libturpial Documentation

Release 1.7.0-b2

Wil Alvarez

March 10, 2014

libturpial is a library that handles multiple microblogging protocols. It implements a lot of features and aims to support all the features for each protocol. At the moment it supports Twitter and Identi.ca and is the backend used for Turpial.

libturpial is currently under heavy development, so probably you will find bugs or undesired behavior. In this cases please report issues at:

https://github.com/satanas/libturpial/issues

We will be very graceful for your contributions.

CHAPTER 1

Features

- Twitter API
- Identi.ca API
- HTTP and OAuth authentication
- Proxy
- Services for shorten URL and upload images
- Multiple accounts, multiple columns
- User configuration
- Image preview
- Filters

CHAPTER 2

Quickstart

2.1 Quickstart

Are you eager to start coding? Take a look to this section, it will get you a quick introduction on how to use libturpial.

Before start, you must check that libturpial is installed and up-to-date. Check our Wiki page for installing libturpial

2.1.1 Create an account

First thing you need to use libturpial is an account. To create an account import the account module:

```
>>> from libturpial.api.models.account import Account
```

Let's create a new Twitter account:

```
>>> account = Account.new('twitter')
```

This account it's empty, hence pretty useless, except for one thing: authentication. Before you can make real use of an account it must be authenticated. To achieve that, let's ask for OAuth access:

```
>>> url = account.request_oauth_access()
```

This will return an OAuth valid URL that you must visit. In this case, it will redirect you to Twitter, so you can authorize Turpial as a third party app. Then copy the PIN (for example: '1234567') returned by Twitter and authorize the account:

```
>>> account.authorize_oauth_access('1234567')
```

Now you have a valid libturpial account to play with. Remember that this account is not stored in disk yet. To save credentials you need to read how to register the account.

2.1.2 Register the account

Let's assume that you have the same account of the previous section (valid and authenticated). To register the account (and with this, save the credentials on disk) you need a new instance of Core. First, let's import the Core module:

```
>>> from libturpial.api.core import Core
```

And now, let's instantiate Core:

```
>>> core = Core()
```

At this moment, Core will try to load any previous registered account to make it available. This shouldn't take more than a few seconds (that time could vary depending on your internet connection). When done, we can register our fresh-new account created in the previous section. To do so:

```
>>> my_account_id = core.register_account(account)
```

That will return the account id if everything went fine.

It's important that you see the Core as a director, it can handle a lot of things at once but we need to specify exactly where to perform what we want to do. That's the reason why almost all methods on core receive as first argument the account_id, it's like if you must to tell:

```
Core, on this account do that.
```

It's not hard once you get used, it's like: "sudo make me a sandwich" ;) (CC-BY-NC by XKCD)

2.1.3 Fetch the timeline

Now that we have a registered account we would be able to tinker a little bit with Twitter. Let's fetch the account timeline:

```
>>> timeline = core.get_column_statuses(my_account_id, 'timeline', count=5)
```

You can print each retrieved status with:

```
>>> for status in timeline:
...     print status
...
<libturpial.api.models.status.Status instance at 0x1031583f8>
<libturpial.api.models.status.Status instance at 0x103159050>
<libturpial.api.models.status.Status instance at 0x1031590e0>
<libturpial.api.models.status.Status instance at 0x1031731b8>
<libturpial.api.models.status.Status instance at 0x103173320>
```

But wait, those are just `libturpial.api.models.status.Status` object. Let's print something more useful:

```
>>> for status in timeline:
...     print "@%s: %s" % (status.username, status.text)
...
@lombokdesign: La calidad de un link #infografia #infographic #internet #marketing http://t.co/8UF3m
@TheHackersNews: Edward #Snowden's Father My Son Is Not A Traitor http://t.co/y6j8uB6832 #nsa
@Lavinotintocom: #FutVe Buda Torrealba espera alejarse de las lesiones http://t.co/XX53yCY2zv
@Lavinotintocom: #Eliminatorias César Farías: "Seguimos teniendo fe" #Vinotinto http://t.co/QfiMsxpA
@razonartificial: SDL 2.0: Release Candidate lista  http://t.co/B6jVOLly3Y vía @genbetadev
```

Interesting, isn't? We can play with all the available attributes of the status object. Check the status reference for more information.

With the *get_column_statuses* we can fetch statuses from any available column, you just need to change 'timeline' for the desired column.

2.1.4 Fetch any other column

We already know how to fetch the timeline, but what about if we want to fetch any other column? Or even an user list? Well, let's check first which are the available options:

```
>>> all_columns = core.all_columns()
```

This will return a dict with all the available columns per account, so let's print the slug of each one:

```
>>> for key, columns in all_columns.iteritems():
...     print "For %s account:" % key
...     for column in columns:
...         print "  %s" % column.slug
...
For foo-twitter account:
  timeline
  replies
  directs
  sent
  favorites
```

Now we can fetch some other statuses, for example our favorites:

```
>>> favorites = core.get_column_statuses(my_account_id, 'favorites')
```

Or maybe our directs:

```
>>> directs = core.get_column_statuses(my_account_id, 'directs')
```

2.1.5 Working with statuses

Update a status is as simple as:

```
>>> core.update_status(my_account_id, 'Test from libturpial')
```

If you want to reply a status made by a friend (identified with the id '123456789') then you will need to do something like this:

```
>>> core.update_status(my_account_id, '@foouser Hey! I am answering your tweet', '123456789')
```

You can even broadcast a status throught several accounts passing a list with all the account and the text you want to update:

```
>>> core.broadcast_status([my_account_id1, my_account_id2], 'This is a broadcast test')
```

Let's say that you loved a tweet recently posted by a friend and identified by the id '123456789', it's easy mark it as favorite:

```
>>> core.mark_status_as_favorite(my_account_id, '123456789')
```

Besides, you want to share that lovely tweet with all your followers? No problem:

```
>>> core.repeat_status(my_account_id, '123456789')
```

You realize about that nasty tweet on your favs? Get ride off it:

```
>>> core.unmark_as_favorite(my_account_id, '123456789')
```

Posted a tweet with a typo again? Let's delete that mistake:

```
>>> core.destroy_status(my_account_id, '123456789')
```

And there are more methods that you can use to handle your statuses. Just take a look to the core documention.

2.1.6 Managing your friendship

Another interesing features about libturpial is that it lets you handle your friends.

Let's assume that you love the tweets that @a_lovely_account do every day. Well you can follow that account with:

```
>>> core.follow(my_account_id, 'a_lovely_account')
```

Or probably you're tired of those boring tweets of @boring_friend, just unfollow (it's therapeutic):

```
>>> core.unfollow(my_account_id, 'boring_friend')
```

But look, you and I know that always there are bots that bother you every single minute, let's block them:

```
>>> core.block(my_account_id, 'annoying_bot')
```

And report it as spam:

```
>>> core.report_as_spam(my_account_id, 'annoying_bot')
```

That way Twitter can do something about it.

Now, there is this friend that you really love but he takes seriously the unfollow thing and you are just tired of the no-sense tweets he does. No problem, *mute* is for you:

```
>>> core.mute('my_psycho_friend')
```

With mute, libturpial simply hides all the tweets related to this guy without unfollow him. He will never notice that you are not reading his post. Neat, isn't? ;)

But wait, this action is reversible. You can give him voice again:

```
>>> core.unmute('my_psycho_friend')
```

A final tip, do you want to know if @your_fav_account follows you? Use this:

```
>>> core.verify_friendship(my_account_id, 'your_fav_account')
```

This return *True* if they actually follows you or *False* otherwise.

2.1.7 Services

libturpial include support for short URLs, upload pictures and preview pictures. For the first two you can chose which to use from a wide of options. To check which services are available for shorten URL:

```
>>> core.available_short_url_services()
['snipr.com', 'short.to', 'twurl.nl', 'buk.me', 'ub0.cc', 'fwd4.me', 'short.ie', 'burnurl.com', 'git
```

You can verify which one is currently selected:

```
>>> core.get_shorten_url_service()
'migre.me'
```

And even select a different one:

```
>>> core.set_shorten_url_service('is.gd')
```

To short a long URL, do something like this:

```
>>> core.short_single_url('http://turpial.org.ve/news/')
'http://is.gd/Qq7Cdo'
```

But there is more, you can short all the URLs detected in a bunch of text:

```
>>> message = "This is the URL of the libturpial documentation source https://github.com/satanas/libt
>>> new_message = core.short_url_in_message(message)
>>> print new_message
This is the URL of the libturpial documentation source http://is.gd/BJn0WO
```

To upload images the process is kind of similar. You check the available services:

```
>>> core.available_upload_media_services()
['mobypicture', 'yfrog', 'twitpic', 'twitgoo', 'img.ly']
```

Verify the current one selected:

```
>>> core.get_upload_media_service()
'yfrog'
```

And select a different one:

```
>>> core.set_upload_media_service('twitpic')
```

Now, to upload a picture you only need the absolute path to the file and maybe a message to post within the picture (only if the service allows pictures with messages):

```
>>> core.upload_media(my_account_id, '/path/to/my/image.png', 'This is my pretty picture')
'http://twitpic.com/cytmf2'
```

Almost all services support JPEG, PNG and GIF format.

libturpial also handle the previsulization process of tweeted images for you. Imagine that your best friend posted a picture and you want to see it, just fetch the image with:

```
>>> preview = core.preview_media('http://twitpic.com/cytmf2')
```

libturpial will fetch the image and will store it on a temporary file, returning a `libturpial.api.models.media.Media` object. You can get the path of the temporary image with:

```
>>> preview.path
'/var/folders/1b/sq85x9v95nl44d2ccdb0_kmc0000gp/T/twitpic.com_cytmf2.jpg'
```

And even check if it's really an image (libturpial will support image, video and maps on the near future):

```
>>> preview.is_image()
True
```

2.1.8 Further information

Previous sections were a brief introduction to the whole power of libturpial. For more information please check the full documentation

CHAPTER 3

Reference

This part of the documentation shows you details about specific functions, methods and classes in libturpial.

3.1 API

The API module contains the modules and classes that will be used for any developer that want to create an application or script based on libturpial. We can call it the *public interface*. Here you can find the Core, all the models that represent libturpial entities and the Managers

3.1.1 Core

class `libturpial.api.core.`**`Core`**(*load_accounts=True*)

This is the main object in libturpial. This should be the only class you need to instantiate to use libturpial. Most important arguments used in Core are *account_id* and *column_id*.

- account_id: Is a composite string formed by the **username** and the **protocol_id** that identify every single account.
- column_id: Is composite string formed by **account_id** and the **column_name** that identify one column of one account.

Examples of account_id:

```
>>> my_twitter_account = 'foo-twitter'
>>> my_identica_account = 'foo-identica'
```

Example of column_id:

```
>>> twitter_timeline = 'foo-twitter-timeline'
>>> identica_replies = 'foo-identica-replies'
```

When you instantiate Core it will load all registered accounts automatically, so you don't need to worry about it. If you already registered the accounts before, they will be available after you create the core object.

All the Core methods will return an object defined in `libturpial.api.models` or a valid python object if request is successful, otherwise they will raise an exception.

If the request returns an array, you can iterate over the elements with:

```
>>> for object in response:
>>>     print object
```

In all the following functions the following apply: *account_id* must be a string ("username-service") *column_id* must be a string ("columnname-username-service")

fetch_image(*url*)
Retrieve an image by it *URL*. Return the binary data of the image

list_accounts()
Return an array with the ids of all registered accounts. For example:

```
>>> ['foo-twitter', 'foo-identica']
```

register_account(*account*)
Register *account* into config files. *account* must be a valid and authenticated `libturpial.api.models.account.Account` object.

When instantiating Core() all accounts get automatically registered.

Return a string with the id of the account registered.

unregister_account(*account_id*, *delete_all=False*)
Removes the account identified by *account_id* from memory. If *delete_all* is **True** it deletes all the files asociated to that account from disk otherwise the account will be available the next time you load Core.

Return a string with the id of the account unregistered.

all_columns()
Return a dictionary with all columns per account. Example:

```
>>> {'foo-twitter': [libturpial.api.models.Column foo-twitter-timeline,
    libturpial.api.models.Column foo-twitter-replies,
    libturpial.api.models.Column foo-twitter-direct,
    libturpial.api.models.Column foo-twitter-sent,
    libturpial.api.models.Column foo-twitter-favorites]}
```

register_column(*column_id*)
Register a column identified by *column_id* column and return a string with the id of the column registered on success.

unregister_column(*column_id*)
Removes the column identified by *column_id* from config and return a string with the id if the column unregistered on success.

list_protocols()
Return an array with the ids of supported protocols. For example:

```
>>> ['twitter', 'identica']
```

available_columns()
Return a dictionary with all available (non-registered-yet) columns per account. Example:

```
>>> {'foo-twitter': [libturpial.api.models.Column foo-twitter-direct,
    libturpial.api.models.Column foo-twitter-replies,
    libturpial.api.models.Column foo-twitter-sent]}
```

registered_columns()
Return a *dict* with `libturpial.api.models.column.Column` objects per column registered. This method DO NOT return columns in the order they have been registered. For ordered columns check `registered_columns_by_order()`

registered_columns_by_order()
Return a *list* with `libturpial.api.models.column.Column` objects per each column in the same order they have been registered.

registered_accounts()
Return a *dict* with all registered accounts as an array of `libturpial.api.models.account.Account` objects registered

get_single_column(*column_id*)
Return the `libturpial.api.models.column.Column` object identified with *column_id*

get_single_account(*account_id*)
Return the `libturpial.api.models.account.Account` object identified with *account_id*

get_column_statuses(*account_id*, *column_id*, *count=20*, *since_id=None*)
Fetch the statuses for the account *account_id* and the column *column_id*. *count* let you specify how many statuses do you want to fetch, values range goes from 0-200. If *since_id* is not **None** libturpial will only fetch statuses newer than that.

get_public_timeline(*account_id*, *count=20*, *since_id=None*)
Fetch the public timeline for the service associated to the account *account_id*. *count* and *since_id* work in the same way that in `libturpial.api.core.Core.get_column_statuses()`

get_followers(*account_id*, *only_id=False*)
Return a `libturpial.api.models.profile.Profile` list with all the followers of the account *account_id*

get_following(*account_id*, *only_id=False*)
Return a `libturpial.api.models.profile.Profile` list of all the accounts that *account_id* follows

get_all_friends_list()
Return a list with all the username friends of all the registered accounts.

get_user_profile(*account_id*, *user=None*)
Return the profile of the *user*, using the *account_id*, if user is None, it returns the profile of account_id itself.

update_status(*account_id*, *text*, *in_reply_id=None*, *media=None*)
Updates the account *account_id* with content of *text*

if *in_reply_id* is not None, specifies the tweets that is being answered.

media can specify the filepath of an image. If not None, the status is posted with the image attached. At this moment, this method is only valid for Twitter.

broadcast_status(*account_id_array*, *text*)
Updates all the accounts in account_id_array with the content of *text*

if account_id_array is None or an empty list all registered accounts get updated.

destroy_status(*account_id*, *status_id*)
Deletes status of *account_id* specified by *status_id*

get_single_status(*account_id*, *status_id*)
Retrieves a single status with *account_id* that corresponds with *status_id*

repeat_status(*account_id*, *status_id*)
Allows to send the same status again by using repeat option in API

mark_status_as_favorite(*account_id*, *status_id*)
Marks status of *account_id* specified by *status_id* as favorite

unmark_status_as_favorite(*account_id*, *status_id*)
Unmarks status of *account_id* specified by *status_id* as favorite

send_direct_message(*account_id*, *username*, *message*)
Sends a direct update with the contant of *message* to *username* using *account_id*

destroy_direct_message(*account_id*, *status_id*)
Deletes a direct update from *account_id* defined by its *status_id*

update_profile(*account_id*, *fullname=None*, *url=None*, *bio=None*, *location=None*)
Updates the *account_id* public profile with the information in variables fullname = Complete account name url = Blog or personal URL of the account bio = Small resume location = Geographic location

follow(*account_id*, *username*, *by_id=False*)
Makes *account_id* a follower of *username*. Return a `libturpial.api.models.profile.Profile` object with the user profile

unfollow(*account_id*, *username*)
Stops *account_id* from being a follower of *username*. Return a `libturpial.api.models.profile.Profile` object with the user profile

block(*account_id*, *username*)
Blocks *username* in *account_id*. Return a `libturpial.api.models.profile.Profile` object with the user profile

unblock(*account_id*, *username*)
Unblocks *username* in *account_id*. Return a `libturpial.api.models.profile.Profile` object with the user profile

report_as_spam(*account_id*, *username*)
Reports *username* as SPAM using *account_id*. Return a `libturpial.api.models.profile.Profile` object with the user profile

mute(*username*)
Adds *username* into the muted list, so that no more statuses from that account are shown

unmute(*username*)
Removes *username* from the muted list, so that statuses from that account are now shown

verify_friendship(*account_id*, *username*)
Return *True* if the owner of *account_id* and *username* are following each other. *False* otherwise.

search(*account_id*, *query*, *count=20*, *since_id=None*, *extra=None*)
Performs a search using Twitter API, defined by: account_id = Account to be used for the search query = Search Term acount = Max number of results since_id = if limited to a status id and on.

get_profile_image(*account_id*, *username*, *use_cache=True*)
Return the local path to a the profile image of *username* in original size. If use_cache is *True* it will try to return the cached file, otherwise it will fetch the real image.

get_status_avatar(*status*)
Return the local path to a the profile image of the username to post *status* in 48x48 px size

get_available_trend_locations(*account_id*)
Return an array of `libturpial.api.models.trend.TrendLocation` objects with all the locations with trending topics registered.

get_trending_topics(*account_id*, *location_id*)
Return an array of `libturpial.api.models.trend.Trend` objects with trending topics for the specified location. *location_id* is the Yahoo! Where On Earth ID for the location.

update_profile_image(*account_id*, *image_path*)
Update profile image of *account_id* with the image specified by *image_path*. Return a `libturpial.api.models.profile.Profile` object with the user profile updated.

register_new_config_option(*section*, *option*, *default_value*)
Register a new configuration *option* in *section* to be handled by external modules. libturpial will set *default_value* as value if the option doesn't exist.

This method should be used if a module that uses libturpial needs to handle configuration options not registered by default.

For example, if you want to register an option to handle notifications on login the code should looks like:

```
>>> core = Core()
>>> core.register_new_config_option('Notifications', 'login', 'on')
```

From this point you can use config methods over this value as usual.

is_muted(*username*)
Return *True* is *username* is muted. *False* otherwise

get_max_statuses_per_column()
Return how many statuses should be fetched in each requests

get_proxy()
Return a `libturpial.api.models.proxy.Proxy` object with the configuration stored in disk.

get_socket_timeout()
Return the timeout set for the socket connections

list_filters()
Return a list with all registered filters

save_filters(*lst*)
Save *lst* a the new filters list

delete_current_config()
Delete current configuration file. This action can not be undone

delete_cache()
Delete all files in cache

get_cache_size()
Return current space used by cache

add_friend(*username*)
Save *username* into the friends list

remove_friend(*username*)
Remove *username* from friends list

3.1.2 Models

Here you can find the representation of the information used by libturpial mapped to Python objects. They handle the methods and properties of each entity and guarantee the data consistency.

Account

class `libturpial.api.models.account.`**Account**(*protocol_id*, *username=None*)
This class represents an user account and holds all it related methods. This is done thanks to one `libturpial.lib.interfaces.protocol.Protocol` instance associated to the user account that handles all the dirty work against the service (Twitter, Identi.ca, etc) as well as one `libturpial.api.models.profile.Profile` model that store the user details.

This is the class you must instanciate if you want to handle/authenticate a user account.

Account let you perform three actions to build an account: create a new account from scratch, create a new account from params and load a previously registered account. To create a new account from scratch do:

```
>>> account = Account.new('twitter')
```

If you know the username too, then you can pass it as argument:

```
>>> account = Account.new('twitter', 'username')
```

At this point, that account is not a valid account yet because it hasn't been authenticated. You should do the authentication by yourself. This is, request OAuth access:

```
>>> url = account.request_oauth_access()
```

That method will return an URL that your user must visit to authorize the app. After that, you must to ask for the PIN returned by the service and execute:

```
>>> account.authorize_oauth_access('the_pin')
```

And voilá! You now have a valid and fully authenticated account ready to be registered in `libturpial.api.core.Core`.

But *Account* let you create accounts passing all the params needed for the OAuth authentication. If you already know those params (user key, user secret and PIN) then you just need to execute:

```
>>> account = Account.new_from_params('twitter', 'username', 'key', 'secret', 'the_pin')
```

And you will have a valid and fully authenticated account ready to be registered in `libturpial.api.core.Core` too.

Now, what if you did all this process before and registered the account in `libturpial.api.core.Core`? Well, you just need to load the account then:

```
>>> account = Account.load('username-twitter')
```

And you will have an account already authenticated and ready to be used.

From this point you can use the method described here to handle the account object.

static new(*protocol_id*, *username=None*)

Return a new account object associated to the protocol identified by *protocol_id*. If *username* is not None it will build the account_id for the account.

This account is empty and must be authenticated before it can be registered in `libturpial.api.core.Core`.

> **Warning:** None information is stored at disk at this point.

static new_from_params(*protocol_id*, *username*, *key*, *secret*, *verifier*)

Return a new account object associated to the protocol identified by *protocol_id* and authenticated against the respective service (Twitter, Identi.ca, etc) using *username*, *key*, *secret* and *verifier* (aka PIN).

This account is authenticated after creation, so it can be registered in `libturpial.api.core.Core` immediately.

> **Warning:** None information is stored at disk at this point.

static load(*account_id*)
Return the account object associated to *account_id* loaded from existing configuration. If the *account_id* does not correspond to a valid account it returns a `libturpial.exceptions.ErrorLoadingAccount` exception.

If credentials in configuration file are empty it returns a `libturpial.exceptions.EmptyOAuthCredentials` exception.

request_oauth_access()
Ask for an OAuth token. Return an URL that must be visited for the user in order to authenticate the app.

authorize_oauth_access(*pin*)
Take the *pin* returned by OAuth service and authenticate the token requested with *request_oauth_access*

save()
Save to disk the configuration and credentials for the account. If the account hasn't been authenticated it will raise an `libturpial.exceptions.AccountNotAuthenticated` exception.

fetch()
Retrieve the user profile information and return the id of the account on success. This method authenticate the account.

fetch_friends()
Retrieve all the friends for the user and return an array of string where each element correspond to the user_id of the friend.

get_columns()
Return an array of `libturpial.api.models.column.Column` with all the available columns for the user

purge_config()
Delete all config files related to this account

delete_cache()
Delete all files cached for this account

get_cache_size()
Return disk space used by cache in bytes

is_authenticated()
Return *True* if the current account has been logged in, *False* otherwise

update_profile(*fullname=None*, *url=None*, *bio=None*, *location=None*)
Update the *fullname*, *url*, *bio* or *location* of the user profile. You may specify one or more arguments. Return an `libturpial.api.models.profile.Profile` object containing the user profile

Column

class `libturpial.api.models.column.`**Column**(*account_id*, *slug*, *singular_unit='tweet'*, *plural_unit='tweets'*)
This model represents a column that holds `libturpial.api.models.status.Status` objects. You need to specify to what *account_id* are they associated, as well as the column *slug*. Available column slugs are available in `libturpial.common.ColumnType`.

Variables

- **id** – Column id (for example: "johndoe-twitter-timeline")
- **slug** – Column slug
- **account_id** – id of account associated to the column

- **size** – max number of statuses that this column can hold
- **singular_unit** – unit used to identify one status (for example: 'tweet')
- **plural_unit** – unit used to identify more than one status (for example: 'tweets')

List

class libturpial.api.models.list.**List**

This class handles the information about a user list. It receives the *id_*, the *user* who owns it, the *name* (also known as slug), the *title* or caption, the number of *suscribers*, it *description* and the units used for plural and singular: *singular_unit*, *plural_unit*.

Variables

- **id** – List id
- **user** – User that owns the list
- **slug** – List name
- **title** – List title
- **suscribers** – Suscribed users (count)
- **description** – List description
- **single_unit** – Singularized unit ('tweet' for example)
- **plural_unit** – Pluralized unit ('tweet' for example)

Status

class libturpial.api.models.status.**Status**

This model represents and holds all the information of a status.

Variables

- **id** – Status id
- **account_id** – Id of the account associated to this status
- **text** – Text of the status
- **username** – Name of the user that updated the status
- **avatar** – Display image of the user that updated the status
- **source** – Client used to upload this status
- **timestamp** – Time of publishing of this status (Unix time)
- **in_reply_to_id** – Contains the id of the status answered (if any)
- **in_reply_to_user** – Contains the user answered with status (if any)
- **is_favorite** – *True* if this status has been marked as favorite. *False* otherwise
- **is_protected** – *True* if this status is from a protected account. *False* otherwise
- **is_verified** – *True* if this status is from a verified account. *False* otherwise
- **repeated** – *True* if this status has been repeated (retweeted) by you. *False* otherwise
- **repeated_by** – More users that have repeated this status

- **repeated_count** – How much times this status has been repeated
- **original_status_id** – Id of the original status (not-repeated)
- **created_at** – Original timestamp from the service
- **datetime** – Humanized representation of the date/time of this status
- **is_own** – *True* if the status belongs to the same user of the associated account. *False* otherwise
- **entities** – A dict with all the entities found in status
- **type** – Status type.

Sometimes a status can hold one or more entities (URLs, hashtags, etc). In this case the entities variable will store a dict with lists for each category. For example:

```
>>> status.entities
{'urls': [], 'hashtags': [], 'mentions': [], 'groups': []}
```

A status can handle two possible types: `libturpial.api.models.status.Status.NORMAL` for regular statuses or `libturpial.api.models.status.Status.DIRECT` for private (direct) statuses.

get_mentions()
Returns all usernames mentioned in status (even the author of the status)

is_direct()
Return *True* if this status is a direct message

get_protocol_id()
Return the *protocol_id* associated to this status

get_source(*source*)
Parse the source text in the status and store it in a `libturpial.api.models.client.Client` object.

Profile

class `libturpial.api.models.profile.`**Profile**
This model handles all the information stored for a user profile.

Variables

- **id** – Profile id
- **account_id** – Account id that fetched this profile
- **fullname** – Full name of the user
- **avatar** – Display image
- **location** – User location
- **url** – User URL
- **bio** – User bio or description
- **following** – Indicate if the user is following to the account_id owner (*True*‘or ‘*False*)
- **followed_by** – Indicate if the user is followed by the account_id owner (*True*‘or ‘*False*)
- **follow_request** – Indicate if there is a pending follow request of this profile
- **followers_count** – Number of followers of this user

- **friends_count** – Number of friends of this user
- **favorites_count** – Number of favorite statuses of this user
- **statuses_count** – Number of statuses this user has done
- **link_color** – Color used to highlight entities (URLs, hashtags, etc)
- **last_update** – Text of the last status updated
- **last_update_id** – Id of the last status updated
- **protected** – Indicate if the profile is protected (*True'or 'False*)
- **verified** – Indicate if the profile is verified (*True'or 'False*)

is_me()
Return *True* if the username of the profile is the same of the associated account, otherwise *False*. This method can be useful to determinate if a status belongs to given account.

get_protocol_id()
Returns the *protocol_id* for this profile

Entity

class libturpial.api.models.entity.**Entity**(*account_id*, *url*, *text*, *search*)
Sometimes a libturpial.api.models.status.Status can content mentions, URLs, hashtags and other class of interactuable objects, so this class models that data in a structured way

Variables

- **account_id** – Id of the account that holds the status
- **url** – A possible URL for the media/content
- **display_text** – Text that must be displayed to user
- **search_for** – Text that must be used to search this object

Client

class libturpial.api.models.client.**Client**(*name=None*, *url=None*)
This class models the information of the client of a given status

Variables

- **name** – Client name
- **url** – Client URL

Media

class libturpial.api.models.media.**Media**(*type_*, *name*, *content*, *path=None*, *info=None*)
This class represents a media object (image, video or map) that can be rendered on the UI. *type_* could be libturpial.api.models.media.IMAGE, libturpial.api.models.media.VIDEO or libturpial.api.models.media.MAP, *name* is the filename and *content* holds the binary data of the media. If *path* is *None* a temporary path will be created using the filename. *info* stores any additional information about the resource.

static new_image(*name*, *content*, *path=None*, *info=None*)
Create an image media

save_content()
Saves the content of the media in the path specified.

is_video()
Returns *True* if the media is a video

is_image()
Returns *True* if the media is an image

is_map()
Returns *True* if the media is a map

Proxy

class libturpial.api.models.proxy.**Proxy**(*host*, *port*, *username=None*, *password=None*, *secure=False*)
This class models the information of a proxy server used to establish an HTTP connection. *host* and *port* are required. For an authenticated proxy set *username* and *password* with according credentilas and if proxy uses HTTPS set *secure* to **True**, otherwise let it empty or **False**

For example, to store the information of a non-authenticated HTTPS proxy:

```
>>> proxy = Proxy('127.0.0.1', '1234', secure=True)
```

And for an authenticated HTTP proxy:

```
>>> proxy = Proxy('127.0.0.1', '1234', 'my_user', 'secret_password')
```

Variables

- **host** – Proxy host address
- **port** – Proxy port
- **username** – (Optional) Username for authentication
- **password** – (Optional) Password for authentication
- **secure** – Indicate whether proxy uses HTTP or HTTPS (Default *False*)

3.1.3 Managers

This classes are used by libturpial.api.core.Core to handle and organize multiple instances of columns and accounts. They are responsible of add, delete and manage both.

Account Manager

class libturpial.api.managers.accountmanager.**AccountManager**(*config*, *load=True*)
This class has methods to manage accounts. You can register new accounts, load and unregister existing accounts.

This manager can be iterated and each element will have the account id and the respective object. For example:

```
>>> for item in accman:
        print item
('foo-twitter', <libturpial.api.models.account.Account object at 0x10c5c2e10>)
('bar-twitter', <libturpial.api.models.account.Account object at 0x10c5c2910>)
```

To check how much accounts are registered simply use the *len* function:

```
>>> len(accman)
2
```

load(*account_id*)
Load and existing account identified by *account_id*. If the load fails an `libturpial.exceptions.ErrorLoadingAccount` exception will raise. Return the id of the account loaded on success

register(*account*)
Register the *account* object passed as argument. If the account hasn't been authenticated it will raise a `libturpial.exceptions.AccountNotAuthenticated` exception. If the account is already registered a `libturpial.exceptions.AccountAlreadyRegistered` exception will raise. Return the id of the account registered on success

unregister(*account_id*, *delete_all*)
Remove the account identified by *account_id* from memory. If *delete_all* is *True* all configuration files are deleted from disk. Be careful because this operation can not be undone. Return the id of the unregistered column on success, *None* otherwise

get(*account_id*)
Obtain the account identified by *account_id*. If the account is not loaded yet, it will be loaded immediately. Return a `libturpial.api.models.account.Account` object on success

list()
Return an alphabetically sorted list with all the ids of the registered accounts.

accounts()
Return a list of `libturpial.api.models.account.Account` objects with all the accounts registered

Column Manager

class `libturpial.api.managers.columnmanager.`**ColumnManager**(*config*)
This class has methods to manage columns. You can register new columns, load and unregister existing columns.

This manager can be iterated and each element will have a list of columns per account. For example:

```
>>> for item in column_manager:
        print item
('foo-twitter', [<libturpial.api.models.column.Column instance at 0x10a9ce368>,
    <libturpial.api.models.column.Column instance at 0x10a9ce8c0>])
```

register(*column_id*)
Register a new column identified by *column_id*. If the column is already registered a `libturpial.exceptions.ColumnAlreadyRegistered` exception will raise. Return the id of the column registered on success

unregister(*column_id*)
Remove the column identified by *column_id* from memory. Return the id of the unregistered column.

get(*column_id*)
Obtain the column identified by *column_id* and return a `libturpial.api.models.column.Column` object on success. None otherwise.

columns()
Return a dict where each key represents an account_id and it value is a list of `libturpial.api.models.column.Column` objects with all the columns registered.

For example:

```
>>> column_manager.columns()
{'foo-twitter': [<libturpial.api.models.column.Column instance at 0x10a9cbb48>,
    <libturpial.api.models.column.Column instance at 0x10a9cb6c8>]}
```

is_registered(*column_id*)
Return *True* if column identified by *column_id* is registered. *False* otherwise.

3.2 Lib

3.2.1 Interfaces

Protocol.py

class libturpial.lib.interfaces.protocol.**Protocol**
Bridge class to define abstract functions that must have any protocol implementation

IDENTICA = 'identica'

TWITTER = 'twitter'

__init__()

__module__ = 'libturpial.lib.interfaces.protocol'

available_trend_locations()
Search for trend locations

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

static availables()

block(*user*)
Blocks the specified user

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

convert_time(*str_datetime*)
Takes the *str_datetime* and convert it into Unix time

destroy_direct_message(*direct_message_id*)
Destroy a direct message

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

destroy_status(*status_id*)
Destroy a posted update

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

follow(*user*, *by_id*)
Follow somebody

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_blocked()
Fetch the list of blocked users

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_conversation(*status_id*)
Fetch a whole conversation starting from *status_id*

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_directs(*count*, *since_id*)
Fetch *count* direct messages received starting from *since_id*. If *since_id* is None it will fetch the last *count* statuses

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_directs_sent(*count*, *since_id*)
Fetch *count* direct messages sent starting from *since_id*. If *since_id* is None it will fetch the last *count* statuses

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_entities(*status*)

get_favorites(*count*)
Fetch *count* favorite statuses starting from *since_id*. If *since_id* is None it will fetch the last *count* statuses

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_followers(*only_id=False*)
Fetch an array of :class list of followers

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_following(*only_id=False*)
Fetch the list of following

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_int_time(*strdate*)
Converts the *strdate* into a Unix time long integer (GMT 0)

get_list_statuses(*list_id*, *since_id*)
Fetch all statuses from *list_id* starting from *since_id*. If *since_id* is None it will fetch the last available statuses

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_lists(*username*)
Fetch all the lists where *username* is part of

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_profile(*user*)
Fetch an especific user profile

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_profile_image(*user*)
Returns the URL for the profile image of the given user

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_public_timeline(*count*, *since_id*)
Fetch *count* statuses from public timeline starting from *since_id*. If *since_id* is None it will fetch the last *count* statuses

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_rate_limits()
Fetch the rate limits for the service

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_repeaters(*status_id*, *only_username=False*)
Fetch all the users that repeated *status_id*

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_replies(*count*, *since_id*)
Fetch *count* mentions starting from *since_id*. If *since_id* is None it will fetch the last *count* statuses

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_sent(*count*, *since_id*)
Fetch *count* sent statuses starting from *since_id*. If *since_id* is None it will fetch the last *count* statuses

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_status(*status_id*)
Fetch the status identified by *status_id*

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

get_str_time(*strdate*)
Converts the *strdate* into a formatted string (GMT 0)

get_timeline(*count*, *since_id*)
Fetch *count* statuses from timeline starting from *since_id*. If *since_id* is None it will fetch the last *count* statuses

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

initialize_http()
Creates a new TurpialHTTP instance that must be stored in self.http

For OAuth do:

```
>>> self.http = TurpialHTTPOAuth(base_url, oauth_options, proxies, timeout)
```

For Basic Auth do:

```
>>> self.http = TurpialHTTPBasicAuth(base_url, proxies, timeout)
```

is_friend(*user*)
Returns True is user follows current account, False otherwise

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

json_to_list(*response*)
Takes a JSON *response* and returns a `libturpial.api.models.list.List` object

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

json_to_profile(*response*)
Takes a JSON *response* and returns a `libturpial.api.models.profile.Profile` object

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

json_to_ratelimit(*response*)
Takes a JSON *response* and returns a `libturpial.api.models.ratelimit.RateLimit` object

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

json_to_status(*response*)
Takes a JSON *response* and returns a `libturpial.api.models.status.Status` object

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

mark_as_favorite(*status_id*)
Mark an update as favorite

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

repeat_status(*status_id*)
Repeat to all your friends an update posted by somebody

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

report_as_spam(*user*)
Blocks and report the specified user as spammer

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

request_access()
Return an OAuth authorization URL. Do not overide if the protocol does not support OAuth

search(*query*, *count*, *since_id=None*, *extra=None*)
Execute a search query in server

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

send_direct_message(*user*, *text*)
Send a direct message

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

setup_user_credentials()
Set the information related to user credentials. *key*, *secret* and *verifier* for the OAuth case and *username*, *password* in the Basic case

trends(*location_id*)
Search for trending topics in *location_id*

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

unblock(*user*)
Unblocks the specified user

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

unfollow(*user*)
Unfollow somebody

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

unmark_as_favorite(*status_id*)
Unmark an update as favorite

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

update_profile(*fullname=None*, *url=None*, *bio=None*, *location=None*)
Update the user profile

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

update_profile_image(*image_path*)
Update user profile image and return user profile object

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

update_status(*text*, *in_reply_to_id=None*, *media=None*)
Post an update

> **Warning:** This is an empty method and must be reimplemented on child class, otherwise it will raise a **NotImplementedError** exception

verify_credentials()

verify_credentials_provider()

Service.py

class libturpial.lib.interfaces.service.**GenericService**

__init__()

__module__ = 'libturpial.lib.interfaces.service'

_get_request(*url*, *data=None*)
Process a GET request and returns a text plain response

_json_request(*url*)
Process a GET request and returns a json hash

_parse_xml(*key*, *xml*)
Simple XML parser

_quote_url(*url*)

do_service(*arg*)

3.2.2 Http

Generic module to handle HTTP requests in libturpial

TurpialHTTPBase

class libturpial.lib.http.**TurpialHTTPBase**(*base_url*, *proxies=None*, *timeout=None*)
This class is the abstraction of the HTTP protocol for libturpial. It handles all the magic behind http requests: building, authenticating and fetching resources, in other words, it standarize the way you interact with end points and services.

You shouldn't instantiate this class, instead you should use the proper implementation for OAuth (libturpial.api.interfaces.http.TurpialHTTPOAuth) or Basic Auth (libturpial.api.interfaces.http.TurpialHTTPBasicAuth) or even develop your own implementation if the authentication method is not supported.

base_url is the part of the URL common for all your requests (api.twitter.com/1.1 for example). *proxies* is a dict where you define HTTP/HTTPS proxies.

```
>>> proxies = {
    "http": "http://10.10.1.10:3128",
    "https": "https://10.10.1.10:1080",
}
```

If your proxy uses HTTP authentication then you can set user and password with something like this:

```
>>> proxies = {
    "http": "http://user:pass@10.10.1.10:3128",
}
```

timeout is the maximum time in seconds that TurpialHTTPBase will wait before cancelling the current request. If *timeout* is not specified then *DEFAULT_TIMEOUT* will be used.

__init__(*base_url*, *proxies=None*, *timeout=None*)

__module__ = 'libturpial.lib.http'

get(*uri*, *args=None*, *_format='json'*, *base_url=None*, *secure=False*, *id_in_url=True*)
Performs a GET request against the *uri* resource with *args*. You can specify the *_format* ('json' or 'xml') and can specify a different *base_url*. If *secure* is True the request will be perform as HTTPS, otherwise it will be performed as HTTP.

This method is an alias for **request** function using 'GET' as method.

post(*uri*, *args=None*, *_format='json'*, *base_url=None*, *secure=False*, *id_in_url=True*, *files=None*)
Performs a POST request against the *uri* resource with *args*. You can specify the *_format* ('json' or 'xml') and can specify a different *base_url*. If *secure* is True the request will be perform as HTTPS, otherwise it will be performed as HTTP.

This method is an alias for **request** function using 'POST' as method.

request (*method*, *uri*, *args=None*, *_format='json'*, *alt_base_url=None*, *secure=False*, *id_in_url=True*, *files=None*)
Performs a GET or POST request against the *uri* resource with *args*. You can specify the *_format* ('json' or 'xml') and can specify a different *base_url*. If *secure* is True the request will be perform as HTTPS, otherwise it will be performed as HTTP

set_proxy (*host*, *port*, *username=''*, *password=''*, *https=False*)
Set an HTTP/HTTPS proxy for all requests. You must pass the *host* and *port*. If your proxy uses HTTP authentication you can pass the *username* and *password* too. If *https* is True your proxy will use HTTPS, otherwise HTTP.

set_timeout (*timeout*)
Configure the maximum time (in seconds) to wait before killing the current request.

sign_request (*httpreq*)
This is the method you need to overwrite if you subclass `libturpial.api.interfaces.http.TurpialHTTPBase`

httpreq is the current request (a `libturpial.api.interfaces.http.TurpialHTTPRequest` object), you need to apply all the authentication/authorization methods to that object and make it valid; then exit. There is no need to return any value because all changes are done directly over the object.

If this method is not overwritten it will return a **NotImplementedError** exception.

TurpialHTTPOAuth

class `libturpial.lib.http.`**TurpialHTTPOAuth** (*base_url*, *oauth_options*, *user_key=None*, *user_secret=None*, *verifier=None*, *proxies=None*, *timeout=None*)
Implementation of TurpialHTTPBase for OAuth. *base_url* is the part of the URL common for all your requests (api.twitter.com/1.1 for example). *oauth_options* is a dict with all the OAuth configuration parameters. It must looks like:

```
>>> oauth_options = {
    'consumer_key': 'APP_CONSUMER_KEY',
    'consumer_secret': 'APP_CONSUMER_SECRET',
    'request_token_url': 'http://request_url',
    'authorize_token_url': 'http://authorize_url',
    'access_token_url': 'http://access_url',
}
```

consumer_key and *consumer_secret* are the credentials for your application (they must be provided by the OAuth service). *request_token_url*, *authorize_token_url* and *access_token_url* are the URLs designed by the OAuth service to fetch and authorize an OAuth token.

user_key, *user_secret* and *verifier* (a.k.a. PIN) are the token credentials granted to the user after the OAuth dance. They are optional and needed only if the user was already authenticated, otherwise you need to fetch a new token.

proxies and *timeout* work in the same way that in `libturpial.api.interfaces.http.TurpialHTTPBase`.

•To request an OAuth token create a new TurpialHTTPOAuth object and request a token to your service:

```
>>> http = TurpialHTTPOAuth(base_url, oauth_options)
>>> url_to_auth = http.request_token()
```

Ask the user go to the *url_to_auth* URL and authorize your app. Then get the PIN the service will deliver to your user and authorize the token:

```
>>> token = http.authorize_token(pin)
```

Use this token to store the key, secret and pin in a safe place to use it from now on. Then inform to TurpialHTTPOAuth that you have access granted:

```
>>> http.set_token_info(token.key, token.secret, token.verifier)
```

- To use an existing token to fetch a resource create a new TurpialHTTPOAuth object and pass the user_key, user_secret and pin (verifier):

```
>>> http = TurpialHTTPOAuth(base_url, oauth_options, user_key, user_secret,
        verifier)
```

- To perform a request use the get or post method:

```
>>> http.get('/my_first/end_point', args={'arg1': '1'}, _format='json')
>>> http.post('/my_second/end_point', args={'arg1': '2'}, _format='json')
```

__init__(*base_url*, *oauth_options*, *user_key=None*, *user_secret=None*, *verifier=None*, *proxies=None*, *timeout=None*)

__module__ = 'libturpial.lib.http'

authorize_token(*pin*)
Uses the *pin* returned by the service to authorize the current token. Returns an `oauth.OAuthToken` object.

request_token()
Ask to the service for a fresh new token. Returns an URL that the user must access in order to authorize the client.

request_xauth_token(*username*, *password*)
Request a limited token without using the whole OAuth flow, it just uses the username and password through xAuth

set_token_info(*user_key*, *user_secret*, *verifier*)
Creates a new token using the existing *user_key*, *user_secret* and *verifier*. Use this method

sign_request(*httpreq*)
Signs the *httpreq* for OAuth using the previously defined user token

TurpialHTTPBasicAuth

class `libturpial.lib.http.`**TurpialHTTPBasicAuth**(*base_url*, *proxies=None*, *timeout=None*)
Implementation of TurpialHTTPBase for the HTTP Basic Authentication. *base_url* is the part of the URL common for all your requests (identi.ca/api for example). *username* and *password* are the username credentials. *proxies* and *timeout* work in the same way that in `libturpial.api.interfaces.http.TurpialHTTPBase`.

proxies and *timeout* work in the same way that in `libturpial.api.interfaces.http.TurpialHTTPBase`.

- To fetch a resource using Basic Authentication just create a new TurpialHTTPBasicAuth object and pass the user credentials as parameters:

```
>>> http = TurpialHTTPBasicAuth(base_url, username, password)
```

Then perform the request desired using get or post methods:

```
>>> http.get('/my_first/end_point', args={'arg1': '1'}, _format='json')
>>> http.post('/my_second/end_point', args={'arg1': '2'}, _format='json')
```

__init__ (*base_url*, *proxies=None*, *timeout=None*)

__module__ = 'libturpial.lib.http'

set_user_info (*username*, *password*)
Set the *username* and *password* for the basic authentication

sign_request (*httpreq*)
The *httpreq* is signed using the Authorization header as documented in the Basic Access Authentication Wiki

TurpialHTTPRequest

class libturpial.lib.http.**TurpialHTTPRequest** (*method*, *uri*, *headers=None*, *params=None*, *_format='json'*, *secure=False*, *files=None*)
Encapsulate an URL request into a python object

3.2.3 Protocols

Twitter

class libturpial.lib.protocols.twitter.twitter.**Main**
Twitter implementation for libturpial

__init__ ()

__module__ = 'libturpial.lib.protocols.twitter.twitter'

authorize_token (*pin*)

available_trend_locations ()

block (*screen_name*)

check_for_errors (*response*)
Receives a json response and raise an exception if there are errors

destroy_direct_message (*direct_message_id*)

destroy_status (*status_id*)

follow (*screen_name*, *by_id=False*)

get_blocked ()

get_conversation (*status_id*)

get_directs (*count=20*, *since_id=None*)

get_directs_sent (*count=20*, *since_id=None*)

get_entities (*tweet*)

get_favorites (*count=20*)

get_followers (*only_id=False*)

get_following (*only_id=False*)

get_list_statuses (*list_id*, *count=20*, *since_id=None*)

get_lists (*username*)

get_oauth_token ()

get_profile(*user*)

get_profile_image(*user*)

get_public_timeline(*count=20*, *since_id=None*)

get_repeaters(*status_id*, *only_username=False*)

get_replies(*count=20*, *since_id=None*)

get_sent(*count=20*, *since_id=None*)

get_status(*status_id*)

get_timeline(*count=20*, *since_id=None*)

initialize_http()

is_friend(*user*)

json_to_list(*response*)

json_to_profile(*response*)

json_to_status(*response*, *column_id=''*, *type_=1*)

json_to_trend(*response*)

json_to_trend_location(*response*)

mark_as_favorite(*status_id*)

repeat_status(*status_id*)

report_as_spam(*screen_name*)

request_token()

search(*query*, *count=20*, *since_id=None*, *extra=None*)

send_direct_message(*screen_name*, *text*)

setup_user_credentials(*account_id*, *key*, *secret*, *verifier=None*)

setup_user_info(*account_id*)

trends(*location_id*)

unblock(*screen_name*)

unfollow(*screen_name*)

unmark_as_favorite(*status_id*)

update_profile(*fullname=None*, *url=None*, *bio=None*, *location=None*)

update_profile_image(*image_path*)

update_status(*text*, *in_reply_id=None*, *media=None*)

verify_credentials()

verify_credentials_provider(*format_='json'*)

Identi.ca

class libturpial.lib.protocols.identica.identica.**Main**

Identi.ca implementation for libturpial

GROUP_PATTERN = <_sre.SRE_Pattern object at 0x1d28b20>

_Main__build_basic_args(*count*, *since_id*)

__init__()

__module__ = 'libturpial.lib.protocols.identica.identica'

block(*screen_name*)

check_for_errors(*response*)

Receives a json response and raise an exception if there are errors

destroy_direct_message(*direct_message_id*)

destroy_status(*status_id*)

follow(*screen_name*, *by_id=False*)

get_blocked()

get_conversation(*status_id*)

get_directs(*count=20*, *since_id=None*)

get_directs_sent(*count=20*, *since_id=None*)

get_entities(*status*)

get_favorites(*count=20*)

get_followers(*only_id=False*)

get_following(*only_id=False*)

get_list_statuses(*list_id*, *user*, *count=20*, *since_id=None*)

get_lists(*username*)

get_profile(*user*)

get_profile_image(*user*)

get_public_timeline(*count=20*, *since_id=None*)

get_rate_limits()

get_replies(*count=20*, *since_id=None*)

get_sent(*count=20*, *since_id=None*)

get_status(*status_id*)

get_timeline(*count=20*, *since_id=None*)

initialize_http()

is_friend(*user*)

json_to_profile(*response*)

json_to_status(*response*, *column_id=''*, *type_=1*)

mark_as_favorite(*status_id*)

repeat_status(*status_id*)

search(*query*, *count=20*, *since_id=None*)

send_direct_message(*screen_name*, *text*)

setup_user_credentials(*account_id*, *username*, *password*)

unblock(*screen_name*)

unfollow(*screen_name*)

unmark_as_favorite(*status_id*)

update_profile(*p_args*)

update_status(*text*, *in_reply_id=None*)

verify_credentials()

3.3 Configuration

This module contains all the classes involved in Turpial configuration. All files are stored on *~/.config/turpial/* and cache is stored on *~/.cache/turpial/*. Absolutes directories will vary depending on the operating system.

In configuration folder you will find:

- accounts: A directory where live sub-directories with the configuration of every single account.
- config: The global configuration file. Here is stored all the settings related to the application behavior (even for the graphic interface)
- filtered: A plain text file where libturpial stores all filters applied to timelines.
- friends: A plain text file that contains the list of all the friends of all the accounts registered

Module to handle basic configuration of Turpial

class libturpial.config.**AppConfig**(*basedir='/home/docs/.config/turpial'*, *default=None*)
Handle app configuration

class libturpial.config.**ConfigBase**(*default=None*)
Base configuration

3.4 Constants

DEFAULT_TIMEOUT
Default time that TurpialHTTPBase waits until killing a request. Value: 20 (seconds)

FORMAT_XML
Constant to identify XML requests

FORMAT_JSON
Constant to identify JSON requests

3.5 Exceptions

Module to handle custom exceptions for libturpial

exception libturpial.exceptions.**URLShortenError**(*message*)

exception libturpial.exceptions.**UploadImageError**(*message=None*)

exception libturpial.exceptions.**EmptyOAuthCredentials**

exception libturpial.exceptions.**EmptyBasicCredentials**

exception libturpial.exceptions.**ErrorCreatingAccount**

exception libturpial.exceptions.**ErrorLoadingAccount**(*message*)

exception libturpial.exceptions.**AccountNotAuthenticated**

exception libturpial.exceptions.**AccountSuspended**

exception libturpial.exceptions.**AccountAlreadyRegistered**

exception libturpial.exceptions.**ColumnAlreadyRegistered**

exception libturpial.exceptions.**StatusMessageTooLong**

exception libturpial.exceptions.**StatusDuplicated**

exception libturpial.exceptions.**ResourceNotFound**

exception libturpial.exceptions.**UserListNotFound**

exception libturpial.exceptions.**ServiceOverCapacity**

exception libturpial.exceptions.**InternalServerError**

exception libturpial.exceptions.**ServiceDown**

exception libturpial.exceptions.**InvalidOrMissingCredentials**

exception libturpial.exceptions.**InvalidOrMissingArguments**

exception libturpial.exceptions.**ExpressionAlreadyFiltered**

exception libturpial.exceptions.**BadOAuthTimestamp**

exception libturpial.exceptions.**ErrorSendingDirectMessage**(*message*)

exception libturpial.exceptions.**RateLimitExceeded**

exception libturpial.exceptions.**InvalidOAuthToken**

exception libturpial.exceptions.**URLShortenError**(*message*)

exception libturpial.exceptions.**NoURLToShorten**

exception libturpial.exceptions.**URLAlreadyShort**

exception libturpial.exceptions.**PreviewServiceNotSupported**

exception libturpial.exceptions.**UploadImageError**(*message=None*)

exception libturpial.exceptions.**NotSupported**

CHAPTER 4

Further information

For more information about the development process, please go to:

http://wiki.turpial.org.ve/dev:welcome

l