPTER 1

Features

- Easy-to-use object oriented high-level API
- Search and query information about games, channels, streams and users
- Get the livestream playlist
- OAauth Authentication. Can retrieve followed streams and more...
- Good documentation and test coverage
- IRC client for the chat (with IRC v3 tag support)

Welcome to pytwitcherapi‘s documentation!

CHAPTER 2

Contents:

Installation

At the command line either via easy_install or pip:

```
$ easy_install pytwitcherapi
$ pip install pytwitcherapi
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv pytwitcherapi
$ pip install pytwitcherapi
```

Usage

This is a litte quickstart guide. For more information go to the Usermanual.

API requests

`pytwitcherapi.TwitchSession` class is the central class for interacting with twitch.tv:

```
1 import pytwitcherapi
2
3 ts = pytwitcherapi.TwitchSession()
```

To query all top games use:

```
5 topgames = ts.top_games()
```

Get streams and playlist for every game:

```
for game in topgames:
    streams = ts.get_streams(game=game)
    for stream in streams:
        channel = stream.channel
        playlist = ts.get_playlist(channel)
```

As you can see games, channels, streams are wrapped into objects. See `pytwitcherapi.Game`, `pytwitcherapi.Channel`, `pytwitcherapi.Stream`, `pytwitcherapi.User`.

You can use your own `Client-ID` for twitch by setting the environment variable `PYTWITCHER_CLIENT_ID`.

Usermanual

In here you find help for using the pytwitcherapi. There are also some simple examples.

Requests

API requests

`pytwitcherapi.TwitchSession` class is the central class for interacting with twitch.tv:

```
import pytwitcherapi

ts = pytwitcherapi.TwitchSession()
```

To query all top games use:

```
topgames = ts.top_games()
```

Get streams and playlist for every game:

```
for game in topgames:
    streams = ts.get_streams(game=game)
    for stream in streams:
        channel = stream.channel
        playlist = ts.get_playlist(channel)
```

As you can see games, channels, streams are wrapped into objects. See `pytwitcherapi.Game`, `pytwitcherapi.Channel`, `pytwitcherapi.Stream`, `pytwitcherapi.User`.

You can use your own `Client-ID` for twitch by setting the environment variable `PYTWITCHER_CLIENT_ID`.

Custom requests

You can also issue custom requests. The `pytwitcherapi.TwitchSession` is actually a subclass of `requests.Session`. So basically you can use `pytwitcherapi.TwitchSession.request()` to issue arbitrary requests. To make it easier to use the different twitch APIs there are a few helpers.

You can get easy access to three different twitch APIs:

- Kraken API witch uses *`pytwitcherapi.session.TWITCH_KRAKENURL`*. Use *`pytwitcherapi.session.TwitchSession.kraken_request()`*.

- Usher API with uses *pytwitcherapi.session.TWITCH_USHERURL*. Use *pytwitcherapi.session.TwitchSession.usher_request()*.
- The old twitch API *pytwitcherapi.session.TWITCH_APIURL*. Use *pytwitcherapi.session.TwitchSession.oldapi_request()*.

```
import pytwitcherapi

ts = pytwitcherapi.TwitchSession()

topgames = ts.top_games()

for game in topgames:
    streams = ts.get_streams(game=game)
    for stream in streams:
        channel = stream.channel
        playlist = ts.get_playlist(channel)
```

Authentication

For some methods of `pytwitcherapi.TwitchSession`, the user needs to grant pytwitcher authorization. Twitch Authentication is based on OAuth. We use the implicit grant workflow. In short, the user visits a website. Has to login, and allow pytwitcher. Twitch will redirect him to *pytwitcherapi.constants.REDIRECT_URI*. In the url fragment of that redirection, one can find the token we need. To make it simple for the user, here is what should be done for authentication:

- Call `pytwitcherapi.TwitchSession.start_login_server()`. This will create a thread that serves a server on *pytwitcherapi.constants.LOGIN_SERVER_ADRESS*. Once the user gets redirected, this server will pick up the request and extract the token:

```
import pytwitcherapi

ts = pytwitcherapi.TwitchSession()
ts.start_login_server()
```

- Get the url `pytwitcherapi.TwitchSession.get_auth_url()` and send the user to visit that url in his favorite webbrowser. He might have to login, and allow pytwitcher, if he did not already:

```
import webbrowser
url = ts.get_auth_url()
webbrowser.open(url)
```

- Wait until the user finished login in. Then call `pytwitcherapi.TwitchSession.shutdown_login_server()` to shutdown the server and join the thread:

```
raw_input("Press ENTER when finished")
ts.shutdown_login_server()
```

- Check if the user authorized the session with `pytwitcherapi.TwitchSession.authorized()`:

```
assert ts.authorized, "Authorization failed! Did the user allow it?"
print "Login User: %s" % ts.current_user
```

- Now you can call methods that require authentication:

```
streams = ts.followed_streams()
```

Chat

The twitch chat is based on the IRC protocol RFC1459. The official documentation on the twitch chat is here: Twitch IRC Doc. The irc python lib might also be useful, because we use it as backend.

`pytwitcherapi.IRCClient` is a simple client, which can only connect to one channel/server at a time. When building applications, you probably wanna run the IRCClient in another thread, so it doesn't block your application. The client is thread safe and has quite a few methods to send IRC commands, if the client is running in another thread. They are wrapped versions of methods from `irc.client.ServerConnection`. E.g. you can simply call `pytwitcherapi.IRCClient.quit()` from another thread. Here is a simple example, where we send messages to the channel. Change `input` to `raw_input` for python 2:

```
import threading

import pytwitcherapi

session = ...  # we assume an authenticated TwitchSession
channel = session.get_channel('somechannel')
client = pytwitcherapi.IRCClient(session, channel)
t = threading.Thread(target=client.process_forever)
t.start()

try:
    while True:
        m = input('Send Message:')
        if not m: break;
        # will be processed in other thread
        # sends a message to the server
        client.send_msg(m)
finally:
    client.shutdown()
    t.join()
```

Important: The connection will wait/block if you send more messages than twitch allows. See `pytwitcherapi.chat.ServerConnection3`.

You can make the client handle different IRC events. Subclass the client and create a method `on_<eventtype>`. For example to greet everyone who joins an IRC channel:

```
import pytwitcherapi

class MyIRCClient(pytwitcherapi.IRCClient):

    def on_join(self, connection, event):
        """Handles the join event and greets everone

        :param connection: the connection with the event
        :type connection: :class:`irc.client.ServerConnection`
        :param event: the event to handle
        :type event: :class:`irc.client.Event`
        :returns: None
        """
```

```
        target = event.source
        self.privmsg(target, 'Hello %s!' % target)
```

If you override `pytwitcherapi.IRCClient.on_pubmsg()` or `pytwitcherapi.IRCClient.on_privmsg()` make sure to call the super method:

```
from pytwitcherapi import chat

class MyIRCClient(pytwitcherapi.IRCClient):

    def on_privmsg(self, connection, event):
        super(MyIRCClient, self).on_privmsg(connection, event)

        print chat.Message3.from_event(event)
```

But printing out messages is not really useful. You probably want to access them in another thread. All private and public messages are stored in a thread safe message queue. By default the queue stores the last 100 messages. You can alter the queuesize when creating a client. `0` will make the queue store all messages.

Note: The Client is using two connections. One for sending messages (`pytwitcherapi.IRCClient.in_connection`) and one for receiving (`pytwitcherapi.IRCClient.in_connection`) them. With one message, you wouldn't revceive your own messages processed from the server (with tags).

Here is a little example. To quit press `CTRL-C`:

```
import threading
import queue  # Queue for python 2

import pytwitcherapi

session = ...  # we assume an authenticated TwitchSession
channel = session.get_channel('somechannel')
client = pytwitcherapi.IRCClient(session, channel, queuesize=0)
t = threading.Thread(target=client.process_forever)
t.start()

try:
    while True:
        try:
            m = client.messages.get(block=False)
        except queue.Empty:
            pass
        else:
            # Now you have the message in the main thread
            # and can display the message in the
            # GUI or wherever you want
            print "Message from %s to %s: %s" % (m.source, m.target, m.text)
finally:
    client.shutdown()
    t.join()
```

Tags and metadata

Twitch does support tags. Tags store metadata about a message, like the color of the user, whether he is a subscriber, the `pytwichterapi.chat.Emote` etc. These messages get safed id the message queue: `pytwitcherapi.`

`IRCClient.messages`. See the `pytwitcherapi.chat.Message3` documentation for the additional metadata.

Here is a little example. To quit press `CTRL-C`:

```
import threading
import queue  # Queue for python 2

import pytwitcherapi

session = ...  # we assume an authenticated TwitchSession
channel = session.get_channel('somechannel')
client = pytwitcherapi.IRCClient(session, channel, queuesize=0)
t = threading.Thread(target=client.process_forever)
t.start()

try:
    while True:
        try:
            m = client.messages.get(block=False)
        except queue.Empty:
            pass
        else:
            print m.color
            print m.subscriber
            print m.turbo
            print m.emotes
            print m.user_type
finally:
    client.shutdown()
    t.join()
```

Developer's Documentation

Welcome to the developer's documenation. All necessary information for contributors who want to extend the project.

Documentation

Build

To build the documentation locally, follow these instructions:

1. Go to the root directory of this project.
2. Install the requirements:

   ```
   $ pip install -r docs/requirements.txt
   ```

   This will install sphinx, some required packages and the project in development mode (`pip install -e .`). That's why you have to be in the root dir.

3. Go to the docs dir:

   ```
   $ cd docs
   ```

4. Invoke sphinx build:

```
$ sphinx-build -b html -d _build/doctrees   source _build/html
```

or alternativly with make:

```
$ make html
```

on windows:

```
$ make.bat html
```

Reference

Automatic generated Documenation by apidoc and autodoc.

pytwitcherapi

Subpackages

pytwitcherapi.chat

Submodules

pytwitcherapi.chat.client

IRC client for interacting with the chat of a channel.

Classes

ChatServerStatus(server[, ip, port, status, ...])	Useful for comparing the performance of servers.
IRCClient(session, channel[, queuesize])	Simple IRC client which can connect to a single pytwitcherapi.Channel.
Reactor([on_connect, on_disconnect])	Reactor that can exit the process_forever loop.
Reactor3([on_connect, on_disconnect])	Reactor that uses irc v3 connections

pytwitcherapi.chat.client.ChatServerStatus

class pytwitcherapi.chat.client.**ChatServerStatus**(*server*, *ip=None*, *port=None*, *status=None*, *errors=None*, *lag=None*, *description=''*, ***kwargs*)

Bases: object

Useful for comparing the performance of servers.

You can query the status of twitch chat servers. This class can easily wrap the result and sort the servers.

__init__(*server*, *ip=None*, *port=None*, *status=None*, *errors=None*, *lag=None*, *description=''*, ***kwargs*)

Initialize a chat server status.

Parameters

- **server** (str) – the server address including port. E.g. "0.0.0.0:80"
- **ip** (str) – the ip address
- **port** (int) – the port number
- **status** (str) – the server status. E.g. "offline", "online"
- **errors** (int) – the amount of errors in the last 5 min.
- **lag** (int) – the latency in ms
- **description** (str) – Wheter it is a main chat server or event server or group chat.

All other keyword arguments are ignored.

Methods

__init__(server[, ip, port, status, errors, ...])	Initialize a chat server status.

pytwitcherapi.chat.client.IRCClient

class pytwitcherapi.chat.client.**IRCClient**(*session*, *channel*, *queuesize=100*)

Bases: irc.client.SimpleIRCClient

Simple IRC client which can connect to a single pytwitcherapi.Channel.

You need an authenticated session with scope chat_login. Call *IRCClient.process_forever()* to start the event loop. This will block the current thread though. Calling *IRCClient.shutdown()* will stop the loop.

There are a lot of methods that can make the client send commands while the client is in its event loop. These methods are wrapped ones of irc.client.ServerConnection. They will always use *IRCClient.out_connection*!

You can implement handlers for all sorts of events by subclassing and creating a method called on_<event.type>. Note that *IRCClient.out_connection* will only get to the *IRCClient.on_welcome()* event (and then join a channel) and the IRCClient.on_join() event. For all other events, the *IRCClient.in_connection* will handle it and the other one will ignore it. This behaviour is implemented in IRCClient._dispatcher()

Little example with threads. Change input to raw_input for python 2:

```
import threading

from pytwitcherapi import chat

session = ...  # we assume an authenticated TwitchSession
channel = session.get_channel('somechannel')
client = chat.IRCClient(session, channel)
t = threading.Thread(target=client.process_forever,
                     kwargs={'timeout': 0.2})
t.start()

try:
    while True:
        m = input('Send Message:')
```

```
        if not m: break;
        # will be processed in other thread
        client.send_msg(m)
finally:
    client.shutdown()
    t.join()
```

__init__(*session*, *channel*, *queuesize=100*)

Initialize a new irc client which can connect to the given channel.

Parameters

- **session** (pytwitcherapi.TwitchSession) – a authenticated session. Used for quering the right server and the login username.
- **channel** (pytwitcherapi.Channel) – a channel
- **queuesize** (int) – The queuesize for storing messages in *IRCClient.messages*. If 0, unlimited size.

Raises exceptions.NotAuthorizedError

Methods

__init__(session, channel[, queuesize])	Initialize a new irc client which can connect to the given channel.
action(*args, **kwargs)	Send a CTCP ACTION command.
admin(*args, **kwargs)	Send an ADMIN command.
cap(*args, **kwargs)	Send a CAP command according to the spec.
connect(*args, **kwargs)	Connect using the underlying connection
ctcp(*args, **kwargs)	Send a CTCP command.
ctcp_reply(*args, **kwargs)	Send a CTCP REPLY command.
dcc_connect(address, port[, dcctype])	Connect to a DCC peer.
dcc_listen([dcctype])	Listen for connections from a DCC peer.
globops(*args, **kwargs)	Send a GLOBOPS command.
info(*args, **kwargs)	Send an INFO command.
invite(*args, **kwargs)	Send an INVITE command.
ison(*args, **kwargs)	Send an ISON command.
join(*args, **kwargs)	Send a JOIN command.
kick(*args, **kwargs)	Send a KICK command.
links(*args, **kwargs)	Send a LINKS command.
list(*args, **kwargs)	Send a LIST command.
lusers(*args, **kwargs)	Send a LUSERS command.
mode(*args, **kwargs)	Send a MODE command.
motd(*args, **kwargs)	Send an MOTD command.
names(*args, **kwargs)	Send a NAMES command.
negotiate_capabilities(connection)	Send *IRCClient.capabilities* to the server.
nick(*args, **kwargs)	Send a NICK command.
notice(*args, **kwargs)	Send a NOTICE command.
on_privmsg(connection, event)	Handle the private message event
on_pubmsg(connection, event)	Handle the public message event
on_welcome(connection, event)	Handle the welcome event
oper(*args, **kwargs)	Send an OPER command.

Continued on next page

Table 2.3 – continued from previous page

part(*args, **kwargs)	Send a PART command.
pass_(*args, **kwargs)	Send a PASS command.
ping(*args, **kwargs)	Send a PING command.
pong(*args, **kwargs)	Send a PONG command.
privmsg(*args, **kwargs)	Send a PRIVMSG command.
privmsg_many(*args, **kwargs)	Send a PRIVMSG command to multiple targets.
quit(*args, **kwargs)	Send a QUIT command.
send_msg(message)	Send the given message to the channel
send_raw(*args, **kwargs)	Send raw string to the server.
squit(*args, **kwargs)	Send an SQUIT command.
start()	Start the IRC client.
stats(*args, **kwargs)	Send a STATS command.
store_message(connection, event)	Store the message of event in IRCClient.messages.
time(*args, **kwargs)	Send a TIME command.
topic(*args, **kwargs)	Send a TOPIC command.
trace(*args, **kwargs)	Send a TRACE command.
user(*args, **kwargs)	Send a USER command.
userhost(*args, **kwargs)	Send a USERHOST command.
users(*args, **kwargs)	Send a USERS command.
version(*args, **kwargs)	Send a VERSION command.
wallops(*args, **kwargs)	Send a WALLOPS command.
who(*args, **kwargs)	Send a WHO command.
whois(*args, **kwargs)	Send a WHOIS command.
whowas(*args, **kwargs)	Send a WHOWAS command.

Attributes

capabilities	List of irc capabilities
channel	The channel to connect to.

reactor_class
The reactor class which dispatches events

alias of Reactor3

capabilities = [':twitch.tv/membership', ':twitch.tv/commands', ':twitch.tv/tags']
List of irc capabilities

in_connection = None
Connection that receives messages

out_connection = None
Connection that sends messages

session = None
an authenticated session. Used for quering the right server and the login username.

login_user = None
The user that is used for logging in to the chat

shutdown = None
Call this method for shutting down the client. This is thread safe.

process_forever = None
Call this method to process messages until shutdown() is called.

Parameters **timeout** (float) – timeout for waiting on data in seconds

messages = None
A queue which stores all private and public *pytwitcherapi.chat.message.Message3*. Usefull for accessing messages from another thread.

channel
The channel to connect to. When setting the channel, automatically connect to it. If channel is None, disconnect.

on_welcome(*connection*, *event*)
Handle the welcome event

Automatically join the channel.

Parameters

- **connection** (irc.client.ServerConnection) – the connection with the event
- **event** (irc.client.Event) – the event to handle

Returns None

negotiate_capabilities(*connection*)
Send *IRCClient.capabilities* to the server.

Parameters **connection** (irc.client.ServerConnection) – the connection to use for sending

Returns None

Return type None

Raises None

store_message(*connection*, *event*)
Store the message of event in *IRCClient.messages*.

Parameters

- **connection** (irc.client.ServerConnection) – the connection with the event
- **event** (irc.client.Event) – the event to handle

Returns None

on_pubmsg(*connection*, *event*)
Handle the public message event

This stores the message in *IRCClient.messages* via *IRCClient.store_message()*.

Parameters

- **connection** (irc.client.ServerConnection) – the connection with the event
- **event** (irc.client.Event) – the event to handle

Returns None

on_privmsg(*connection*, *event*)
Handle the private message event

This stores the message in *IRCClient.messages* via *IRCClient.store_message()*.

Parameters

- **connection** (irc.client.ServerConnection) – the connection with the event
- **event** (irc.client.Event) – the event to handle

Returns None

send_msg(*message*)
Send the given message to the channel

This is a convenience method for *IRCClient.privmsg()*, which uses the current channel as target. This method is thread safe and can be called from another thread even if the client is running in *IRCClient.process_forever()*.

Parameters **message** (str) – The message to send

Returns None

Return type None

Raises None

action(**args*, ***kwargs*)
Send a CTCP ACTION command.

admin(**args*, ***kwargs*)
Send an ADMIN command.

cap(**args*, ***kwargs*)
Send a CAP command according to the spec.

Arguments:

subcommand – LS, LIST, REQ, ACK, CLEAR, END args – capabilities, if required for given subcommand

Example:

.cap('LS') .cap('REQ', 'multi-prefix', 'sasl') .cap('END')

ctcp(**args*, ***kwargs*)
Send a CTCP command.

ctcp_reply(**args*, ***kwargs*)
Send a CTCP REPLY command.

globops(**args*, ***kwargs*)
Send a GLOBOPS command.

info(**args*, ***kwargs*)
Send an INFO command.

invite(**args*, ***kwargs*)
Send an INVITE command.

ison(**args*, ***kwargs*)
Send an ISON command.

Arguments:

nicks – List of nicks.

join(**args*, ***kwargs*)
Send a JOIN command.

kick(**args*, ***kwargs*)
Send a KICK command.

links(**args*, ***kwargs*)
Send a LINKS command.

list(**args*, ***kwargs*)
Send a LIST command.

lusers(**args*, ***kwargs*)
Send a LUSERS command.

mode(**args*, ***kwargs*)
Send a MODE command.

motd(**args*, ***kwargs*)
Send an MOTD command.

names(**args*, ***kwargs*)
Send a NAMES command.

nick(**args*, ***kwargs*)
Send a NICK command.

notice(**args*, ***kwargs*)
Send a NOTICE command.

oper(**args*, ***kwargs*)
Send an OPER command.

part(**args*, ***kwargs*)
Send a PART command.

pass_(**args*, ***kwargs*)
Send a PASS command.

ping(**args*, ***kwargs*)
Send a PING command.

pong(**args*, ***kwargs*)
Send a PONG command.

privmsg(**args*, ***kwargs*)
Send a PRIVMSG command.

privmsg_many(**args*, ***kwargs*)
Send a PRIVMSG command to multiple targets.

quit(**args*, ***kwargs*)
Send a QUIT command.

send_raw(**args*, ***kwargs*)
Send raw string to the server.

The string will be padded with appropriate CR LF.

squit(**args*, ***kwargs*)
Send an SQUIT command.

stats(**args*, ***kwargs*)
Send a STATS command.

time(**args*, ***kwargs*)
Send a TIME command.

topic(**args*, ***kwargs*)
Send a TOPIC command.

trace(**args*, ***kwargs*)
Send a TRACE command.

user(**args*, ***kwargs*)
Send a USER command.

userhost(**args*, ***kwargs*)
Send a USERHOST command.

users(**args*, ***kwargs*)
Send a USERS command.

version(**args*, ***kwargs*)
Send a VERSION command.

wallops(**args*, ***kwargs*)
Send a WALLOPS command.

who(**args*, ***kwargs*)
Send a WHO command.

whois(**args*, ***kwargs*)
Send a WHOIS command.

whowas(**args*, ***kwargs*)
Send a WHOWAS command.

pytwitcherapi.chat.client.Reactor

class pytwitcherapi.chat.client.**Reactor**(*on_connect=<function __do_nothing>*, *on_disconnect=<function __do_nothing>*)

Bases: irc.client.Reactor

Reactor that can exit the process_forever loop.

The reactor is responsible for managing the connections, and handling the events that come in to the connections.

Simply call *Reactor.shutdown()* while the reactor is in a loop.

For more information see irc.client.Reactor.

__init__(*on_connect=<function __do_nothing>*, *on_disconnect=<function __do_nothing>*)
Initialize a reactor.

Parameters

- **on_connect** – optional callback invoked when a new connection is made.
- **on_disconnect** – optional callback invoked when a socket is disconnected.

Methods

__init__([on_connect, on_disconnect])	Initialize a reactor.
add_global_handler(event, handler[, priority])	Adds a global handler function for a specific event type.
dcc([dcctype])	Creates and returns a DCCConnection object.
disconnect_all([message])	Disconnects all connections.
process_data(sockets)	Called when there is more data to read on connection sockets.

Continued on next page

Table 2.5 – continued from previous page

process_forever([timeout])	Run an infinite loop, processing data from connections.
process_once([timeout])	Process data from connections once.
process_timeout()	Called when a timeout notification is due.
remove_global_handler(event, handler)	Removes a global handler function.
server()	Creates and returns a ServerConnection object.
shutdown()	Disconnect all connections and end the loop

Attributes

sockets

process_forever(*timeout=0.2*)
Run an infinite loop, processing data from connections.

This method repeatedly calls process_once.

Parameters **timeout** (float) – Parameter to pass to irc.client.Reactor.process_once()

shutdown()
Disconnect all connections and end the loop

Returns None

Return type None

Raises None

pytwitcherapi.chat.client.Reactor3

class pytwitcherapi.chat.client.**Reactor3**(*on_connect=<function __do_nothing>*, *on_disconnect=<function __do_nothing>*)
Bases: *pytwitcherapi.chat.client.Reactor*

Reactor that uses irc v3 connections

Uses the ServerConnection3 class for connections. They support Event3 with tags.

__init__(*on_connect=<function __do_nothing>*, *on_disconnect=<function __do_nothing>*)
Initialize a reactor.

Parameters

- **on_connect** – optional callback invoked when a new connection is made.
- **on_disconnect** – optional callback invoked when a socket is disconnected.

Methods

__init__([on_connect, on_disconnect])	Initialize a reactor.
add_global_handler(event, handler[, priority])	Adds a global handler function for a specific event type.
dcc([dcctype])	Creates and returns a DCCConnection object.
disconnect_all([message])	Disconnects all connections.

Continued on next page

Table 2.7 – continued from previous page

process_data(sockets)	Called when there is more data to read on connection sockets.
process_forever([timeout])	Run an infinite loop, processing data from connections.
process_once([timeout])	Process data from connections once.
process_timeout()	Called when a timeout notification is due.
remove_global_handler(event, handler)	Removes a global handler function.
server()	Creates and returns a ServerConnection
shutdown()	Disconnect all connections and end the loop

Attributes

sockets

server()
Creates and returns a ServerConnection

Returns a server connection

Return type connection.ServerConnection3

Raises None

Functions

add_serverconnection_methods(cls)	Add a bunch of methods to an irc.client.SimpleIRCClient to send commands and messages.

Data

absolute_import	
log	Instances of the Logger class represent a single logging channel.

pytwitcherapi.chat.client.**add_serverconnection_methods**(*cls*)
Add a bunch of methods to an irc.client.SimpleIRCClient to send commands and messages.

Basically it wraps a bunch of methdos from irc.client.ServerConnection to be irc.schedule.IScheduler.execute_after(). That way, you can easily send, even if the IRCClient is running in *IRCClient.process_forever* in another thread.

On the plus side you can use positional and keyword arguments instead of just positional ones.

Parameters **cls** (irc.client.SimpleIRCClient) – The class to add the methods do.

Returns None

pytwitcherapi.chat.client.**absolute_import** = _Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)

pytwitcherapi.chat.client.**log** = <logging.Logger object>
Instances of the Logger class represent a single logging channel. A "logging channel" indicates an area of an application. Exactly how an "area" is defined is up to the application developer. Since an application can have

any number of areas, logging channels are identified by a unique string. Application areas can be nested (e.g. an area of "input processing" might include sub-areas "read CSV files", "read XLS files" and "read Gnumeric files"). To cater for this natural nesting, channel names are organized into a namespace hierarchy where levels are separated by periods, much like the Java or Python package namespace. So in the instance given above, channel names might be "input" for the upper level, and "input.csv", "input.xls" and "input.gnu" for the sub-levels. There is no arbitrary limit to the depth of nesting.

pytwitcherapi.chat.connection

Classes

Event3(type, source, target[, arguments, tags])	An IRC event with tags
ServerConnection3(reactor[, msglimit, ...])	ServerConncetion that can handle irc v3 tags

pytwitcherapi.chat.connection.Event3

class pytwitcherapi.chat.connection.**Event3**(*type*, *source*, *target*, *arguments=None*, *tags=None*)

Bases: irc.client.Event

An IRC event with tags

See tag specification.

__init__(*type*, *source*, *target*, *arguments=None*, *tags=None*)

Initialize a new event

Parameters

- **type** (str) – a string describing the event
- **source** (irc.client.NickMask | str) – The originator of the event. NickMask or server
- **target** (str) – The target of the event
- **arguments** (list | None) – Any specific event arguments

Raises None

Methods

__init__(type, source, target[, arguments, tags])	Initialize a new event

pytwitcherapi.chat.connection.ServerConnection3

class pytwitcherapi.chat.connection.**ServerConnection3**(*reactor*, *msglimit=20*, *limitinterval=30*)

Bases: irc.client.ServerConnection

ServerConncetion that can handle irc v3 tags

Tags are only handled for privmsg, pubmsg, notice events. All other events might be handled the old way.

__init__(*reactor*, *msglimit=20*, *limitinterval=30*)
Initialize a connection that has a limit to sending messages

Parameters

- **reactor** (irc.client.Reactor) – the reactor of the connection
- **msglimit** (int) – the maximum number of messages to send in limitinterval
- **limitinterval** (int) – the timeframe in seconds in which you can only send as many messages as in msglimit

Raises None

Methods

__init__(reactor[, msglimit, limitinterval])	Initialize a connection that has a limit to sending messages
action(target, action)	Send a CTCP ACTION command.
add_global_handler(*args)	Add global handler.
admin([server])	Send an ADMIN command.
as_nick(*args, **kwds)	Set the nick for the duration of the context.
cap(subcommand, *args)	Send a CAP command according to the spec.
close()	Close the connection.
connect(*args, **kwargs)	Connect/reconnect to a server.
ctcp(ctcptype, target[, parameter])	Send a CTCP command.
ctcp_reply(target, parameter)	Send a CTCP REPLY command.
disconnect([message])	Hang up the connection.
get_nickname()	Get the (real) nick name.
get_server_name()	Get the (real) server name.
get_waittime()	Return the appropriate time to wait, if we sent too many messages
globops(text)	Send a GLOBOPS command.
info([server])	Send an INFO command.
invite(nick, channel)	Send an INVITE command.
is_connected()	Return connection status.
ison(nicks)	Send an ISON command.
join(channel[, key])	Send a JOIN command.
kick(channel, nick[, comment])	Send a KICK command.
links([remote_server, server_mask])	Send a LINKS command.
list([channels, server])	Send a LIST command.
lusers([server])	Send a LUSERS command.
mode(target, command)	Send a MODE command.
motd([server])	Send an MOTD command.
names([channels])	Send a NAMES command.
nick(newnick)	Send a NICK command.
notice(target, text)	Send a NOTICE command.
oper(nick, password)	Send an OPER command.
part(channels[, message])	Send a PART command.
pass_(password)	Send a PASS command.
ping(target[, target2])	Send a PING command.
pong(target[, target2])	Send a PONG command.
privmsg(target, text)	Send a PRIVMSG command.

Continued on next page

Table 2.13 – continued from previous page

privmsg_many(targets, text)	Send a PRIVMSG command to multiple targets.
process_data()	read and process input from self.socket
quit([message])	Send a QUIT command.
reconnect()	Reconnect with the last arguments passed to self.connect()
remove_global_handler(*args)	Remove global handler.
send_items(*items)	Send all non-empty items, separated by spaces.
send_raw(string)	Send raw string to the server.
set_keepalive(interval)	Set a keepalive to occur every interval on this connection.
set_rate_limit(frequency)	Set a *frequency* limit (messages per second) for this connection.
squit(server[, comment])	Send an SQUIT command.
stats(statstype[, server])	Send a STATS command.
time([server])	Send a TIME command.
topic(channel[, new_topic])	Send a TOPIC command.
trace([target])	Send a TRACE command.
user(username, realname)	Send a USER command.
userhost(nicks)	Send a USERHOST command.
users([server])	Send a USERS command.
version([server])	Send a VERSION command.
wallops(text)	Send a WALLOPS command.
who([target, op])	Send a WHO command.
whois(targets)	Send a WHOIS command.
whowas(nick[, max, server])	Send a WHOWAS command.

Attributes

socket

sentmessages = None
A queue with timestamps form the last sent messages. So we can track if we send to many messages.

limitinterval = None
the timeframe in seconds in which you can only send as many messages as in ServerConncetion3msglimit

get_waittime()
Return the appropriate time to wait, if we sent too many messages

Returns the time to wait in seconds

Return type float

Raises None

send_raw(*string*)
Send raw string to the server.

The string will be padded with appropriate CR LF. If too many messages are sent, this will call time.sleep() until it is allowed to send messages again.

Parameters **string** (str) – the raw string to send

Returns None

Raises irc.client.InvalidCharacters, irc.client.MessageTooLong, irc.client.ServerNotConnectedError

Data

log	Instances of the Logger class represent a single logging channel.

pytwitcherapi.chat.connection.**log = <logging.Logger object>**
Instances of the Logger class represent a single logging channel. A "logging channel" indicates an area of an application. Exactly how an "area" is defined is up to the application developer. Since an application can have any number of areas, logging channels are identified by a unique string. Application areas can be nested (e.g. an area of "input processing" might include sub-areas "read CSV files", "read XLS files" and "read Gnumeric files"). To cater for this natural nesting, channel names are organized into a namespace hierarchy where levels are separated by periods, much like the Java or Python package namespace. So in the instance given above, channel names might be "input" for the upper level, and "input.csv", "input.xls" and "input.gnu" for the sub-levels. There is no arbitrary limit to the depth of nesting.

pytwitcherapi.chat.message

Classes

Chatter(source)	A chat user object
Emote(emoteid, occurences)	Emote from the emotes tag
Message(source, target, text)	A messag object
Message3(source, target, text[, tags])	A message which stores information from irc v3 tags
Tag(name[, value, vendor])	An irc v3 tag

pytwitcherapi.chat.message.Chatter

class pytwitcherapi.chat.message.**Chatter**(*source*)
Bases: object

A chat user object

Stores information about a chat user (source of an ircevent).

See irc.client.NickMask for how the attributes are constructed.

__init__(*source*)
Initialize a new chatter

Parameters **source** – the source of an irc.client.Event. E.g. 'pinky!username@example.com'

Raises None

Methods

__init__(source)	Initialize a new chatter

full = None
The full name (nickname!user@host)

nickname = None
The irc nickname

user = None
The irc user

host = None
The irc host

userhost = None
The irc user @ irc host

pytwitcherapi.chat.message.Emote

class pytwitcherapi.chat.message.**Emote**(*emoteid*, *occurences*)
Bases: object

Emote from the emotes tag

An emote has an id and occurences in a message. So each emote is tied to a specific message.

You can get the pictures here:

```
``cdn.jtvnw.net/emoticons/v1/<emoteid>/1.0``
```

__init__(*emoteid*, *occurences*)
Initialize a new emote

Parameters

- **emoteid** (int) – The emote id
- **occurences** (list) – a list of occurences, e.g. [(0, 4), (8, 12)]

Raises None

Methods

__init__(emoteid, occurences)	Initialize a new emote
from_str(emotestr)	Create an emote from the emote tag key

emoteid = None
The emote identifier

occurences = None
A list of occurences, e.g. [(0, 4), (8, 12)]

classmethod from_str(*emotestr*)
Create an emote from the emote tag key

Parameters **emotestr** (str) – the tag key, e.g. '123:0-4'

Returns an emote

Return type *Emote*

Raises None

pytwitcherapi.chat.message.Message

class pytwitcherapi.chat.message.**Message**(*source*, *target*, *text*)
Bases: object

A messag object

Can be a private|public message from a server or user.

__init__(*source*, *target*, *text*)
Initialize a new message from source to target with the given text

Parameters

- **source** (*Chatter*) – The source chatter
- **target** (*str*) – the target
- **text** (str) – the content of the message

Raises None

Methods

__init__(source, target, text)	Initialize a new message from source to target with the given text

pytwitcherapi.chat.message.Message3

class pytwitcherapi.chat.message.**Message3**(*source*, *target*, *text*, *tags=None*)
Bases: *pytwitcherapi.chat.message.Message*

A message which stores information from irc v3 tags

__init__(*source*, *target*, *text*, *tags=None*)
Initialize a new message from source to target with the given text

Parameters

- **source** (*Chatter*) – The source chatter
- **target** (*str*) – the target
- **text** (str) – the content of the message
- **tags** (list of *Tag*) – the irc v3 tags

Raises None

Methods

__init__(source, target, text[, tags])	Initialize a new message from source to target with the given text
from_event(event)	Create a message from an event
set_tags(tags)	For every known tag, set the appropriate attribute.

Attributes

emotes	Return the emotes
subscriber	Return whether the message was sent from a subscriber
turbo	Return whether the message was sent from a turbo user

color = None
the hex representation of the user color

user_type = None
Turbo type. None for regular ones. Other user types are mod, global_mod, staff, admin.

classmethod from_event(*event*)
Create a message from an event

Parameters **event** (`Event3`) – the event that was received of type `pubmsg` or `privmsg`

Returns a message that resembles the event

Return type *Message3*

Raises None

set_tags(*tags*)
For every known tag, set the appropriate attribute.

Known tags are:

color The user color

emotes A list of emotes

subscriber True, if subscriber

turbo True, if turbo user

user_type None, mod, staff, global_mod, admin

Parameters **tags** (`list` of *Tag* | None) – a list of tags

Returns None

Return type None

Raises None

emotes
Return the emotes

Returns the emotes

Return type `list`

Raises None

subscriber
Return whether the message was sent from a subscriber

Returns True, if subscriber

Return type bool

Raises None

turbo
Return whether the message was sent from a turbo user

Returns True, if turbo

Return type bool

Raises None

pytwitcherapi.chat.message.Tag

class pytwitcherapi.chat.message.**Tag**(*name*, *value=None*, *vendor=None*)
Bases: object

An irc v3 tag

Specification for tags. A tag will associate metadata with a message.

To get tags in twitch chat, you have to specify it in the **'capability negotiation<http://ircv3.net/specs/core/capability-negotiation-3.1.html>'_**.

__init__(*name*, *value=None*, *vendor=None*)
Initialize a new tag called name

Parameters

- **name** (str) – The name of the tag
- **value** (str | None) – The value of a tag
- **vendor** (str | tag) – the vendor for vendor specific tags

Raises None

Methods

__init__(name[, value, vendor])	Initialize a new tag called name
from_str(tagstring)	Create a tag by parsing the tag of a message

classmethod from_str(*tagstring*)
Create a tag by parsing the tag of a message

Parameters **tagstring** (str) – A tag string described in the irc protocol

Returns A tag

Return type *Tag*

Raises None

Module contents

Package for interacting with the IRC chat of a channel.

The main client for connecting to the channel is *IRCClient*.

Classes

IRCClient(session, channel[, queuesize])	Simple IRC client which can connect to a single pytwitcherapi.Channel.

pytwitcherapi.chat.IRCClient

class pytwitcherapi.chat.**IRCClient**(*session*, *channel*, *queuesize=100*)

Bases: irc.client.SimpleIRCClient

Simple IRC client which can connect to a single pytwitcherapi.Channel.

You need an authenticated session with scope chat_login. Call *IRCClient.process_forever()* to start the event loop. This will block the current thread though. Calling *IRCClient.shutdown()* will stop the loop.

There are a lot of methods that can make the client send commands while the client is in its event loop. These methods are wrapped ones of irc.client.ServerConnection. They will always use *IRCClient.out_connection*!

You can implement handlers for all sorts of events by subclassing and creating a method called on_<event.type>. Note that *IRCClient.out_connection* will only get to the *IRCClient.on_welcome()* event (and then join a channel) and the IRCClient.on_join() event. For all other events, the *IRCClient.in_connection* will handle it and the other one will ignore it. This behaviour is implemented in IRCClient._dispatcher()

Little example with threads. Change input to raw_input for python 2:

```
import threading

from pytwitcherapi import chat

session = ...  # we assume an authenticated TwitchSession
channel = session.get_channel('somechannel')
client = chat.IRCClient(session, channel)
t = threading.Thread(target=client.process_forever,
                     kwargs={'timeout': 0.2})
t.start()

try:
    while True:
        m = input('Send Message:')
        if not m: break;
        # will be processed in other thread
        client.send_msg(m)
finally:
    client.shutdown()
    t.join()
```

__init__(*session*, *channel*, *queuesize=100*)
Initialize a new irc client which can connect to the given channel.

Parameters

- **session** (pytwitcherapi.TwitchSession) – a authenticated session. Used for quering the right server and the login username.
- **channel** (pytwitcherapi.Channel) – a channel
- **queuesize** (int) – The queuesize for storing messages in *IRCClient.messages*. If 0, unlimited size.

Raises exceptions.NotAuthorizedError

Methods

__init__(session, channel[, queuesize])	Initialize a new irc client which can connect to the given channel.
action(*args, **kwargs)	Send a CTCP ACTION command.
admin(*args, **kwargs)	Send an ADMIN command.
cap(*args, **kwargs)	Send a CAP command according to the spec.
connect(*args, **kwargs)	Connect using the underlying connection
ctcp(*args, **kwargs)	Send a CTCP command.
ctcp_reply(*args, **kwargs)	Send a CTCP REPLY command.
dcc_connect(address, port[, dcctype])	Connect to a DCC peer.
dcc_listen([dcctype])	Listen for connections from a DCC peer.
globops(*args, **kwargs)	Send a GLOBOPS command.
info(*args, **kwargs)	Send an INFO command.
invite(*args, **kwargs)	Send an INVITE command.
ison(*args, **kwargs)	Send an ISON command.
join(*args, **kwargs)	Send a JOIN command.
kick(*args, **kwargs)	Send a KICK command.
links(*args, **kwargs)	Send a LINKS command.
list(*args, **kwargs)	Send a LIST command.
lusers(*args, **kwargs)	Send a LUSERS command.
mode(*args, **kwargs)	Send a MODE command.
motd(*args, **kwargs)	Send an MOTD command.
names(*args, **kwargs)	Send a NAMES command.
negotiate_capabilities(connection)	Send *IRCClient.capabilities* to the server.
nick(*args, **kwargs)	Send a NICK command.
notice(*args, **kwargs)	Send a NOTICE command.
on_privmsg(connection, event)	Handle the private message event
on_pubmsg(connection, event)	Handle the public message event
on_welcome(connection, event)	Handle the welcome event
oper(*args, **kwargs)	Send an OPER command.
part(*args, **kwargs)	Send a PART command.
*pass_*(*args, **kwargs)	Send a PASS command.
ping(*args, **kwargs)	Send a PING command.
pong(*args, **kwargs)	Send a PONG command.
privmsg(*args, **kwargs)	Send a PRIVMSG command.
privmsg_many(*args, **kwargs)	Send a PRIVMSG command to multiple targets.
quit(*args, **kwargs)	Send a QUIT command.

Continued on next page

Table 2.24 – continued from previous page

send_msg(message)	Send the given message to the channel
send_raw(*args, **kwargs)	Send raw string to the server.
squit(*args, **kwargs)	Send an SQUIT command.
start()	Start the IRC client.
stats(*args, **kwargs)	Send a STATS command.
store_message(connection, event)	Store the message of event in *IRCClient.messages*.
time(*args, **kwargs)	Send a TIME command.
topic(*args, **kwargs)	Send a TOPIC command.
trace(*args, **kwargs)	Send a TRACE command.
user(*args, **kwargs)	Send a USER command.
userhost(*args, **kwargs)	Send a USERHOST command.
users(*args, **kwargs)	Send a USERS command.
version(*args, **kwargs)	Send a VERSION command.
wallops(*args, **kwargs)	Send a WALLOPS command.
who(*args, **kwargs)	Send a WHO command.
whois(*args, **kwargs)	Send a WHOIS command.
whowas(*args, **kwargs)	Send a WHOWAS command.

Attributes

capabilities	List of irc capabilities
channel	The channel to connect to.

reactor_class
The reactor class which dispatches events

alias of Reactor3

capabilities = [':twitch.tv/membership', ':twitch.tv/commands', ':twitch.tv/tags']
List of irc capabilities

in_connection = None
Connection that receives messages

out_connection = None
Connection that sends messages

session = None
an authenticated session. Used for quering the right server and the login username.

login_user = None
The user that is used for logging in to the chat

shutdown = None
Call this method for shutting down the client. This is thread safe.

process_forever = None
Call this method to process messages until shutdown() is called.

Parameters **timeout** (float) – timeout for waiting on data in seconds

messages = None
A queue which stores all private and public *pytwitcherapi.chat.message.Message3*. Usefull for accessing messages from another thread.

channel
The channel to connect to. When setting the channel, automatically connect to it. If channel is None, disconnect.

on_welcome(*connection*, *event*)
Handle the welcome event

Automatically join the channel.

Parameters

- **connection** (irc.client.ServerConnection) – the connection with the event
- **event** (irc.client.Event) – the event to handle

Returns None

negotiate_capabilities(*connection*)
Send *IRCClient.capabilities* to the server.

Parameters **connection** (irc.client.ServerConnection) – the connection to use for sending

Returns None

Return type None

Raises None

store_message(*connection*, *event*)
Store the message of event in *IRCClient.messages*.

Parameters

- **connection** (irc.client.ServerConnection) – the connection with the event
- **event** (irc.client.Event) – the event to handle

Returns None

on_pubmsg(*connection*, *event*)
Handle the public message event

This stores the message in *IRCClient.messages* via *IRCClient.store_message()*.

Parameters

- **connection** (irc.client.ServerConnection) – the connection with the event
- **event** (irc.client.Event) – the event to handle

Returns None

on_privmsg(*connection*, *event*)
Handle the private message event

This stores the message in *IRCClient.messages* via *IRCClient.store_message()*.

Parameters

- **connection** (irc.client.ServerConnection) – the connection with the event
- **event** (irc.client.Event) – the event to handle

Returns None

send_msg(*message*)
Send the given message to the channel

This is a convenience method for *IRCClient.privmsg()*, which uses the current channel as target. This method is thread safe and can be called from another thread even if the client is running in *IRCClient.process_forever()*.

Parameters **message** (str) – The message to send

Returns None

Return type None

Raises None

action(**args*, ***kwargs*)
Send a CTCP ACTION command.

admin(**args*, ***kwargs*)
Send an ADMIN command.

cap(**args*, ***kwargs*)
Send a CAP command according to the spec.

Arguments:

subcommand – LS, LIST, REQ, ACK, CLEAR, END args – capabilities, if required for given subcommand

Example:

.cap('LS') .cap('REQ', 'multi-prefix', 'sasl') .cap('END')

ctcp(**args*, ***kwargs*)
Send a CTCP command.

ctcp_reply(**args*, ***kwargs*)
Send a CTCP REPLY command.

globops(**args*, ***kwargs*)
Send a GLOBOPS command.

info(**args*, ***kwargs*)
Send an INFO command.

invite(**args*, ***kwargs*)
Send an INVITE command.

ison(**args*, ***kwargs*)
Send an ISON command.

Arguments:

nicks – List of nicks.

join(**args*, ***kwargs*)
Send a JOIN command.

kick(**args*, ***kwargs*)
Send a KICK command.

links(**args*, ***kwargs*)
Send a LINKS command.

list(**args*, ***kwargs*)
Send a LIST command.

lusers(**args*, ***kwargs*)
Send a LUSERS command.

mode(**args*, ***kwargs*)
Send a MODE command.

motd(**args*, ***kwargs*)
Send an MOTD command.

names(**args*, ***kwargs*)
Send a NAMES command.

nick(**args*, ***kwargs*)
Send a NICK command.

notice(**args*, ***kwargs*)
Send a NOTICE command.

oper(**args*, ***kwargs*)
Send an OPER command.

part(**args*, ***kwargs*)
Send a PART command.

pass_(**args*, ***kwargs*)
Send a PASS command.

ping(**args*, ***kwargs*)
Send a PING command.

pong(**args*, ***kwargs*)
Send a PONG command.

privmsg(**args*, ***kwargs*)
Send a PRIVMSG command.

privmsg_many(**args*, ***kwargs*)
Send a PRIVMSG command to multiple targets.

quit(**args*, ***kwargs*)
Send a QUIT command.

send_raw(**args*, ***kwargs*)
Send raw string to the server.

The string will be padded with appropriate CR LF.

squit(**args*, ***kwargs*)
Send an SQUIT command.

stats(**args*, ***kwargs*)
Send a STATS command.

time(**args*, ***kwargs*)
Send a TIME command.

topic(**args*, ***kwargs*)
Send a TOPIC command.

trace(**args*, ***kwargs*)
Send a TRACE command.

user(**args*, ***kwargs*)
Send a USER command.

userhost (**args*, ***kwargs*)
Send a USERHOST command.

users (**args*, ***kwargs*)
Send a USERS command.

version (**args*, ***kwargs*)
Send a VERSION command.

wallops (**args*, ***kwargs*)
Send a WALLOPS command.

who (**args*, ***kwargs*)
Send a WHO command.

whois (**args*, ***kwargs*)
Send a WHOIS command.

whowas (**args*, ***kwargs*)
Send a WHOWAS command.

Data

absolute_import

pytwitcherapi.chat.**absolute_import = _Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)**

Submodules

pytwitcherapi.constants

Collection of constants

These constants might be needed in multiple modules, so we pull them together here.

Data

LOGIN_SERVER_ADRESS	Server adress of server that catches the redirection and the oauth token.
REDIRECT_URI	The redirect url of pytwitcher.

pytwitcherapi.constants.**LOGIN_SERVER_ADRESS = ('', 42420)**
Server adress of server that catches the redirection and the oauth token.

pytwitcherapi.constants.**REDIRECT_URI = 'http://localhost:42420'**
The redirect url of pytwitcher. We do not need to redirect anywhere so localhost is set in the twitch prefrences of pytwitcher

pytwitcherapi.exceptions

Collection exceptions

Exceptions

NotAuthorizedError	Exception that is raised, when the session is not authorized.
PytwitcherException	Base exception for pytwitcher

pytwitcherapi.exceptions.NotAuthorizedError

exception pytwitcherapi.exceptions.**NotAuthorizedError**
Bases: *pytwitcherapi.exceptions.PytwitcherException*

Exception that is raised, when the session is not authorized. The user has to login first

pytwitcherapi.exceptions.PytwitcherException

exception pytwitcherapi.exceptions.**PytwitcherException**
Bases: exceptions.Exception

Base exception for pytwitcher

pytwitcherapi.models

Contains classes that wrap the jsons returned by the twitch.tv API

Classes

Channel(name, status, displayname, game, ...)	Channel on twitch.tv
Game(name, box, logo, twitchid[, viewers, ...])	Game on twitch.tv
Stream(game, channel, twitchid, viewers, preview)	A stream on twitch.tv
User(usertype, name, logo, twitchid, ...)	A user on twitch.tv

pytwitcherapi.models.Channel

class pytwitcherapi.models.**Channel**(*name*, *status*, *displayname*, *game*, *twitchid*, *views*, *followers*, *url*, *language*, *broadcaster_language*, *mature*, *logo*, *banner*, *video_banner*, *delay*)
Bases: object

Channel on twitch.tv

__init__(*name*, *status*, *displayname*, *game*, *twitchid*, *views*, *followers*, *url*, *language*, *broadcaster_language*, *mature*, *logo*, *banner*, *video_banner*, *delay*)
Initialize a new channel

Parameters

- **name** (str) – The name of the channel
- **status** (str) – The status
- **displayname** (str) – The name displayed by the interface
- **game** (str) – the game of the channel

- **twitchid** (int) – the internal twitch id
- **views** (int) – the overall views
- **followers** (int) – the follower count
- **url** (str) – the url to the channel
- **language** (str) – the language of the channel
- **broadcaster_language** (str) – the language of the broadcaster
- **mature** (bool) – If true, the channel is only for mature audiences
- **logo** (str) – the link to the logos
- **banner** (str) – the link to the banner
- **video_banner** (str) – the link to the video banner
- **delay** (int) – stream delay

Raises None

Methods

__init__(name, status, displayname, game, ...)	Initialize a new channel
wrap_get_channel(response)	Wrap the response from getting a channel into an instance
wrap_json(json)	Create a Channel instance for the given json
wrap_search(response)	Wrap the response from a channel search into instances

classmethod wrap_search (*response*)
Wrap the response from a channel search into instances and return them

Parameters **response** (requests.Response) – The response from searching a channel

Returns the new channel instances

Return type list of channel

Raises None

classmethod wrap_get_channel (*response*)
Wrap the response from getting a channel into an instance and return it

Parameters **response** (requests.Response) – The response from getting a channel

Returns the new channel instance

Return type list of channel

Raises None

classmethod wrap_json (*json*)
Create a Channel instance for the given json

Parameters **json** (dict) – the dict with the information of the channel

Returns the new channel instance

Return type *Channel*

Raises None

name = None
The name of the channel

status = None
The current status message

displayname = None
The name displayed by the interface

game = None
The game of the channel

twitchid = None
THe internal twitch id

views = None
The overall views

followers = None
The follower count

url = None
the link to the channel page

language = None
Language of the channel

broadcaster_language = None
Language of the broadcaster

mature = None
If true, the channel is only for mature audiences

logo = None
the link to the logo

banner = None
the link to the banner

video_banner = None
the link to the video banner

delay = None
stream delay

pytwitcherapi.models.Game

class pytwitcherapi.models.**Game**(*name*, *box*, *logo*, *twitchid*, *viewers=None*, *channels=None*)
Bases: object

Game on twitch.tv

__init__(*name*, *box*, *logo*, *twitchid*, *viewers=None*, *channels=None*)
Initialize a new game

Parameters

- **name** (str) – The name of the game
- **box** (dict) – Links for the box logos
- **logo** (dict) – Links for the game logo

- **twitchid** (int) – The id used by twitch
- **viewers** (int) – The current amount of viewers
- **channels** (int) – The current amount of channels

Raises None

Methods

__init__(name, box, logo, twitchid[, ...])	Initialize a new game
wrap_json(json[, viewers, channels])	Create a Game instance for the given json
wrap_search(response)	Wrap the response from a game search into instances
wrap_topgames(response)	Wrap the response from quering the top games into instances

classmethod wrap_search (*response*)
Wrap the response from a game search into instances and return them

Parameters **response** (requests.Response) – The response from searching a game

Returns the new game instances

Return type list of *Game*

Raises None

classmethod wrap_topgames (*response*)
Wrap the response from quering the top games into instances and return them

Parameters **response** (requests.Response) – The response for quering the top games

Returns the new game instances

Return type list of *Game*

Raises None

classmethod wrap_json (*json*, *viewers=None*, *channels=None*)
Create a Game instance for the given json

Parameters

- **json** (dict) – the dict with the information of the game
- **viewers** (int) – The viewer count
- **channels** (int) – The viewer count

Returns the new game instance

Return type *Game*

Raises None

name = None
The name of the game

box = None
Links for the box logos

logo = None
Links for the logos

twitchid = None
Id used by twitch

viewers = None
Current amount of viewers

channels = None
Current amount of channels

pytwitcherapi.models.Stream

class pytwitcherapi.models.**Stream**(*game*, *channel*, *twitchid*, *viewers*, *preview*)
Bases: object

A stream on twitch.tv

__init__(*game*, *channel*, *twitchid*, *viewers*, *preview*)
Initialize a new stream

Parameters

- **game** (str) – name of the game
- **channel** (*Channel*) – the channel that is streaming
- **twitchid** (int) – the internal twitch id
- **viewers** (int) – the viewer count
- **preview** (dict) – a dict with preview picture links of the stream

Raises None

Methods

__init__(game, channel, twitchid, viewers, ...)	Initialize a new stream
wrap_get_stream(response)	Wrap the response from getting a stream into an instance
wrap_json(json)	Create a Stream instance for the given json
wrap_search(response)	Wrap the response from a stream search into instances

classmethod wrap_search(*response*)
Wrap the response from a stream search into instances and return them

Parameters **response** (requests.Response) – The response from searching a stream

Returns the new stream instances

Return type list of stream

Raises None

classmethod wrap_get_stream(*response*)
Wrap the response from getting a stream into an instance and return it

Parameters **response** (requests.Response) – The response from getting a stream

Returns the new stream instance

Return type list of stream

Raises None

classmethod wrap_json(*json*)
Create a Stream instance for the given json

Parameters **json** (dict | None) – the dict with the information of the stream

Returns the new stream instance

Return type *Stream* | None

Raises None

game = None
Name of the game that is beeing streamed

channel = None
The channel instance

twitchid = None
The internal twitch id

viewers = None
the viewer count

preview = None
A dict with preview picture links of the stream

pytwitcherapi.models.User

class pytwitcherapi.models.**User**(*usertype*, *name*, *logo*, *twitchid*, *displayname*, *bio*)
Bases: object

A user on twitch.tv

__init__(*usertype*, *name*, *logo*, *twitchid*, *displayname*, *bio*)
Initialize a new user

Parameters

- **usertype** (str) – the user type on twitch, e.g. "user"
- **name** (str) – the username
- **logo** (str) – the link to the logo
- **twitchid** (int) – the internal twitch id
- **displayname** (str) – the name diplayed by the interface
- **bio** (str) – the user bio

Raises None

Methods

__init__(usertype, name, logo, twitchid, ...)	Initialize a new user
wrap_get_user(response)	Wrap the response from getting a user into an instance
wrap_json(json)	Create a User instance for the given json

classmethod wrap_get_user(*response*)
Wrap the response from getting a user into an instance and return it

Parameters **response** (requests.Response) – The response from getting a user

Returns the new user instance

Return type list of *User*

Raises None

classmethod wrap_json(*json*)
Create a User instance for the given json

Parameters **json** (dict | None) – the dict with the information of the user

Returns the new user instance

Return type *User*

Raises None

usertype = None
the user type on twitch, e.g. "user"

name = None
the username

logo = None
link to the logo

twitchid = None
internal twitch id

displayname = None
name displayed by the interface

bio = None
the user bio

pytwitcherapi.oauth

Twitch.tv uses OAuth2 for authorization. We use the Implicit Grant Workflow. The user has to visit an authorization site, login, authorize PyTwitcher. Once he allows PyTwitcher, twitch will redirect him to pytwitcherapi.REDIRECT_URI. In the url fragment, there is the access token.

This module features a server, that will respond to the redirection of the user. So if twitch is redirecting to pytwitcherapi.REDIRECT_URI, the server is gonna send a website, which will extract the access token, send it as a post request and give the user a response, that everything worked.

Classes

LoginServer(session)	This server responds to the redirection of the user after he granted authorization.
RedirectHandler(request, client_address, server)	This request handler will handle the redirection of the user when he grants authorization to PyTwitcher and twitch redirects him.

Continued on next page

Table 2.34 – continued from previous page

TwitchOAuthClient(client_id[, ...])	This is a client needed for oauthlib.oauth2.OAuth2Session.

pytwitcherapi.oauth.LoginServer

class pytwitcherapi.oauth.**LoginServer**(*session*)

Bases: BaseHTTPServer.HTTPServer

This server responds to the redirection of the user after he granted authorization.

__init__(*session*)

Initialize a new server.

The server will be on constants.LOGIN_SERVER_ADRESS.

Parameters **session** (requests_oauthlib.OAuth2Session) – the session that needs a token

Raises None

Methods

__init__(session)	Initialize a new server.
close_request(request)	Called to clean up an individual request.
fileno()	Return socket file number.
finish_request(request, client_address)	Finish one request by instantiating RequestHandlerClass.
get_request()	Get the request and client address from the socket.
handle_error(request, client_address)	Handle an error gracefully.
handle_request()	Handle one request, possibly blocking.
handle_timeout()	Called if no new request arrives within self.timeout.
process_request(request, client_address)	Call finish_request.
serve_forever([poll_interval])	Handle one request at a time until shutdown.
server_activate()	Called by constructor to activate the server.
server_bind()	Override server_bind to store the server name.
server_close()	Called to clean-up the server.
set_token(redirecturl)	Set the token on the session
shutdown()	Stops the serve_forever loop.
shutdown_request(request)	Called to shutdown and close an individual request.
verify_request(request, client_address)	Verify the request.

Attributes

address_family
allow_reuse_address
request_queue_size
socket_type
timeout

session = None
The session that needs a token

set_token (*redirecturl*)
Set the token on the session

Parameters **redirecturl** (str) – the original full redirect url

Returns None

Return type None

Raises None

pytwitcherapi.oauth.RedirectHandler

class pytwitcherapi.oauth.**RedirectHandler** (*request*, *client_address*, *server*)
Bases: BaseHTTPServer.BaseHTTPRequestHandler

This request handler will handle the redirection of the user when he grants authorization to PyTwitcher and twitch redirects him.

__init__ (*request*, *client_address*, *server*)

Methods

__init__(request, client_address, server)	
address_string()	Return the client address formatted for logging.
date_time_string([timestamp])	Return the current date and time formatted for a message header.
do_GET()	Handle GET requests
do_POST()	Handle POST requests
end_headers()	Send the blank line ending the MIME headers.
finish()	
handle()	Handle multiple requests if necessary.
handle_one_request()	Handle a single HTTP request.
log_date_time_string()	Return the current time formatted for logging.
log_error(format, *args)	Log an error.
log_message(format, *args)	Log an arbitrary message.
log_request([code, size])	Log an accepted request.
parse_request()	Parse a request (internal).
send_error(code[, message])	Send and log an error reply.
send_header(keyword, value)	Send a MIME header.
send_response(code[, message])	Send the response header and log the response code.
setup()	
version_string()	Return the server software version string.

Attributes

default_request_version
disable_nagle_algorithm
error_content_type

Continued on next page

Table 2.38 – continued from previous page

error_message_format
extract_site_url
monthname
protocol_version
rbufsize
responses
server_version
success_site_url
sys_version
timeout
wbufsize
weekdayname

extract_site_url = '/'

success_site_url = '/success'

do_GET()
Handle GET requests

If the path is '/', a site which extracts the token will be generated. This will redirect the user to the '/sucess' page, which shows a success message.

Returns None

Return type None

Raises None

do_POST()
Handle POST requests

When the user is redirected, this handler will respond with a website which will send a post request with the url fragment as parameters. This will get the parameters and store the original redirection url and fragments in `LoginServer.tokenurl`.

Returns None

Return type None

Raises None

pytwitcherapi.oauth.TwitchOAuthClient

class pytwitcherapi.oauth.**TwitchOAuthClient**(*client_id*, *default_token_placement=u'auth_header'*, *token_type=u'Bearer'*, *access_token=None*, *refresh_token=None*, *mac_key=None*, *mac_algorithm=None*, *token=None*, *scope=None*, *state=None*, *redirect_url=None*, *state_generator=<function generate_token>*, ***kwargs*)

Bases: oauthlib.oauth2.rfc6749.clients.mobile_application.MobileApplicationClient

This is a client needed for `oauthlib.oauth2.OAuth2Session`. It fixes the Authorization header for twitch.

Usually the Authorization Header looks like this:

```
{'Authorization': 'Bearer <<token>>'}
```

But Twitch needs it to be like this:

```
{'Authorization': 'OAuth <<token>>'}
```

So we override `TwitchOAuthClient._add_bearer_token()` to fix the header.

__init__(*client_id*, *default_token_placement=u'auth_header'*, *token_type=u'Bearer'*, *access_token=None*, *refresh_token=None*, *mac_key=None*, *mac_algorithm=None*, *token=None*, *scope=None*, *state=None*, *redirect_url=None*, *state_generator=<function generate_token>*, ***kwargs*)

Initialize a client with commonly used attributes.

Parameters **client_id** – Client identifier given by the OAuth provider upon registration.

Parameters **default_token_placement** – Tokens can be supplied in the Authorization header (default), the URL query component (`query`) or the request body (`body`).

Parameters **token_type** – OAuth 2 token type. Defaults to Bearer. Change this if you specify the `access_token` parameter and know it is of a different token type, such as a MAC, JWT or SAML token. Can also be supplied as `token_type` inside the `token` dict parameter.

Parameters **access_token** – An access token (string) used to authenticate requests to protected resources. Can also be supplied inside the `token` dict parameter.

Parameters **refresh_token** – A refresh token (string) used to refresh expired tokens. Can also be supplied inside the `token` dict parameter.

Parameters

- **mac_key** – Encryption key used with MAC tokens.
- **mac_algorithm** – Hashing algorithm for MAC tokens.
- **token** – A dict of token attributes such as `access_token`, `token_type` and `expires_at`.

Parameters

- **scope** – A list of default scopes to request authorization for.
- **state** – A CSRF protection string used during authorization.
- **redirect_url** – The redirection endpoint on the client side to which the user returns after authorization.

Parameters **state_generator** – A no argument state generation callable. Defaults to `oauthlib.common.generate_token()`.

Methods

__init__(client_id[, ...])	Initialize a client with commonly used attributes.

Continued on next page

Table 2.39 – continued from previous page

add_token(uri[, http_method, body, headers, ...])	Add token to the request uri, body or authorization header.
parse_request_body_response(body[, scope])	Parse the JSON response body.
parse_request_uri_response(uri[, state, scope])	Parse the response URI fragment.
prepare_authorization_request(authorization_url)	Prepare the authorization request.
prepare_refresh_body([body, refresh_token, ...])	Prepare an access token request, using a refresh token.
prepare_refresh_token_request(token_url[, ...])	Prepare an access token refresh request.
prepare_request_body(*args, **kwargs)	Abstract method used to create request bodies.
prepare_request_uri(uri[, redirect_uri, ...])	Prepare the implicit grant request URI.
prepare_token_request(token_url[, ...])	Prepare a token creation request.
prepare_token_revocation_request(...[, ...])	Prepare a token revocation request.

Attributes

token_types	Supported token types and their respective methods

Data

log	Instances of the Logger class represent a single logging channel.

pytwitcherapi.oauth.**log = <logging.Logger object>**

Instances of the Logger class represent a single logging channel. A "logging channel" indicates an area of an application. Exactly how an "area" is defined is up to the application developer. Since an application can have any number of areas, logging channels are identified by a unique string. Application areas can be nested (e.g. an area of "input processing" might include sub-areas "read CSV files", "read XLS files" and "read Gnumeric files"). To cater for this natural nesting, channel names are organized into a namespace hierarchy where levels are separated by periods, much like the Java or Python package namespace. So in the instance given above, channel names might be "input" for the upper level, and "input.csv", "input.xls" and "input.gnu" for the sub-levels. There is no arbitrary limit to the depth of nesting.

pytwitcherapi.session

API for communicating with twitch

Classes

OAuthSession()	Session with oauth2 support.
TwitchSession()	Session for making requests to the twitch api

pytwitcherapi.session.OAuthSession

class pytwitcherapi.session.**OAuthSession**

Bases: requests_oauthlib.oauth2_session.OAuth2Session

Session with oauth2 support.

You can still use http requests.

__init__()

Initialize a new oauth session

Raises None

Methods

__init__()	Initialize a new oauth session
authorization_url(url[, state])	Form an authorization URL.
close()	Closes all adapters and as such the session
delete(url, **kwargs)	Sends a DELETE request.
fetch_token(token_url[, code, ...])	Generic method for fetching an access token from the token endpoint.
get(url, **kwargs)	Sends a GET request.
get_adapter(url)	Returns the appropriate connection adapter for the given URL.
get_auth_url()	Return the url for the user to authorize PyTwitcher
get_redirect_target(resp)	Receives a Response.
head(url, **kwargs)	Sends a HEAD request.
merge_environment_settings(url, proxies, ...)	Check the environment and merge it with some settings.
mount(prefix, adapter)	Registers a connection adapter to a prefix.
new_state()	Generates a state string to be used in authorizations.
options(url, **kwargs)	Sends a OPTIONS request.
patch(url[, data])	Sends a PATCH request.
post(url[, data, json])	Sends a POST request.
prepare_request(request)	Constructs a PreparedRequest for transmission and returns it.
put(url[, data])	Sends a PUT request.
rebuild_auth(prepared_request, response)	When being redirected we may want to strip authentication from the request to avoid leaking credentials.
rebuild_method(prepared_request, response)	When being redirected we may want to change the method of the request based on certain specs or browser behavior.
rebuild_proxies(prepared_request, proxies)	This method re-evaluates the proxy configuration by considering the environment variables.
refresh_token(token_url[, refresh_token, ...])	Fetch a new access token using a refresh token.
register_compliance_hook(hook_type, hook)	Register a hook for request/response tweaking.
request(method, url, **kwargs)	Constructs a requests.Request, prepares it and sends it.
resolve_redirects(resp, req[, stream, ...])	Receives a Response.
send(request, **kwargs)	Send a given PreparedRequest.
shutdown_login_server()	Shutdown the login server and thread

Continued on next page

Table 2.43 – continued from previous page

start_login_server()	Start a server that will get a request from a user logging in.
token_from_fragment(authorization_response)	Parse token from the URI fragment, used by MobileApplicationClients.

Attributes

access_token	
authorized	Boolean that indicates whether this session has an OAuth token or not.
client_id	
token	

login_server = None
The server that handles the login redirect

login_thread = None
The thread that serves the login server

request (*method*, *url*, ***kwargs*)
Constructs a requests.Request, prepares it and sends it. Raises HTTPErrors by default.

Parameters

- **method** (str) – method for the new Request object.
- **url** (str) – URL for the new Request object.
- **kwargs** – keyword arguments of requests.Session.request()

Returns a resonse object

Return type requests.Response

Raises requests.HTTPError

start_login_server()
Start a server that will get a request from a user logging in.

This uses the Implicit Grant Flow of OAuth2. The user is asked to login to twitch and grant PyTwitcher authorization. Once the user agrees, he is redirected to an url. This server will respond to that url and get the oauth token.

The server serves in another thread. To shut him down, call TwitchSession.shutdown_login_server().

This sets the TwitchSession.login_server, TwitchSession.login_thread variables.

Returns The created server

Return type BaseHTTPServer.HTTPServer

Raises None

shutdown_login_server()
Shutdown the login server and thread

Returns None

Return type None

Raises None

get_auth_url()
Return the url for the user to authorize PyTwitcher

Returns The url the user should visit to authorize PyTwitcher

Return type str

Raises None

pytwitcherapi.session.TwitchSession

class pytwitcherapi.session.**TwitchSession**
Bases: *pytwitcherapi.session.OAuthSession*

Session for making requests to the twitch api

Use *TwitchSession.kraken_request()*, *TwitchSession.usher_request()*, *TwitchSession.oldapi_request()* to make easier calls to the api directly.

To get authorization, the user has to grant PyTwitcher access. The workflow goes like this:

1. Start the login server with `TwitchSession.start_login_server()`.
2. User should visit `TwitchSession.get_auth_url()` in his browser and follow insturctions (e.g Login and Allow PyTwitcher).
3. Check if the session is authorized with `TwitchSession.authorized()`.
4. Shut the login server down with `TwitchSession.shutdown_login_server()`.

Now you can use methods that need authorization.

__init__()
Initialize a new TwitchSession

Raises None

Methods

__init__()	Initialize a new TwitchSession
authorization_url(url[, state])	Form an authorization URL.
close()	Closes all adapters and as such the session
delete(url, **kwargs)	Sends a DELETE request.
fetch_token(token_url[, code, ...])	Generic method for fetching an access token from the token endpoint.
fetch_viewers(game)	Query the viewers and channels of the given game and
followed_streams(*args, **kwargs)	Return the streams the current user follows.
get(url, **kwargs)	Sends a GET request.
get_adapter(url)	Returns the appropriate connection adapter for the given URL.
get_auth_url()	Return the url for the user to authorize PyTwitcher
get_channel(name)	Return the channel for the given name
get_channel_access_token(channel)	Return the token and sig for the given channel
get_chat_server(channel)	Get an appropriate chat server for the given channel
get_emote_picture(emote[, size])	Return the picture for the given emote

Continued on next page

Table 2.45 – continued from previous page

get_game(name)	Get the game instance for a game name
get_playlist(channel)	Return the playlist for the given channel
get_quality_options(channel)	Get the available quality options for streams of the given channel
get_redirect_target(resp)	Receives a Response.
get_stream(channel)	Return the stream of the given channel
get_streams([game, channels, limit, offset])	Return a list of streams queried by a number of parameters
get_user(name)	Get the user for the given name
head(url, **kwargs)	Sends a HEAD request.
kraken_request(method, endpoint, **kwargs)	Make a request to one of the kraken api endpoints.
merge_environment_settings(url, proxies, ...)	Check the environment and merge it with some settings.
mount(prefix, adapter)	Registers a connection adapter to a prefix.
new_state()	Generates a state string to be used in authorizations.
oldapi_request(method, endpoint, **kwargs)	Make a request to one of the old api endpoints.
options(url, **kwargs)	Sends a OPTIONS request.
patch(url[, data])	Sends a PATCH request.
post(url[, data, json])	Sends a POST request.
prepare_request(request)	Constructs a PreparedRequest for transmission and returns it.
put(url[, data])	Sends a PUT request.
query_login_user(*args, **kwargs)	Query and return the currently logined user
rebuild_auth(prepared_request, response)	When being redirected we may want to strip authentication from the request to avoid leaking credentials.
rebuild_method(prepared_request, response)	When being redirected we may want to change the method of the request based on certain specs or browser behavior.
rebuild_proxies(prepared_request, proxies)	This method re-evaluates the proxy configuration by considering the environment variables.
refresh_token(token_url[, refresh_token, ...])	Fetch a new access token using a refresh token.
register_compliance_hook(hook_type, hook)	Register a hook for request/response tweaking.
request(method, url, **kwargs)	Constructs a requests.Request, prepares it and sends it.
resolve_redirects(resp, req[, stream, ...])	Receives a Response.
search_channels(query[, limit, offset])	Search for channels and return them
search_games(query[, live])	Search for games that are similar to the query
search_streams(query[, hls, limit, offset])	Search for streams and return them
send(request, **kwargs)	Send a given PreparedRequest.
shutdown_login_server()	Shutdown the login server and thread
start_login_server()	Start a server that will get a request from a user logging in.
token_from_fragment(authorization_response)	Parse token from the URI fragment, used by MobileApplicationClients.
top_games([limit, offset])	Return the current top games
usher_request(method, endpoint, **kwargs)	Make a request to one of the usher api endpoints.

Attributes

access_token

Continued on next page

Table 2.46 – continued from previous page

authorized	Boolean that indicates whether this session has an OAuth token or not.
client_id	
token	Return the oauth token

baseurl = None
The baseurl that gets prepended to every request url

current_user = None
The currently logined user.

token
Return the oauth token

Returns the token

Return type dict

Raises None

kraken_request(*method*, *endpoint*, ***kwargs*)
Make a request to one of the kraken api endpoints.

Headers are automatically set to accept *TWITCH_HEADER_ACCEPT*. Also the client id from *CLIENT_ID* will be set. The url will be constructed of *TWITCH_KRAKENURL* and the given endpoint.

Parameters

- **method** (str) – the request method
- **endpoint** (str) – the endpoint of the kraken api. The base url is automatically provided.
- **kwargs** – keyword arguments of requests.Session.request()

Returns a resonse object

Return type requests.Response

Raises requests.HTTPError

usher_request(*method*, *endpoint*, ***kwargs*)
Make a request to one of the usher api endpoints.

The url will be constructed of *TWITCH_USHERURL* and the given endpoint.

Parameters

- **method** (str) – the request method
- **endpoint** (str) – the endpoint of the usher api. The base url is automatically provided.
- **kwargs** – keyword arguments of requests.Session.request()

Returns a resonse object

Return type requests.Response

Raises requests.HTTPError

oldapi_request(*method*, *endpoint*, ***kwargs*)
Make a request to one of the old api endpoints.

The url will be constructed of *TWITCH_APIURL* and the given endpoint.

Parameters

- **method** (str) – the request method
- **endpoint** (str) – the endpoint of the old api. The base url is automatically provided.
- **kwargs** – keyword arguments of requests.Session.request()

Returns a resonse object

Return type requests.Response

Raises requests.HTTPError

fetch_viewers(*game*)
Query the viewers and channels of the given game and set them on the object

Returns the given game

Return type models.Game

Raises None

search_games(*query*, *live=True*)
Search for games that are similar to the query

Parameters

- **query** (str) – the query string
- **live** (bool) – If true, only returns games that are live on at least one channel

Returns A list of games

Return type list of models.Game instances

Raises None

top_games(*limit=10*, *offset=0*)
Return the current top games

Parameters

- **limit** (int) – the maximum amount of top games to query
- **offset** (int) – the offset in the top games

Returns a list of top games

Return type list of models.Game

Raises None

get_game(*name*)
Get the game instance for a game name

Parameters **name** (str) – the name of the game

Returns the game instance

Return type models.Game | None

Raises None

get_channel(*name*)
Return the channel for the given name

Parameters **name** (str) – the channel name

Returns the model instance

Return type models.Channel

Raises None

search_channels(*query*, *limit=25*, *offset=0*)
Search for channels and return them

Parameters

- **query** (str) – the query string
- **limit** (int) – maximum number of results
- **offset** (int) – offset for pagination

Returns A list of channels

Return type list of models.Channel instances

Raises None

get_stream(*channel*)
Return the stream of the given channel

Parameters **channel** (str | models.Channel) – the channel that is broadcasting. Either name or models.Channel instance

Returns the stream or None, if the channel is offline

Return type models.Stream | None

Raises None

get_streams(*game=None*, *channels=None*, *limit=25*, *offset=0*)
Return a list of streams queried by a number of parameters sorted by number of viewers descending

Parameters

- **game** (str | models.Game) – the game or name of the game
- **channels** (list of models.Channel or str) – list of models.Channels or channel names (can be mixed)
- **limit** (int) – maximum number of results
- **offset** (int) – offset for pagination

Returns A list of streams

Return type list of models.Stream

Raises None

search_streams(*query*, *hls=False*, *limit=25*, *offset=0*)
Search for streams and return them

Parameters

- **query** (str) – the query string
- **hls** (bool) – If true, only return streams that have hls stream
- **limit** (int) – maximum number of results
- **offset** (int) – offset for pagination

Returns A list of streams

Return type list of models.Stream instances

Raises None

followed_streams(**args*, ***kwargs*)
Return the streams the current user follows.

Needs authorization `user_read`.

Parameters

- **limit** (`int`) – maximum number of results
- **offset** (`int`) – offset for pagination

Returns A list of streams

Return type `list`of :class:`models.Stream` instances

Raises `exceptions.NotAuthorizedError`

get_user(*name*)
Get the user for the given name

Parameters **name** (`str`) – The username

Returns the user instance

Return type `models.User`

Raises None

query_login_user(**args*, ***kwargs*)
Query and return the currently logined user

Returns The user instance

Return type `models.User`

Raises `exceptions.NotAuthorizedError`

get_playlist(*channel*)
Return the playlist for the given channel

Parameters **channel** (`models.Channel` | `str`) – the channel

Returns the playlist

Return type `m3u8.M3U8`

Raises `requests.HTTPError` if channel is offline.

get_quality_options(*channel*)
Get the available quality options for streams of the given channel

Possible values in the list:

•source

•high

•medium

•low

•mobile

•audio

Parameters **channel** (`models.Channel` | `str`) – the channel or channel name

Returns list of quality options

Return type list of str

Raises requests.HTTPError if channel is offline.

get_channel_access_token(*channel*)
Return the token and sig for the given channel

Parameters **channel** (channel | str) – the channel or channel name to get the access token for

Returns The token and sig for the given channel

Return type (unicode, unicode)

Raises None

get_chat_server(*channel*)
Get an appropriate chat server for the given channel

Usually the server is irc.twitch.tv. But because of the delicate twitch chat, they use a lot of servers. Big events are on special event servers. This method tries to find a good one.

Parameters **channel** (models.Channel) – the channel with the chat

Returns the server address and port

Return type (str, int)

Raises None

get_emote_picture(*emote*, *size=1.0*)
Return the picture for the given emote

Parameters

- **emote** (*pytwitcherapi.chat.message.Emote*) – the emote object
- **size** (float) – the size of the picture. Choices are: 1.0, 2.0, 3.0

Returns A string resembling the picturedata of the emote

Return type str

Raises None

Functions

needs_auth(meth)	Wraps a method of *TwitchSession* and raises an exceptions.NotAuthorizedError if before calling the method, the session isn't authorized.

Data

AUTHORIZATION_BASE_URL	Authorisation Endpoint
CLIENT_ID	The client id of pytwitcher on twitch.
SCOPES	The scopes that PyTwitcher needs
TWITCH_APIURL	The baseurl for the old twitch api
TWITCH_HEADER_ACCEPT	The header for the Accept key to tell twitch which api version it should use

Continued on next page

Table 2.48 – continued from previous page

TWITCH_KRAKENURL	The baseurl for the twitch api
TWITCH_STATUSURL	str(object='') -> string
TWITCH_USHERURL	The baseurl for the twitch usher api
absolute_import	
log	Instances of the Logger class represent a single logging channel.

pytwitcherapi.session.**needs_auth**(*meth*)
Wraps a method of *TwitchSession* and raises an exceptions.NotAuthorizedError if before calling the method, the session isn't authorized.

Parameters **meth** –

Returns the wrapped method

Return type Method

Raises None

pytwitcherapi.session.**AUTHORIZATION_BASE_URL = 'https://api.twitch.tv/kraken/oauth2/authorize'**
Authorisation Endpoint

pytwitcherapi.session.**CLIENT_ID = '642a2vtmqfumca8hmfcpkosxlkmqifb'**
The client id of pytwitcher on twitch. Use environment variable PYTWITCHER_CLIENT_ID or pytwitcher default value.

pytwitcherapi.session.**SCOPES = ['user_read', 'chat_login']**
The scopes that PyTwitcher needs

pytwitcherapi.session.**TWITCH_APIURL = 'http://api.twitch.tv/api/'**
The baseurl for the old twitch api

pytwitcherapi.session.**TWITCH_HEADER_ACCEPT = 'application/vnd.twitchtv.v3+json'**
The header for the Accept key to tell twitch which api version it should use

pytwitcherapi.session.**TWITCH_KRAKENURL = 'https://api.twitch.tv/kraken/'**
The baseurl for the twitch api

pytwitcherapi.session.**TWITCH_STATUSURL = 'http://twitchstatus.com/api/status?type=chat'**
str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

pytwitcherapi.session.**TWITCH_USHERURL = 'http://usher.twitch.tv/api/'**
The baseurl for the twitch usher api

pytwitcherapi.session.**absolute_import = _Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)**

pytwitcherapi.session.**log = <logging.Logger object>**
Instances of the Logger class represent a single logging channel. A "logging channel" indicates an area of an application. Exactly how an "area" is defined is up to the application developer. Since an application can have any number of areas, logging channels are identified by a unique string. Application areas can be nested (e.g. an area of "input processing" might include sub-areas "read CSV files", "read XLS files" and "read Gnumeric files"). To cater for this natural nesting, channel names are organized into a namespace hierarchy where levels are separated by periods, much like the Java or Python package namespace. So in the instance given above, channel names might be "input" for the upper level, and "input.csv", "input.xls" and "input.gnu" for the sublevels. There is no arbitrary limit to the depth of nesting.

Module contents

Data

absolute_import

pytwitcherapi.**absolute_import** = _Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/Pytwitcher/pytwitcherapi/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

pytwitcherapi could always use more documentation, whether as part of the official pytwitcherapi docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/Pytwitcher/pytwitcherapi/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.

- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *pytwitcherapi* for local development.

1. Fork the *pytwitcherapi* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pytwitcherapi.git
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

4. When you're done making changes, check that your changes pass style and unit tests, including testing other Python versions with tox:

```
$ tox
```

To get tox, just pip install it.

5. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, and 3.3, and for PyPy. Check https://travis-ci.org/Pytwitcher/pytwitcherapi under pull requests for active pull requests or run the `tox` command and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ py.test test/test_pytwitcherapi.py
```

Credits

Development Lead

- David Zuber <zuber.david@gmx.de>

Contributors

- Benjamin Coriou
- Tim Woocker

History

0.1.1 (2015-03-15)

- First release on PyPI.
- Pulled pytwitcherapi out of main project pytwitcher

0.1.2 (2015-03-15)

- Fix wrapping search stream results due to incomplete channel json

0.1.3 (2015-03-23)

- Refactor twitch module into models and session module

0.1.4 (2015-03-23)

- Fix wrap json using actual class instead of cls

0.2.0 (2015-04-12)

- Authentication: User can login and TwitchSession can retrieve followed streams.

0.3.0 (2015-05-08)

- Easier imports. Only import the package for most of the cases.
- Added logging. Configure your logger and pytwitcher will show debug messages.

0.3.1 (2015-05-09)

- Fix login server shutdown by correctly closing the socket

0.4.0 (2015-05-12)

- IRC client for twitch chat

0.5.0 (2015-05-13)

- IRC v3 Tags for messages

0.5.1 (2015-05-13)

- Fix coverage reports via travis

0.6.0 (2015-05-16)

- Add limit for sending messages

0.7.0 (2015-05-16)

- IRCCLient manages two connections. Receives own messages from the server (with tags).
- Improved test thread saftey

0.7.1 (2015-05-22)

- IRCClient shutdown is now thread-safe through events

0.7.2 (2015-05-30)

- Add TwitchSession.get_emote_picture(emote, size).
- Capabilities for chat: twitch.tv/membership, twitch.tv/commands, twitch.tv/tags

0.8.0 (2015-05-31)

- Replace context managers for apis with dedicated methods. The context managers made it difficult to use a session thread-safe because they relied (more heavily) on the state of the session.

0.9.0 (2016-09-16)

- Remove `on_schedule` argument for irc client. `irc >=15.0` required.
- #17: Always submit a client id in the headers. Credits to Coriou.
- Client ID can be provided via environment variable `PYTWITCHER_CLIENT_ID`.

0.9.1 (2016-09-18)

- Make example chat client python 3 compatible
- #16: Ignore unknown arguments from twitchstatus
- Use Client ID for old api requests as well

0.9.2 (2017-08-27)

- Fix compatibility to `irc>=16.0`. Thanks to crey4fun.

0.9.3 (2017-08-27)

- Re-release of `0.9.2`

CHAPTER 3

Feedback

If you have any suggestions or questions about **pytwitcherapi** feel free to email me at zuber.david@gmx.de.

If you encounter any errors or problems with **pytwitcherapi**, please let me know! Open an Issue at the GitHub https://github.com/Pytwitcher/pytwitcherapi main repository.

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

M

N

O

P

Q

R

S

T

U

V

W