Groupy Documentation

Release 0.7.1

Robert Grant

Mar 29, 2017

Contents

The simple yet powerful wrapper for the GroupMe API.

CHAPTER 1

Table of Contents

Introduction

About GroupMe

GroupMe is a messaging app that allows you to create groups and have others join them with you. In addition to group messaging, fellow group members can be messaged directly. GroupMe is available for most platforms, lets you share links, images, and locations, and messages can be favorited (or "liked"). You can read more about GroupMe, but the best part about it is that they provide an API!

The GroupMe API is documented, but there are some notable omissions. Many of the properties of groups and messages are not documented, and some features are only hinted at by the documentation. Regardless, all of the information about your groups, their members, their messages, you, and your bots can be obtained through the GroupMe API. You can read the API documentation for more (or less) detailed information.

About Groupy

Groupy lets you forget about the GroupMe API and focus on what you need to get done!

It is composed of two main parts:

- API wrapper (*groupy.api*)
- Object abstraction (*groupy.object*)

Current Features

Groups

- Create, update, and destroy your own groups
- List and filter your current and former groups
- Add and remove members from your current groups

- List and filter group members
- List and filter group messages

Members

- List and filter all known members
- List and filter direct messages
- Post direct messages to members

Messages

- Collect all messages from a group or member
- Like and unlike messages (even direct messages!)
- List and filter members who liked a message
- Inspect and create attachments

Bots

- List and filter your bots
- Use your bots to post messages
- Create, update, and destroy bots

Users

- Get your user information
- Enable and disable SMS mode

Planned Development

(in no particular order)

- Unit tests
- Installation via pip
- More direct way to add and remove yourself from groups
- Remove multiple members in one method call
- Porcelain for checking results of adding members
- Automatic updating of object attributes without the need to re-fetch objects
- Member objects that are aware of membership in all groups
- Additional ways to access objects
- More convenience methods instead of accessing API attributes directly
- Documentation about the API wrapper package

- Python 2.7 support

Installation

Prerequisites

To get started, you'll need to *get an account* at Groupme.com.

Got it? Great!

Now you'll need to *obtain your access token* so you can make API requests:

1. Login to the developer portal.
2. Click the "Access Token" button on the top menu bar.
3. Your access token is displayed in bold text.

You must also *create a key file*.

1. Paste your access token into a new file.
2. Save it as `.groupy.key` in your user's home directory.

Lastly, ensure you're running Python >= 3! Now you're ready to install Groupy!

Instructions

Below are instructions for various ways of performing installation.

Using `pip`

```
$ pip install GroupyAPI
```

From Source

Basic Steps

1. Download Groupy from GitHub.
2. Copy the `groupy` directory (`Groupy/groupy`) into your package directory for `Python3`.

Note: Your package directory may be elsewhere. For help determining the correct location, see this StackOverflow question.

With `git`

If you have `git`, it's as easy as:

```
$ git clone https://github.com/rhgrant10/Groupy.git
$ cd Groupy
$ cp -r groupy /usr/lib/python3/dist-packages       # see note above
```

Without `git`

If you don't have `git` installed, ask yourself why?

If you're satisfied with your answer to that question and you're still reading this section, fine. You don't *need* `git`. You can download it as a ZIP file.

- master branch
- dev branch

Installation is a simple matter of unzipping the file and copying over the `groupy` directory to your `Python3` package directory.

```
$ wget https://github.com/rhgrant10/Groupy/archive/master.zip
$ unzip master.zip
$ cd Groupy-master
$ cp -r groupy /usr/lib/python3/dist-packages    # see note above
```

Note: This is the least-recommended means of installing a python package!

For Development

So, you want to improve Groupy? Awesome! The easiest way to get started is by cloning the repository and then pip installing in development mode:

```
$ git clone git clone https://github.com/rhgrant10/Groupy.git
$ cd Groupy
$ pyvenv env
$ souce env/bin/activate
$ pip install -r requirements.txt && pip install -r testing_requirements.txt
$ pip install -e .
```

Now you're all set to start hacking on the code. You probably want to see how the existing tests are doing:

```
$ tox
```

Note: You do *not* need an API token to run tests.

Troubleshooting

Sometimes things go wrong. Here are some common things to check when encountering problems after installing.

It says no such package when I import groupy... Check whether you copied the `groupy` package into the correct python package directory. It must be a directory on your `sys.path`.

I get an unauthorized error when I try to do anything... Check whether your key file (`.groupy.key` by default) contains your API token, and that the value for `KEY_LOCATION` in `groupy.config` correctly specifies the location and name of your key file.

```
>>> import groupy
>>> groupy.config.KEY_LOCATION
'~/.groupy.key'
```

I get a weird error when installing Groupy... something about compiling Pillow... Make sure you've installed the developer packages for python. On debian systems:

```
$ sudo apt-get install python-dev python3.4-dev
```

Basic Usage

This page gives an overview of all but the most advanced features of **Groupy**.

First, you'll want to make sure that

- **Groupy** is *installed*
- **Groupy** can *find your API key*

See the *Installation* page for instructions. Now that that's out of the way, let's get started!

Listing Things

The most basic operation is listing things. `Group`s, `Member`s, and `Bot`s can be listed directly.

```
>>> import groupy
>>> groups = groupy.Group.list()
>>> members = groupy.Member.list()
>>> bots = groupy.Bot.list()
```

The object lists are returned as a `FilterList`. These behave just like the built-in `list` does with some convenient additions.

You can read more about the types of lists used by **Groupy** in the *Advanced Usage* section, but for the remainder of this page, the following truth should suffice.

```
>>> groups.first == groups[0]
True
>>> groups.last == groups[-1]
True
```

Groups

From a `Group`, you can list its `Member`s and `Message`s.

```
>>> from groupy import Group
>>> groups = Group.list()
>>> group = groups.first
>>> messages = group.messages()
>>> members = group.members()
```

A group returns all of its members in a single list. So determining the number of members in a group should be a familiar task.

```
>>> len(members)
5
```

*Message*s, however, are a different matter. Since there may be thousands of messages in a group, messages are returned in pages. The default (and maximum) number of messages per page is 100. To determine the total number of messages in a group, simply access the `message_count` attribute. Additional pages of messages can be obtained using *older()* and *newer()*.

```
>>> len(messages)
100
>>> group.message_count
3014
>>> older = messages.older()
>>> newer = messages.newer()
```

There are also methods for collecting a newer or older page of messages into one list: *iolder()* and *inewer()*. An example of using the former to retrieve all messages in a group:

```
>>> from groupy import Group
>>> group = Group.list().first
>>> messages = group.messages()
>>> while messages.iolder():
...       pass
...
>>> len(messages) == group.message_count
True
```

Often you'll want to post a new message to a group. New messages can be posted to a group using its *post()* method.

```
>>> from groupy import Group
>>> group = Group.list().first
>>> group.post('Hello to you')
>>> group.messages().newest.text
'Hello to you'
```

Note: Posting a message does not affect `message_count`. However, retrieving any page of messages *does* update it.

---

*Group*s have many attributes, some of which can be changed.

```
>>> group.name
'My Family'
>>> group.image_url
'http://i.groupme.com/123456789'
>>> group.description
'Group of my family members - so we can keep up with each other.'
>>> group.update(name="My Group of Family Members")
>>> group.name
'My Group of Family Members'
>>> group.update(name="[old] Family Group", description="The old family group")
>>> group.name
'[old] Family Group'
>>> group.description
'The old family group'
```

Some *Group*s also have a `share_url` that others can visit to join the group.

```
>>> group.share_url
'https://groupme.com/join_group/1234567890/SHARE_TOKEN'
```

Beware that not every group is created with a share link, in which case the value of `share_url` would be `None`. However, this can be changed in the same way as other group information.

```
>>> print(group.share_url)
None
>>> group.update(share=True)
>>> group.share_url
'https://groupme.com/join_group/1234567890/SHARE_TOKEN'
```

Note: The `SHARE_TOKEN` is specific to each group's share link.

The remainder of a *Group*s attributes cannot be changed. Some of the more important attributes are shown below.

```
>>> group.group_id
'1234567890'
>>> group.creator_user_id
'0123456789'
>>> print(group.created_at)
2013-12-25 9:53:33
>>> print(group.updated_at)
2013-12-26 4:21:08
```

Messages

Unlike *Group*s, *Member*s, and *Bot*s, *Message*s *cannot* be listed directly. Instead, *Message*s are listed either from *Group* or *Member* instances.

To list the messages from a group, use a group's *messages()* method.

```
>>> from groupy import Group
>>> group = Group.list().first
>>> messages = group.messages()
```

To list the messages from a member, use a member's *messages()* method.

```
>>> from groupy import Member
>>> member = Member.list().first
>>> messages = member.messages()
```

Messages have several properties. Let's look at a few of them. Messages have a timestamp indicating when the message was created as a `datetime.datetime` instance, as well as information about the member who posted it. Of course, messages can have text and attachments.

```
>>> message = messages.newest
>>> print(message.created_at)
2014-4-29 12:19:05
>>> message.user_id
'0123456789'
>>> message.name
'Kevin'
```

```
>>> message.avatar_url
'http://i.groupme.com/123456789'
>>> message.text
'Hello'
>>> message.attachments
[Image(url='http://i.groupme.com/123456789')]
```

Note: Not every message will have text and not every message will have attachments but every message must have one or the other.

Note: Although the majority of messages will have just one attachment, there is no limit on the number of attachments. In fact, despite that most clients are incapable of displaying more than one of each type of attachment, the API doesn't limit the types of attachments in any way. For example, a single message might have two images, three locations, and one emoji, but it's not likely that any client would show them all or handle the message without error.

There are multiple types of messages. System messages are messages that are not sent by a member, but generated by member actions. Many things generate system messages, including membership changes (entering/leaving, adding/removing), group updates (name, avatar, etc.), and member updates (nickname, avatar, etc.), and changing the topic.

Additionally there are group messages and direct messages. Group messages are messages in a group, whereas direct messages are messages between two members.

Each message has a few properties that can be used to differentiate among the types.

```
>>> message.group_id
'1234567890'
>>> message.recipient_id
None
>>> message.system
False
```

In the above example, we can see that `message.system` is `False`, which indicates that the message was sent by a member, not the system. We can also see that although the message has a `message.group_id`, it does *not* have a `message.recipient_id`, which means it is a group message. Had it been a system message, `message.system` would have been `True`. Had it been a direct message, `message.group_id` would have been `None` and `message.recipient_id` would contain a valid user ID.

Lastly, each message contains a list of user IDs to indicate which members have "liked" it.

```
>>> message.favorited_by
['2345678901', '3456789012']
```

Because often more information about the member is desired, a list of actual *Member* instances can be retrieved using the *likes()* method.

```
>>> message.likes()
[Rob, Jennifer, Vlad]
```

Messages can also be liked and unliked.

```
>>> message.like()
True
```

```
>>> message.unlike()
True
```

Note: Currently, the message instance itself does **not** update its own attributes. You must re-fetch the message.

Members

Member instances represent other GroupMe users. Finding members can be accomplished in one of three ways.

Firstly, members may be listed from a group. This lists just the members of a particular group.

```
>>> from groupy import Group
>>> group = Group.list().first
>>> members = group.members()
```

Secondly, members may be listed from a message. This lists just the members who have "liked" a particular message.

```
>>> messages = group.messages()
>>> message = message.newest
>>> members = message.likes()
```

Lastly, *all* the members you've seen thus far can be listed directly.

```
>>> from groupy import Member
>>> members = Member.list()
```

Note: Although many attributes of a member are specific to a particular group, members listed in this fashion are taken from a single group with one exception: the nickname of each member listed from *list()* is the most frequent of the names that the member uses among the groups of which you are both members.

Each member has a user ID, a nickname, and a URL indicating their avatar image that are specific to the group from which the member was listed.

```
>>> member = members.first
>>> member.user_id
'0123456789'
>>> member.nickname
'Bill'
>>> member.avatar_url
'http://i.groupme.com/123456789'
```

Members have one more property of interest: `muted`. This indicates whether the member has that group muted.

```
>>> member1, member2 = members[:2]
>>> member1.muted
False
>>> member2.muted
True
```

Messaging a member and retrieving the messages between you and the member is done in the same way as when messaging a group.

```
>>> member.post("Hello")
>>> member.messages().newest.text
'Hello'
```

Groups and Members

Members can be added and removed from groups. Adding one or multiple members to a group is quite intuitive. The following examples assume that no one from `group1` is a member of `group2` (although the API doesn't care if you add a member who is already a member).

```
>>> from groupy import Group
>>> group1, group2 = Group.list()[:2]
>>> member = group1.members().first
>>> group2.add(member)
```

Multiple members can be added simultaneously as well. Suppose you wanted to add everyone from `group1` to `group2`.

```
>>> group2.add(*group1.members())
```

Removing members, however, must be done one at a time:

```
>>> for m in group2.members():
...   group2.remove(m)
...
```

GroupMe and You

One of the most basic pieces of information you'll want to obtain is your own! **Groupy** makes this very simple:

```
>>> from groupy import User
>>> your_info = User.get()
```

It contains your GroupMe profile/account information and settings:

```
>>> print(your_info.user_id)
12345678
>>> print(your_info.name)
Billy Bob <-- the MAN!
>>> print(your_info.image_url)
http://i.groupme.com/123456789
>>> print(your_info.sms)
False
>>> print(your_info.phone_number)
+1 5055555555
>>> print(your_info.email)
bb@example.com
```

It also contains some meta information:

```
>>> print(your_info.created_at)
2011-3-14 14:11:12
>>> print(your_info.updated_at)
2013-4-20 6:58:26
```

`created_at` and `updated_at` are returned as `datetime` objects.

Bots

Bots can be a useful tool because each has a callback URL to which every message in the group is POSTed. This allows your bot the chance to do... well, something (whatever that may be) in response to every message!

Note: Keep in mind that bots can only post messages to groups, so if anything else is going to get done, it'll be done by you, not your bot. That means adding and removing users, liking messages, direct messaging a member, and creating or modifying group will be done under your name.

Bot creation is simple. You'll need to give the bot a name and associate it with a specific group.

```
>>> from groupy import Bot, Group
>>> group = Group.list().first
>>> bot = Bot.create('R2D2', group)
```

`bot` is now the newly created bot and is ready to be used. If you want, you can also specify a callback URL *(recommended)*, as well as an image URL to be used for the bot's avatar.

Just about the only thing a bot can do is post a message to a group. **Groupy** makes it easy:

```
>>> from group import Bot
>>> bot = Bot.list().first
>>> bot.post("I'm a bot!")
```

Note that the bot always posts its messages to the group in which it belongs. You can create multiple bots. Listing all of your bots is straightforward.

```
>>> from groupy import Bot
>>> bots = Bot.list()
```

Now `bots` contains a list of all of your bots.

Advanced Usage

This part of the documentation contains explanations and examples of more obscure aspects of **Groupy**.

Filter Lists

*`FilterList`*s are exactly like the built-in `list` but with some convenient additions.

`first` and `last`

`first` and `last` are merely convenience properties. `first` corresponds to the item at index `0`, while `last` corresponds to the item at index `-1`.

```
>>> from groupy.object.listers import FilterList
>>> fl = FilterList(range(1, 11))
>>> fl
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> fl.first
1
>>> fl.last
10
```

One important difference, however, is when there are no elements in the list.

```
>>> fl = FilterList()
>>> fl
[]
>>> print(fl.first)
None
>>> fl[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> print(fl.last)
None
>>> fl[-1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Note that no exception is raised and `None` is returned instead.

filter()

The *filter()* method parses its keyword arguments as filtering criteria. Only the items meeting all criteria are returned.

The keywords correspond to object properties, but also indicate how to test the relation to the value of the keyword argument. Thus a keyword-value pair such as `name='Bob'` would keep only those items with a `name` property equal to `"Bob"`, whereas a pair like `age__lt=20` keeps only those items with an `age` property *less than* `20`.

This is probably better explained with some simple examples.

```
>>> from groupy import Group
>>> groups = Group.list()
>>> for g in groups:
...     print(g.name)
...
My Family
DevTeam #6
Friday Night Trivia
>>> for g in groups.filter(name__contains='am'):
...     print(g.name)
My Family
DevTeam #6
>>>
>>> members = groups.first.members()
>>> for m in members:
...     print(m.nickname)
...
Dan the Man
Manuel
Fred
Dan
```

```
>>> for m in members.filter(nickname='Dan'):
...     print(m.nickname)
...
Dan
>>> for m in members.filter(nickname__contains='Dan'):
...     print(m.nickname)
...
Dan the Man
Dan
>>> for m in members.filter(nickname__ge='F'):
...     print(m.nickname)
...
Manuel
Fred
```

Attachments

Attachments are a common part of *Message*s and there are several different types. Currently, **Groupy** supports the following types of attachments:

- *Location* - for locations
- *Image* - for images
- *Mentions* - for "@" mentions
- *Emoji* - for emoticons
- *Split* - for splitting bills*[0]

For all other types of attachments (such as those introduced in the future) there exists a *GenericAttachment*.

Types

The section covers the various types of attachments and how to create them.

Locations

Location attachments are the simplest of all attachment types. Each includes a `name`, a latitude `lat`, and a longitude `lng`. Some location attachments also contain a `foursqure_venue_id`.

```
>>> from groupy import attachments
>>> loc = attachments.Location('My house', lat=34, lng=-84)
>>> loc
Location('My house', lat=34, lng=-84)
>>> loc.name
'My house'
>>> loc.lat, loc.lng
(34, -84)
```

[0] Split attachments are depreciated.

Images

Image attachments are unique in that they do not actually contain the image data. Instead, they specify the URL from which you can obtain the actual image. To create a new image from a local file object, use the *file()* method.

```
>>> from groupy import attachments
>>> image_attachment = attachments.Image.file(open(filename, 'rb'))
>>> image_attachment
Image(url='http://i.groupme.com/123456789')
>>> image_attachment.url
'http://i.groupme.com/123456789'
```

We can see that the image has been uploaded in exchange for a URL via the GroupMe image service.

To fetch the actual image from an image attachment, simply use its *download()* method. The image is returned as a `Pillow Image`, so saving it to a file is simple.

```
>>> image_file = image_attachment.download()
>>> image_file.save(filename)
```

Mentions

Mentions are a new type of attachment and have yet to be documented. However, they are simple to understand. Mentions capture the details necessary to highlight "@" mentions of members in groups. They contain a list of `loci` and an equal-sized list of `user_ids`. Let's find a good example to demonstrate mentions.

```
>>> from groupy import Group
>>> message = None
>>> mention = None
>>> for g in Group.list():
...   for m in g.messages():
...     for a in m.attachments:
...       if a.type == 'mentions' and len(a.user_ids) > 1:
...         message = m
...         mention = a
...         break
>>> message.text
'@Bill hey I saw you with @Zoe Childs at the park!'
>>> mention.user_ids
['1234567', '5671234']
>>> mention.loci
[[0, 5], [25, 11]]
```

As you can see, each element in `loci` has two integers, the first of which indicates the starting index of the mentioning text, while second indicates its length. The strings in `user_ids` correspond *by index* to the elements in `loci`. You can use the `loci` to extract the mentioning portion of the text, as well as obtain the mentioned member via `user_ids`.

```
>>> for uid, (start, length) in zip(mention.user_ids, mention.loci):
...   end = start + length
...   uid, message.text[start:end]
...   member = message.group.members().filter(user_id=uid).first
...   member.uid, member.nickname
('1234567', '@Bill')
('1234567', 'Bill')
```

```
('5671234', '@Zoe Childs')
('5671234', 'Zoe Childs')
```

To create a mention, simply pass in a `list` of user IDs and an equally-sized `list` of loci.

```
>>> from groupy.attachments import Mentions
>>> Mentions(['1234567', '2345671'], [[0, 4], [5, 3]])
Mentions(['1234567', '2345671'])
```

Emojis

Emojis are relatively undocumented but frequently appear in messages. More documentation will come as more is learned.

Emoji attachments have a `placeholder` and a `charmap`. The `placeholder` is a high-point or unicode character designed to mark the location of the emoji in the text of the message. The `charmap` serves as some sort of translation or lookup tool for obtaining the actual emoji.

Splits

Note: This type of attachment is depreciated. They were part of GroupMe's bill splitting feature that seems to no longer be implemented in their clients. **Groupy**, however, still supports them due to their presence in older messages.

`Split` attachments have a single attribute: `token`.

Sending Attachments

To send an attachment along with a message, simply append it to the *`post()`* method as another argument.

```
>>> from groupy import Group
>>> from groupy.attachment import Location
>>> loc = Location.create('My house', lat=33, lng=-84)
>>> group = Group.list().first
>>> group.post("Hey meet me here", loc)
```

If there are several attachments you'd like to send in a single message, simply keep appending them!

```
>>> from groupy.attachment import Image
>>> img = Image.file('front-door.png')
>>> group.post("I said meet me here!", loc, img)
```

Alternatively, you can collect multiple attachments into an `iterable`.

```
>>> attachments = [img, loc]
>>> group.post("Are you listening?", *attachments)
```

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/rhgrant10/Groupy/issues.

If you are reporting a bug, please include:

- Your python version
- Your groupy version
- 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

Groupy could always use more documentation, whether as part of the official Groupy 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/rhgrant10/Groupy/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 *Groupy* for local development.

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

```
$ git clone git@github.com:YOUR_NAME_HERE/Groupy.git
```

3. Install your local copy into a virtualenv. Since 3.3, Python ships with its own virutal environment creator: *venv*. Usage is simple:

```
$ cd Groupy/
$ pyvenv env
$ source env/bin/activate
$ pip install -r requirements.txt && pip install testing_requirements.txt
```

4. Create a branch *from the dev branch* for local development:

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

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8, have great coverage, and pass all tests on all supported versions of python. Sounds tough, but *tox* makes this easy:

```
$ tox
```

Note that if you update `requirements.txt` or `testing_requirements.txt` you must tell tox to recreate its virtual environments using the `-r` flag:

```
$ tox -r
```

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

```
$ git add .
$ git commit
$ git push origin name-of-your-bugfix-or-feature
```

Please make sure to:

- *not* to commit sensitive data or extra files. You can use `git add -p` to add parts of files if necessary.
- follow proper git commit message standards. In particular, the first line should be under 60 characters long, and any detail should begin on the 3rd line (the second line must be blank).

7. 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 3.4 and 3.5. Check https://travis-ci.org/rhgrant10/Groupy/pull_requests and make sure that the tests pass for all supported Python versions.

Developer Docs

This section of the documentation is for other developers, and contains the complete information about each package, module, class, and method.

The api Package

This module is a direct wrapper around GroupMe API calls.

The api.endpoint Module

This module contains classes that represent the many endpoints in the GroupMe API.

class groupy.api.endpoint.**Bots**
Endpoint for the bots API.

Bots can be listed, created, updated, and destroyed. Bots can also post messages to groups.

classmethod create(*name*, *group_id*, *avatar_url=None*, *callback_url=None*)
Create a new bot.

Parameters

- **name** (*str*) – the name of the bot
- **group_id** (*str*) – the ID of the group to which the bot will belong
- **avatar_url** (*str*) – the GroupMe image URL for the bot's avatar
- **callback_url** (*str*) – the callback URL for the bot

Returns the new bot

Return type dict

classmethod destroy(*bot_id*)
Destroy a bot.

Parameters **bot_id** (*str*) – the ID of the bot to destroy

classmethod index()
List bots.

Returns a list of bots

Return type list

classmethod post(*bot_id*, *text*, **attachments*, *picture_url=None*)
Post a message to a group as a bot.

Parameters

- **bot_id** (*str*) – the ID of the bot
- **text** (*str*) – the message text
- **picture_url** (*str*) – the GroupMe image URL for a picture
- **attachments** (list) – a list of attachments to include

Returns the created message

Return type dict

class groupy.api.endpoint.**DirectMessages**
Endpoint for the direct message API.

classmethod create(*recipient_id*, *text*, **attachments*)
Create a direct message to a recipient user.

Parameters

- **recipient_id** (*str*) – the ID of the recipient
- **text** (*str*) – the message text
- **attachments** (list) – a list of attachments to include

Returns the created direct message

Return type dict

classmethod index(*other_user_id*, *before_id=None*, *since_id=None*, *after_id=None*)
List the direct messages with another user.

Parameters

- **other_user_id** (*str*) – the ID of the other party
- **before_id** (*str*) – a reference message ID; specify this to list messages prior to it

Returns a list of direct messages

Return type list

class groupy.api.endpoint.**Endpoint**
An API endpoint capable of building a url and extracting data from the response.

This class serves as the base class for all of the API endpoints.

classmethod build_url(*path=None*, **args*)
Build and return a url extended by *path* and filled in with *args*.

Parameters

- **path** (*str*) – a suffix for the final URL. If *args* are present, this should be a python format string pertaining to the given *args*.
- **args** (list) – a list of arguments for the format string *path*.

Returns a complete URL

Return type str

static clamp(*value*, *lower*, *upper*)
Utility method for clamping a *value* between a *lower* and an *upper* value.

Parameters

- **value** – the value to clamp
- **lower** – the "smallest" possible value
- **upper** – the "largest" possible value

Returns *value* such that `lower <= value <= upper`

classmethod response(*r*)
Extract the data from the API response *r*.

This method essentially strips the actual response of the envelope while raising an ApiError if it contains one or more errors.

Parameters **r** (requests.Response) – the HTTP response from an API call

Returns API response data

Return type json

class groupy.api.endpoint.**Groups**
Endpoint for the groups API.

Groups can be listed, loaded, created, updated, and destroyed.

classmethod create(*name*, *description=None*, *image_url=None*, *share=True*)
Create a new group.

Parameters

- **name** (*str*) – the name of the new group
- **description** (*str*) – the description of the new group
- **image_url** (*str*) – the group avatar image as a GroupMe image URL
- **share** (*bool*) – whether to generate a join link for the group

Returns the new group

Return type dict

classmethod destroy(*group_id*)
Destroy (or leave) a group.

Note: If you are not the owner of a group, you cannot destroy it.

Parameters **group_id** (*str*) – the ID of the group to destroy/leave

Return type dict

classmethod index(*page=1*, *per_page=500*, *former=False*)
Return a list of groups.

Parameters

- **page** (*int*) – the page of groups to return
- **per_page** (*int*) – the number of groups in the page
- **former** (*bool*) – whether to list former groups instead

Returns a list of groups

Return type list

classmethod show(*group_id*)
Return a specific group by its *group_id*.

Parameters **group_id** (*str*) – the ID of the group to show.

Returns the group with the given *group_id*

Return type dict

classmethod update(*group_id*, *name=None*, *description=None*, *share=None*, *image_url=None*)
Update the information for a group.

Parameters

- **group_id** (*str*) – the ID of the group to update
- **name** (*str*) – the new name of the group
- **description** (*str*) – the new description of the group

- **share** (*bool*) – whether to generate a join link for the group
- **image_url** (*str*) – the GroupMe image URL for the new group avatar.

Returns the modified group

Return type dict

class groupy.api.endpoint.**Images**
Endpoint for the image service API.

GroupMe images are created through an upload service that returns a URL at which it can be accessed.

classmethod create(*image*)
Submit a new image.

Parameters **image** (file) – object with a file-like interface and containing an image

Returns the URL at which the image can be accessed

Return type dict

classmethod response(*r*)
Extract the data from the image service API response *r*.

This method basically returns the inner "payload."

Parameters **r** (requests.Response) – the HTTP response from an API call

Returns API response data

Return type json

class groupy.api.endpoint.**Likes**
Endpoint for the likes API.

Likes can be created or destroyed.

Note: The conversation_id is poorly documented. For messages in a group, it corresponds to the group_id (or id since they seem to always be identical). For direct messages, it corresponds to the user_id of both conversation participants sorted lexicographically and concatenated with a plus sign ("+").

---

classmethod create(*conversation_id*, *message_id*)
Like a message.

Parameters

- **conversation_id** (*str*) – the ID of the group or recipient
- **message_id** (*str*) – the ID of the message

classmethod destroy(*conversation_id*, *message_id*)
Unlike a message.

Parameters

- **conversation_id** (*str*) – the ID of the group or recipient
- **message_id** (*str*) – the ID of the message

class groupy.api.endpoint.**Members**
Endpoint for the members API.

Members can be added and removed from a group, and the results of adding members can be obtained.

classmethod **add**(*group_id*, **members*)
Add one or more members to a group.

Parameters

- **group_id** (*str*) – the ID of the group to which the members should be added
- **members** (list) – the members to add.

Returns the results ID for this request

Return type dict

classmethod **remove**(*group_id*, *member_id*)
Remove a member from a group.

Parameters

- **group_id** (*str*) – the ID of the group from which the member should be removed
- **member_id** (*str*) – the ID of the member to remove

classmethod **results**(*group_id*, *result_id*)
Check the result of adding one or more members.

Parameters

- **group_id** (*str*) – the ID of the group to which the add call was made
- **result_id** (*str*) – the GUID returned by the add call

Returns the successfully added members

Return type list

class groupy.api.endpoint.**Messages**
Endpoint for the messages API.

Messages can be listed and created.

classmethod **create**(*group_id*, *text*, **attachments*)
Create a new message in a group.

All messages must have either text or one attachment. Note that while the API provides for an unlimited number of attachments, most clients can only handle one of each attachment type (location, image, split, or emoji).

Parameters

- **group_id** (*str*) – the ID of the group in which to create the message
- **text** (*str*) – the text of the message
- **attachments** (list) – a list of attachments to include

Returns the created message

Return type dict

classmethod **index**(*group_id*, *before_id=None*, *since_id=None*, *after_id=None*, *limit=100*)
List the messages from a group.

Listing messages gives the most recent 100 by default. Additional messages can be obtained by specifying a reference message, thereby facilitating paging through messages.

Use before_id and after_id to "page" through messages. since_id is odd in that it returns the *most recent* messages since the reference message, which means there may be messages missing between the reference message and the oldest message in the returned list of messages.

Note: Only one of `before_id`, `after_id`, or `since_id` can be specified in a single call.

Parameters

- **group_id** (*str*) – the ID of the group from which to list messages
- **before_id** (*str*) – a reference message ID; specify this to list messages just prior to it
- **since_id** (*str*) – a reference message ID; specify this to list the *most recent* messages after it (**not** the messages right after the reference message)
- **after_id** (*str*) – a reference message ID; specifying this will return the messages just after the reference message
- **limit** (*int*) – a limit on the number of messages returned (between 1 and 100 inclusive)

Returns a `dict` containing `count` and `messages`

Return type `dict`

Raises **ValueError** – if more than one of `before_id`, `after_id` or `since_id` are specified

class groupy.api.endpoint.**Sms**
Endpoint for the SMS API.

SMS mode can be enabled or disabled.

classmethod create(*duration=4*, *registration_id=None*)
Enable SMS mode.

Parameters

- **duration** (*int*) – duration of SMS mode in hours (max of 48)
- **registration_id** (*str*) – the push registration_id or token to suppress (if omitted, SMS and push notifications will both be enabled)

classmethod delete()
Disable SMS mode.

class groupy.api.endpoint.**Users**
Endpoint for the users API.

classmethod me()
Get the user's information.

Returns the user's information

Return type `dict`

The api.errors Module

The `error` module contains all of the exceptions thrown by the GroupMe API.

exception groupy.api.errors.**ApiError**
Error raised when errors are returned in a GroupMe response.

exception groupy.api.errors.**GroupMeError**
A general GroupMe error.

All exceptions raised by Groupy are descendents of this exception.

exception groupy.api.errors.**InvalidOperatorError**
Error thrown when an unsupported filter is used.

The api.status Module

The status module contains API response status code constants and a method that returns the textual description of such a constant.

groupy.api.status.**OK = 200**
Success

groupy.api.status.**CREATED = 201**
Resource was created successfully

groupy.api.status.**NO_CONTENT = 204**
Resource was deleted successfully

groupy.api.status.**NOT_MODIFIED = 304**
There was no new data to return

groupy.api.status.**BAD_REQUEST = 400**
Invalid format or invalid data is specified in the request

groupy.api.status.**UNAUTHORIZED = 401**
Authentication credentials were missing or incorrect

groupy.api.status.**FORBIDDEN = 403**
The request was understood, but it has been refused

groupy.api.status.**NOT_FOUND = 404**
The URI requested is invalid or the requested resource does not exist

groupy.api.status.**ENHANCE_YOUR_CLAIM = 420**
You are being rate limited

groupy.api.status.**INTERNAL_SERVER_ERROR = 500**
Something unexpected occurred

groupy.api.status.**BAD_GATEWAY = 502**
GroupMe is down or being upgraded

groupy.api.status.**SERVICE_UNAVAILABLE = 503**
The GroupMe servers are up but overloaded with requests

groupy.api.status.**description**(*code*)
Return the text description for a code.

Parameters **code** (*int*) – the HTTP status code

Returns the text description for the status code

Return type str

The object Package

This module abstracts the objects returned by GroupMe API calls.

The `object.responses` Module

This module contains classes that encapsulate the information returned in API responses.

class `groupy.object.responses.`**`Recipient`**(*endpoint*, *mkey*, *idkey*, ***kwargs*)
Base class for Group and Member.

Recipients can post and receive messages.

Parameters

- **`endpoint`** (*`Endpoint`*) – the API endpoint for messages
- **`mkey`** (*`str`*) – the `dict` key under which the endpoint returns messages
- **`idkey`** (*`str`*) – the `dict` key whose value represents the key for posting and retrieving messages

`messages`(*before=None*, *since=None*, *after=None*, *limit=None*)
Return a page of messages from the recipient.

Note: Only one of `before`, `after`, or `since` can be specified in a single call.

Parameters

- **`before`** (*`str`*) – a reference message ID
- **`since`** (*`str`*) – a reference message ID
- **`after`** (*`str`*) – a reference message ID
- **`limit`** (*`int`*) – maximum number of messages to include in the page

Returns a page of messages

Return type *`MessagePager`*

Raises **`ValueError`** – if more than one of `before`, `after` or `since` are specified

`post`(*text*, **attachments*)
Post a message to the recipient.

Although the API limits messages to 1000 characters, this method will split the text component into as many as necessary and include the attachments in the final message. Note that a list of messages sent is always returned, even if it contains only one element.

Parameters

- **`text`** (*`str`*) – the message text
- **`attachments`** (`list`) – the attachments to include

Returns a list of raw API responses (sorry!)

Return type `list`

class `groupy.object.responses.`**`Group`**(***kwargs*)
A GroupMe group.

`add`(**members*, *refresh=False*)
Add a member to a group.

Each member can be either an instance of *`Member`* or a `dict` containing `nickname` and one of `email`, `phone_number`, or `user_id`.

Parameters

- **members** (*list*) – members to add to the group
- **refresh** (*bool*) – `True` if the group information should be automatically refreshed from the API, `False` by default

Returns the results ID of the add call

Return type str

classmethod create(*name*, *description=None*, *image_url=None*, *share=True*)
Create a new group.

Parameters

- **name** (*str*) – the group name
- **description** (*str*) – the group description
- **image_url** (*str*) – the GroupMe image service URL for a group avatar
- **share** (*bool*) – whether to generate a join link

Returns the newly created group

Return type *Group*

destroy()
Disband (destroy) a group that you created.

If unsuccessful, this raises an *ApiError*

Returns *OK*

classmethod list(*former=False*)
List all of your current or former groups.

Parameters **former** (*bool*) – `True` if former groups should be listed, `False` (default) lists current groups

Returns a list of groups

Return type *FilterList*

members()
Return a list of the members in the group.

Returns the members of the group

Return type *FilterList*

refresh()
Refresh the group information from the API.

remove(*member*, *refresh=False*)
Remove a member from the group.

Note: The group must contain the member to be removed. This will *not* be the case if the group information has not been requested since the member was *added*. When in doubt, use the *refresh()* method to update the internal list of members before attempting to remove them.

Parameters

- **member** (*Member*) – the member to remove

- **refresh** (*bool*) – `True` if the group information should be automatically refreshed from the API, `False` by default

Returns `True` if successful

Return type bool

Raises ***groupy.api.errors.ApiError*** – if removal is not successful

update(*name=None*, *description=None*, *image_url=None*, *share=None*)
Change group information.

Parameters

- **name** (*str*) – the new name of the group
- **description** (*str*) – the new description of the group
- **image_url** (*str*) – the URL for the new group image
- **share** (*bool*) – whether to generate a share URL

class groupy.object.responses.**Member**(***kwargs*)
A GroupMe member.

identification()
Return the identification of the member.

A member is identified by their `nickname` and `user_id` properties. If the member does not yet have a GUID, a new one is created and assigned to them (and is returned alongside the `nickname` and `user_id` properties).

Returns the `nickname`, `user_id`, and `guid` of the member

Return type `dict`

classmethod identify(*member*)
Return or create an identification for a member.

Member identification is required for adding them to groups. If member is a `dict`, it must contain the following keys:

- `nickname`
- `user_id` or `email` or `phone_number`

If an identification cannot be created then raise an `ValueError`.

Parameters **member** – either a *Member* or a `dict` with the required keys

Returns the identification of member

Return type `dict`

Raises **ValueError** – if an identification cannot be made

classmethod list()
List all known members regardless of group membership.

Returns a list of all known members

Return type `FilterList`

class groupy.object.responses.**Message**(*recipient*, ***kwargs*)
A GroupMe message.

Parameters **recipient** (*Recipient*) – the reciever of the message

is_from_me()
Return `True` if the message was sent by you.

Return type bool

is_liked_by_me()
Return `True` if the message was liked by you.

Return type bool

like()
Like the message.

Returns `True` if successful

Return type bool

Raises ***groupy.api.errors.ApiError*** – if unsuccessful

likes()
Return a *FilterList* of the members that like the message.

Returns a list of the members who "liked" this message

Return type *FilterList*

mentions_me()
Return `True` if the message "@" mentions you.

Return type bool

recipient
Return the source of the message.

If the message is a direct message, this returns a member. Otherwise, it returns a group.

Returns the source of the message

Return type *Recipient*

unlike()
Unlike the message.

Returns `True` if successful

Return type bool

Raises ***groupy.api.errors.ApiError*** – if unsuccessful

class `groupy.object.responses.`**Bot**(***kwargs*)
A GroupMe bot.

Each bot belongs to a single group. Messages posted by the bot are always posted to the group to which the bot belongs.

classmethod create(*name*, *group*, *avatar_url=None*, *callback_url=None*)
Create a new bot.

Parameters

- **name** (*str*) – the name of the bot
- **group** (*Group*) – the group to which the bot will belong
- **avatar_url** (*str*) – the URL for a GroupMe image to be used as the bot's avatar
- **callback_url** (*str*) – the URL to which each group message will be POSTed

Returns the new bot

Return type *Bot*

destroy()
Destroy the bot.

Returns `True` if successful

Return type bool

Raises ***groupy.api.errors.ApiError*** – if unsuccessful

classmethod list()
Return a list of your bots.

Returns a list of your bots

Return type *FilterList*

post(*text*, **attachments*, *picture_url=None*)
Post a message to the group of the bot.

Parameters

- **text** (*str*) – the message text
- **picture_url** (*str*) – the GroupMe image URL for an image
- **attachments** (`list`) – the attachments to include

Returns `True` if successful

Return type bool

Raises ***groupy.api.errors.ApiError*** – if unsuccessful

class groupy.object.responses.**User**(***kwargs*)
A GroupMe user.

This is you, as determined by your API key.

classmethod disable_sms()
Disable SMS mode.

Disabling SMS mode causes push notifications to resume and SMS text messages to be discontinued.

Returns `True` if successful

Return type `bool`

Raises ***groupy.api.errors.ApiError*** – if unsuccessful

classmethod enable_sms(*duration=4*, *registration_token=None*)
Enable SMS mode.

Each client has a unique registration token that allows it to recieve push notifications. Enabling SMS mode causes GroupMe to suppress those push notification and send SMS text messages instead for a number of hours no greater than 48.

Note: If the `registration_token` is omitted, no push notifications will be suppressed and the user will recieve *both* text messages *and* push notifications.

Parameters

- **duration** (*int*) – the number of hours for which to send text messages
- **registration_token** (*str*) – the push notification token for which messages should be suppressed

Returns `True` if successful

Return type `bool`

Raises ***groupy.api.errors.ApiError*** – if unsuccessful

classmethod get()
Return your user information.

Returns your user information

Return type `dict`

nickname
Your user name.

The `object.attachments` Module

This module contains classes for the different types of attachments.

class groupy.object.attachments.**Attachment**(*type_*)
Base class for attachments.

Parameters **type** (*str*) – the type of the attachment

as_dict()
Return the attachment as a dictionary.

Returns the attachment as a dictionary

Return type `dict`

class groupy.object.attachments.**AttachmentFactory**
A factory for creating attachments from dictionaries.

classmethod create(***kwargs*)
Create and return an attachment.

Parameters **type** (*str*) – the type of attachment to create; if unrecognized, a generic attachment is returned

Returns a subclass of *Attachment*

class groupy.object.attachments.**Emoji**(*placeholder*, *charmap*)
An attachment containing emoticons.

Emoji attachments do not contain any emoticon images. Instead, a placeholder specifies the location of the emoticon in the text, and a `charmap` facilitates translation into the emoticons.

Parameters

- **placeholder** (*str*) – a high-point/invisible character indicating the position of the emoticon
- **charmap** (`list`) – a list of lists containing pack IDs and offsets

class groupy.object.attachments.**GenericAttachment**(*type*, ***kwargs*)
A generic attachment.

This attachment accepts any keyword arguments, but must be given a particular type.

Parameters **type** (*str*) – the type of attachment

class groupy.object.attachments.**Image**(*url*, *source_url=None*)
An image attachemnt.

Image attachments do not contain an image. Instead, they specify a URL from which the image can be downloaded and must have a domain of "i.groupme.com". Such URLs are known as "i" URLs, and are from the GroupMe image service.

Note: Use the direct initializer *if and only if* the image already has a known GroupMe image service URL. Otherwise, use the *file()* method.

Parameters

- **url** (*str*) – the URL at which the image can be fetched from the GroupMe image service
- **source_url** (*str*) – the original URL of the image (optional)

download()
Download the image data of the image attachment.

Returns the actual image the image attachment references

Return type PIL.Image.Image

classmethod file(*image*)
Upload an image file and return it as an attachment.

Parameters **image** (file) – the file containing the image data

Returns an image attachment

Return type *Image*

class groupy.object.attachments.**Location**(*name*, *lat*, *lng*, *foursquare_venue_id=None*)
An attachment that specifies a geo-location.

In addition to latitude and longitude, every location attachment also specifies a name. Some (especially older) location attachments also contain a foursquare_venue_id attribute.

Parameters

- **name** (*str*) – the location name
- **lat** (*float*) – the latitude
- **lng** (*float*) – the longitude
- **foursquare_venue_id** (*str*) – the FourSquare venue ID (optional)

class groupy.object.attachments.**Mentions**(*user_ids*, *loci=None*)
An attachment that specifies "@" mentions.

Mentions are a new addition to the types of attachments. Each contains two parallel lists: user_ids and loci. The elements in loci specify the start index and length of the mention, while the elements in user_ids specify by user_id which user was mentioned in the corresponding element of loci.

Note: The length of user_ids must be equal to the length of loci!

Parameters

- **user_ids** (list) – a list of user IDs
- **loci** (list) – a list of (start, length) elements

class groupy.object.attachments.**Split**(*token*)
An attachment containing information for splitting a bill.

This type of attachment is depreciated. However, such attachments are still present in older messages.

Parameters **token** (*str*) – the token that splits the bill

The object.listers Module

This module contains classes that provide filterable lists and message pagers.

class groupy.object.listers.**FilterList**
A filterable list.

Acts just like a regular list, except it can be filtered using a special keyword syntax. Also, the first and last items are special properties.

filter(***kwargs*)
Filter the list and return a new instance.

Arguments are keyword arguments only, and can be appended with operator method names to indicate relationships other than equals. For example, to filter the list down to only items whose name property contains "ie":

```
new_list = filter_list.filter(name__contains='ie')
```

As another example, this filters the list down to only those with a created property that is less than 1234567890:

```
new_list = filter_list.filter(created__lt=1234567890)
```

Acceptable operators are:

- __lt: less than
- __gt: greater than
- __contains: contains
- __eq: equal to
- __ne: not equal to
- __le: less than or equal to
- __ge: greater than or equal to

Use of any operator listed here results in a *InvalidOperatorError*.

Returns a new list with potentially less items than the original

Return type *FilterList*

first
The first element in the list.

last
The last element in the list.

class groupy.object.listers.**MessagePager**(*group*, *messages*, *backward=False*)
A filterable, extendable page of messages.

Parameters

- **group** (*Group*) – the group from which to page through messages
- **messages** (list) – the initial page of messages
- **backward** (*bool*) – whether the oldest message is at index 0

inewer()
Add in-place the next (newer) page of messages.

Returns True if successful, False otherwise

Return type bool

iolder()
Add in-place the previous (older) page of messages.

Returns True if successful, False otherwise

Return type bool

newer()
Return the next (newer) page of messages.

Returns a newer page of messages

Return type *MessagePager*

newest
Return the newest message in the list.

Returns the newest message in the list

Return type *Message*

older()
Return the previous (older) page of messages.

Returns an older page of messages

Return type *MessagePager*

oldest
Return the oldest message in the list.

Returns the oldest message in the list

Return type *Message*

prepend(*messages*)
Prepend a list of messages to the list.

Parameters **messages** (list) – the messages to prepend

The config Module

The config module contains all the configuration options.

groupy.config.**API_URL = 'https://api.groupme.com/v3'**
The URL for the GroupMe API

`groupy.config.`**`IMAGE_API_URL`** **= 'https://image.groupme.com'**
The URL for the GroupMe Image Service API

`groupy.config.`**`KEY_LOCATION`** **= '~/.groupy.key'**
Full path to the file in which your access token can be found

Change Log

v0.7.1 (March 29, 2017)

- Fixed an issue serializing attachments posted by the user (not a bot) (thanks to a-vilmin for reporting the issue)

v0.7.0 (November 19, 2016)

- Fixed an issue with attachments not being serializable. Now an attempt to call their `as_dict` method is made (thank you to awctomlinson for pointing it out)
- Fixed problem with `is_liked_by_me`, `is_from_me` and `metions_me` when used on `DirectMessages` (thank you to mmigrate)
- Added attachment support to `Bot`'s `post` method (thank you again to mmigrate)
- Fixed a mispelling in the `mentions_me` method name (thank you adelq)

v0.6.6 (April 23, 2016)

- Fixed a typo in the docs regarding the type of the `group` parameter of the `Bot` class (kudos to JCDJulian)
- Fixed the `Group.update` method signature to include the `group_id` (kudos to mmirate)
- Fixed `Member.identification` such that it uses `Member.guid` rather than `Member._guid` (kudos to mmirate)
- Fixed the uncaught exception chain that occurred when a 304 was returned in `Recipient.messages` (thanks to dvmorris and sbonds for pointing it out)
- Updated the list of contributors

v0.6.5 (January 17, 2016)

- Fixed typo the `Bot` class that caused the bots to have a "gorup_id" (kudos to JCDJulian)
- All modules except `object/listers.py` and `object/responses.py` now have full test coverage
- Updated AUTHORS.rst with all contributors to date (feel free to PR with an email address added to your username)
- Fixed leftover markdown formatting in the CHANGELOG.rst file

v0.6.4 (December 31, 2015)

- Fixed bugs with creating bots (kudos to qlyoung)
- Fixed bugs with posting messages as bots (kudos again to qlyoung)

- Fixed typo bugs in `Group` class (kudos to t3zla)
- Fixed missing Python 3 trove classifier
- Added documentation for contributions
- Updated documentation for setup and installation
- Added a couple more unit tests
- Reconfigured tox test results to not clobber results from other environments

v0.6.3 (December 23, 2015)

- Added support for `tox` (envs py34,py35)
- Added support for `bumpversion`
- Added `make` file for handy development
- Moved to `nosetests` and `coverage`
- Split requirements into regular and testing
- Updated some of the installation/troubleshooting docs
- Merged in open pull-requests for various oversights (kudos to ScufyfNrdHrdr, rAntonioH, and JacobAMason)

v0.6.2 (May 3, 2015)

- Fixed problem when posting messages as a bot
- Added `refresh` option for automatically updating group information after addition/removal of members
- Updated documentation

v0.6.1 (April 25, 2015)

- Fixed code in `responses.py` that was still using the old exception class name
- Changed the `Member.remove()` method to correctly use the `id` of the member rather than the `user_id`
- Slight beefing up of some documentation

v0.5.8 (December 9, 2014)

- Fixed problems with `requirements.txt` and `setup.py` that caused problems installing from `pip`
- Re-wrote many of the unittests (still in progress)
- Added Travis-CI and PyPI badges to the readme
- Bumped requirement for dropbox's `responses` to 0.3.0
- Now uses `setup` from `setuptools` rather than `distutils.core`

v0.5.3 (September 19, 2014)

- Fix packaging bug that caused inner packages to not be installed via `pip3`

v0.5.2 (September 14, 2014)

- Now installable via `pip3`:

```
$ pip3 install GroupyAPI
```

v0.5.1 (August 25, 2014)

Groups

- Added a class method for creating a new group
- Added an instance method for destroying a group

Members

- Fixed member identification on dictionaries

User

- Fixed the enable/disable SMS methods (now class methods as they should be)

Documentation

- Added some module docstrings
- Added API docs for all attachment classes
- Added docs for split attachments
- Moved FilterList docs into the Advanced Usage section
- Rewrote API docs for enabling SMS mode
- Fixed bad sphinx references
- Fixed typos
- Added miscellaneous sections to the README
- Updated feature list

v0.5.0 (August 20, 2014)

- Added support for downloaded the image of an image attachment
- Reorganized modules and project structure
- Updated documentation

v0.4.0 (August 18, 2014)

- Added ability to list all known members
- Re-wrote attachments classes

v0.3.1 (August 14, 2014)

- Fixed bug when adding members to a group
- Many additions to the documentation

v0.3.0 (August 12, 2014)

- Added post and messages methods to members
- Added after_id parameter for direct messages
- Fixed liking and unliking direct messages
- Fixed listing former groups
- Fixed group lists being limited to a max of 500 items
- Documentation now available on Read the Docs!

v0.2.0 (August 11, 2014)

- Added MessagePager class for returning lists of messages

v0.1.3 (August 10, 2014)

- Added attachment class
- Added basic documentation
- Fixed the automatic splitting of long texts
- Fixed invalid response error issue

v0.1.0 (August 9, 2014)

- Initial release

Python Module Index

Index

I

K

L

M

N

O

P

R

S

U