
tg*bottingDocumentation*

Release latest

Jul 18, 2023

CONTENTS

1	Documentation Contents	3
1.1	Introduction	3
1.2	Quickstart	4
1.3	Commands	6
1.4	Cogs	8
1.5	API Reference	9
2	Indices and tables	17
	Index	19

tg-botting is a basic package to build async Telegram chatbots

DOCUMENTATION CONTENTS

1.1 Introduction

This is the documentation for tg-botting, a library for Python to aid in creating applications that utilise the Telegram Bot API.

1.1.1 Prerequisites

tg-botting works with Python 3.6.0 or higher. Support for earlier versions of Python is not provided. Python 2.7 or lower is not supported.

1.1.2 Installing

You can get the library directly from PyPI:

```
python3 -m pip install -U tg-botting
```

If you are using Windows, then the following should be used instead:

```
py -3 -m pip install -U tg-botting
```

Virtual Environments

Sometimes you want to keep libraries from polluting system installs or use a different version of libraries than the ones installed on the system. You might also not have permissions to install libraries system-wide. For this purpose, the standard library as of Python 3.3 comes with a concept called “Virtual Environment”s to help maintain these separate versions.

A more in-depth tutorial is found on [Virtual Environments and Packages](#).

However, for the quick and dirty:

1. Go to your project’s working directory:

```
$ cd your-bot-source  
$ python3 -m venv bot-env
```

2. Activate the virtual environment:

```
$ source bot-env/bin/activate
```

On Windows you activate it with:

```
$ bot-env\Scripts\activate.bat
```

3. Use pip like usual:

```
$ pip install -U tg-botting
```

Congratulations. You now have a virtual environment all set up.

1.1.3 Basic Concepts

vk-botting revolves around the concept of *events*. An event is something you listen to and then respond to. For example, when a message happens, you will receive an event about it that you can respond to.

for find user_id and user_hash:

Visit <https://my.telegram.org/apps> and log in with your Telegram account.

Fill out the form with your details and register a new Telegram application.

Done. The API key consists of two parts: user_id and user_hash. Keep it secret.

A quick example to showcase how events work:

```
from tg_botting.bot import Bot

bot = Bot(['your', 'prefixs'], user_id, user_hash)

@bot.listener()
async def on_message_new(message):
    print(message.text)

@bot.listener()
async def on_start():
    print('start')

bot.run(bot-token)
```

1.2 Quickstart

This page gives a brief introduction to the library. It assumes you have the library installed, if you don't check the *Installing* portion.

1.2.1 A Minimal Bot

Let's make a bot that replies to a specific message and walk you through it.

It looks something like this:

```
from tg_botting.bot import Bot

bot = Bot(['your', 'prefixs'], user_id, user_hash)

@bot.listener()
async def on_message_new(message):
    if message.text.startswith('Hello'):
        await message.send("Hello!")

@bot.listener()
async def on_start():
    print('start')

bot.run(bot-token)
```

Let's name this file `example_bot.py`.

There's a lot going on here, so let's walk you through it step by step.

1. The first line just imports the library, if this raises a *ModuleNotFoundError* or *ImportError* then head on over to [Installing](#) section to properly install.
2. Next, we create an instance of a Bot. This bot is our connection to Telegram.
3. We then use the `@bot.listener()` decorator to register an event. This library has many events. Since this library is asynchronous, we do things in a “callback” style manner.

A callback is essentially a function that is called when something happens. In our case, the `on_start()` event is called when the bot has finished logging in and setting things up and the `on_message_new()` event is called when the bot has received a message.

4. Afterwards, we check if the `Message.text` starts with '`$hello`'. If it is, then we reply to the sender with '`Hello!`'.
5. Finally, we run the bot with our login token. If you need help getting your token or creating a bot, look in the [tg-intro](#) section.

Now that we've made a bot, we have to *run* the bot. Luckily, this is simple since this is just a Python script, we can run it directly.

On Windows:

```
$ py -3 example_bot.py
```

On other systems:

```
$ python3 example_bot.py
```

Now you can try playing around with your basic bot.

1.2.2 Commands usage

tg-botting package has a lot of possibilities for creating commands easily.

Look at this example:

```
from tg_botting.bot import Bot

bot = Bot(['your', 'prefixs'], user_id, user_hash)

@bot.listener()
async def on_ready():
    print('start!')

@bot.listener()
async def on_message_new(message):
    if message.text.startswith('Hello'):
        await message.send('Hello!')

@bot.command(name='greet')
async def greet(message):
    await message.reply('Greetings!')

bot.run(bot-token)
```

As you can see, this is a slightly modified version of previous bot.

The difference is the `bot.command()` part

The commands are automatically processed messages. You may have noticed that we used a prefix when creating our bot, and the commands are what this prefix is needed for.

They are created using `Bot.command()` decorator, that can take several arguments, for example name we used here. By default it will be function name, so we didn't really need it here, but it is just more human-readable this way

So, for example, let's say your prefix of choice was '!'. It can really be anything, but we will talk about that later.

So, now when user sends ! greet to the bot, the bot will reply with Greetings!

`message` here is the instance of the `Message` class, which is automatically put into every command's first argument, so be aware of it.

`Message` has all the information you need to process the command You can find more information in the `Message` class reference

1.3 Commands

One of the most appealing aspect of the library is how easy it is to define commands and how you can arbitrarily nest commands to have a rich command system.

Commands are defined by attaching it to a regular Python function. The command is then invoked by the user using a similar signature to the Python function.

For example, in the given command definition:

```
@bot.command('foo')
async def foo(message):
    await message.send('oof!')
```

With the following prefix (\$), it would be invoked by the user via:

```
$ foo some text
```

A command must always have one parameter, `message`, which is the `Message`.

1.3.1 Invocation Message

As seen earlier, every command must take a single parameter, called the `objects.Message`.

This parameter gives you access to something called the “invocation message”. Essentially all the information you need to know how the command was executed. It contains a lot of useful information:

- `Message.user.id` to fetch the id of message author.
- `Message.chat.id` to fetch id of conversation.
- `Message.get_text()` to fetch the text of the message with out his name and prefix
- `Message.send()` to send a message to the conversation the command was used in.

1.3.2 Error Handling

When our commands fail to parse we will, by default, receive a noisy error in `stderr` of our console that tells us that an error has happened and has been silently ignored.

In order to handle our errors, we must use something called an error handler. There is a global error handler (listener), who can called `:func:`

In order to handle our errors, we must use something called an error handler. There is a global error handler, called `on_command_error()`. This global error handler is called for every error reached.

Most of the time however, we want to handle an error local to the command itself. `on_command_error()` can also handle this error

```
@bot.command('ping', ignore_filter=True)
async def ping(message):
    user = message.user
    print(0/2)
    await message.send('pong')

@bot.listener(ignore_filter=True)
async def on_command_error(message, command, exception):
    await message.reply(f"some error in {''.join(traceback.format_tb(exception.___
↳ traceback__))}")
```

The first parameter of the error handler is `Message`, because of which the error was caused, the second parameter is `Command` - the command in which the error was caused, and the third parameter is `Exception` [<https://docs.python.org/3/tutorial/errors.html>](https://docs.python.org/3/tutorial/errors.html) - an error that was called in the command.

1.3.3 Unknow commands

this method will be called when the user uses a command that the bot doesn't know. Eg:

```
@bot.listener()
async def on_unknow_command(message):
    await message.reply('the bot doesn't know this command, who called {message.text}')
```

also, almost all listeners and commands receive only one parameter as input `Message`. You can find more about other handlers below.

1.4 Cogs

There comes a point in your bot's development when you want to organize a collection of commands, listeners, and some state into one class. Cogs allow you to do just that.

The gist:

- Each cog is a Python class `Cog`.
- Every command is marked with the `cog.command()` decorator.
- Every listener is marked with the `cog.listener()` decorator.
- Cogs are then registered with the `Bot.add_cog()` call.

1.4.1 Quick Example

This example cog defines a `Greetings` category for your commands, with a single *command* named `hello` as well as a listener to listen to an *Event*.

```
from tg_botting.cog import Cog, command, listener

class Greetings(Cog):
    def __init__(self, bot):
        self.bot = bot
        self._last_user = None

    @listener()
    async def on_new_member(self, message):
        user = message.new_chat_member
        # or user = message.new_chat_participant
        # I recommend using user = message.new_chat_member or message.new_chat_
        ↪ participant
        await message.send('Welcome {}!'.format(user.first_name))

    @command('hello')
    async def hello(self, message):
        """Says hello"""
        user_id = message.user.id
        # if you need, you can try to load user by pyrogram who has in tg-botting
        # user = await User.load(user_id)
        if self._last_user is None or self._last_user != user_id:
```

(continues on next page)

(continued from previous page)

```

        await message.send('Hello {}'.format(user.first_name))
    else:
        await message.send('Hello {}... This feels familiar.'.format(user.first_
↪name))
    self._last_user = user_id

```

A couple of technical notes to take into consideration:

- All commands must now take a `self` parameter to allow usage of instance attributes that can be used to maintain state.

1.4.2 Cog Registration

Once you have defined your cogs, you need to tell the bot to register the cogs to be used. We do this via the `add_cog()` method.

```
bot.add_cog(Greetings(bot))
```

This binds the cog to the bot, adding all commands and listeners to the bot automatically.

1.4.3 Inspection

Since cogs ultimately are classes, we have some tools to help us inspect certain properties of the cog.

To get a list of commands, we can refer to dict inside the Bot class

```

>>> commands = bot.all_commands().get(cog_class_name)
>>> print([c.name for c in commands])

```

1.5 API Reference

1.5.1 Bot

1.5.2 Message

1.5.3 Event Reference

This page outlines the different types of events listened by Bot.

There are two ways to register an event, the first way is through the use of `Bot.listen()`. The second way is through subclassing Bot and overriding the specific events. For example:

```

import vk_botting

class MyBot(vk_botting.Bot):
    async def on_message_new(self, message):
        if message.from_id == self.group.id:
            return

```

(continues on next page)

(continued from previous page)

```
if message.text.startswith('$hello'):
    await message.send('Hello World!')
```

If an event handler raises an exception, `on_error()` will be called to handle it, which defaults to print a traceback and ignoring the exception.

Warning: All the events must be a [coroutine link](#). If they aren't, then you might get unexpected errors. In order to turn a function into a coroutine they must be `async def` functions.

`on_ready()`

Called when the bot is done preparing the data received from VK. Usually after login is successful and the `Bot.group` and `co.` are filled up.

`on_error(event, *args, **kwargs)`

Usually when an event raises an uncaught exception, a traceback is printed to `stderr` and the exception is ignored. If you want to change this behaviour and handle the exception for whatever reason yourself, this event can be overridden. Which, when done, will suppress the default action of printing the traceback.

The information of the exception raised and the exception itself can be retrieved with a standard call to `sys.exc_info()`.

If you want exception to propagate out of the `Bot` class you can define an `on_error` handler consisting of a single empty `py:raise`. Exceptions raised by `on_error` will not be handled in any way by `Bot`.

Parameters

- **event** (str) – The name of the event that raised the exception.
- **args** – The positional arguments for the event that raised the exception.
- **kwargs** – The keyword arguments for the event that raised the exception.

`on_command_error(ctx, error)`

An error handler that is called when an error is raised inside a command either through user input error, check failure, or an error in your own code.

A default one is provided (`Bot.on_command_error()`).

Parameters

- **ctx** (Context) – The invocation context.
- **error** (CommandError derived) – The error that was raised.

`on_command(ctx)`

An event that is called when a command is found and is about to be invoked.

This event is called regardless of whether the command itself succeeds via error or completes.

Parameters

ctx (Context) – The invocation context.

`on_command_completion(ctx)`

An event that is called when a command has completed its invocation.

This event is called only if the command succeeded, i.e. all checks have passed and the user input it correctly.

Parameters

ctx (Context) – The invocation context.

on_message_new(*message*)

Called when bot receives a message.

Parameters

message (`message.Message`) – Received message.

on_message_event(*event*)

Called when a callback button is pressed.

Parameters

event – Received event.

on_message_reply(*message*)

Called when bot replies with a message.

Parameters

message (`message.Message`) – Sent message.

on_message_edit(*message*)

Called when message is edited.

Parameters

message (`message.Message`) – Edited message.

on_message_typing_state(*state*)

Called when typing state is changed (e.g. someone starts typing).

Parameters

state (`states.State`) – New state.

on_conversation_start(*message*)

Called when user starts conversation using special button.

Parameters

message (`message.Message`) – Message sent when conversation is started.

on_chat_kick_user(*message*)

Called when user is kicked from the chat.

Parameters

message (`message.Message`) – Message sent when user is kicked.

on_chat_invite_user(*message*)

Called when user is invited to the chat.

Parameters

message (`message.Message`) – Message sent when user is invited.

on_chat_invite_user_by_link(*message*)

Called when user is invited to the chat by link.

Parameters

message (`message.Message`) – Message sent when user is invited.

on_chat_photo_update(*message*)

Called when chat photo is updated.

Parameters

message (`message.Message`) – Message sent when photo is updated.

on_chat_photo_remove(*message*)

Called when chat photo is removed.

Parameters

message (message.Message) – Message sent when photo is removed.

on_chat_create(*message*)

Called when chat is created.

Parameters

message (message.Message) – Message sent when chat is created.

on_chat_title_update(*message*)

Called when chat title is updated.

Parameters

message (message.Message) – Message sent when chat title is updated.

on_chat_pin_message(*message*)

Called when message is pinned in chat.

Parameters

message (message.Message) – Message sent when message is pinned in chat.

on_chat_unpin_message(*message*)

Called when message is unpinned in chat.

Parameters

message (message.Message) – Message sent when message is unpinned in chat.

on_message_allow(*user*)

Called when user allows getting messages from bot.

Parameters

user (user.User) – User who allowed messages.

on_message_deny(*user*)

Called when user denies getting messages from bot.

Parameters

user (user.User) – User who denied messages.

on_photo_new(*photo*)

Called when new photo is uploaded to bot group.

Parameters

photo (attachments.Photo) – Photo that got uploaded.

on_audio_new(*audio*)

Called when new audio is uploaded to bot group.

Parameters

audio (attachments.Audio) – Audio that got uploaded.

on_video_new(*video*)

Called when new video is uploaded to bot group.

Parameters

video (attachments.Video) – Video that got uploaded.

on_photo_comment_new(comment)

Called when new comment is added to photo.

Parameters

comment (group.PhotoComment) – Comment that got send.

on_photo_comment_edit(comment)

Called when comment on photo gets edited.

Parameters

comment (group.PhotoComment) – Comment that got edited.

on_photo_comment_restore(comment)

Called when comment on photo is restored.

Parameters

comment (group.PhotoComment) – Comment that got restored.

on_photo_comment_delete(comment)

Called when comment on photo is deleted.

Parameters

comment (group.DeletedPhotoComment) – Comment that got deleted.

on_video_comment_new(comment)

Called when new comment is added to video.

Parameters

comment (group.VideoComment) – Comment that got send.

on_video_comment_edit(comment)

Called when comment on video gets edited.

Parameters

comment (group.VideoComment) – Comment that got edited.

on_video_comment_restore(comment)

Called when comment on video is restored.

Parameters

comment (group.VideoComment) – Comment that got restored.

on_video_comment_delete(comment)

Called when comment on video is deleted.

Parameters

comment (group.DeletedVideoComment) – Comment that got deleted.

on_market_comment_new(comment)

Called when new comment is added to market.

Parameters

comment (group.MarketComment) – Comment that got send.

on_market_comment_edit(comment)

Called when comment on market gets edited.

Parameters

comment (group.MarketComment) – Comment that got edited.

on_market_comment_restore(comment)

Called when comment on market is restored.

Parameters

comment (group.MarketComment) – Comment that got restored.

on_market_comment_delete(comment)

Called when comment on market is deleted.

Parameters

comment (group.DeletedMarketComment) – Comment that got deleted.

on_board_post_new(comment)

Called when new post is added to board.

Parameters

comment (group.BoardComment) – New post on the board.

on_board_post_edit(comment)

Called when post on board gets edited.

Parameters

comment (group.BoardComment) – Post that got edited.

on_board_post_restore(comment)

Called when post on board is restored.

Parameters

comment (group.BoardComment) – Post that got restored.

on_board_post_delete(comment)

Called when post on board is deleted.

Parameters

comment (group.DeletedBoardComment) – Post that got deleted.

on_wall_post_new(post)

Called when new post in added to wall.

Parameters

post (group.Post) – Post that got added.

on_wall_repost(post)

Called when wall post is reposted.

Parameters

post (group.Post) – Post that got reposted.

on_wall_reply_new(comment)

Called when new comment is added to wall.

Parameters

comment (group.WallComment) – Comment that got send.

on_wall_reply_edit(comment)

Called when comment on wall gets edited.

Parameters

comment (group.WallComment) – Comment that got edited.

on_wall_reply_restore(comment)

Called when comment on wall is restored.

Parameters

comment (group.WallComment) – Comment that got restored.

on_wall_reply_delete(comment)

Called when comment on wall is deleted.

Parameters

comment (group.DeletedWallComment) – Comment that got deleted.

on_group_join(user, join_type)

Called when user joins bot group.

Parameters

- **user** (user.User) – User that joined the group.
- **join_type** (str) – User join type. Can be 'join' if user just joined, 'unsure' for events, 'accepted' if user was invited, 'approved' if user join request was approved or 'request' if user requested to join

on_group_leave(user, self)

Called when user leaves bot group.

Parameters

- **user** (user.User) – User that left the group.
- **self** (bool) – If user left on their own (True) or was kicked (False).

on_user_block(user)

Called when user is blocked in bot group.

Parameters

user (user.BlockedUser) – User that was blocked.

on_user_unblock(user)

Called when user is unblocked in bot group.

Parameters

user (user.UnblockedUser) – User that was unblocked.

on_poll_vote_new(vote)

Called when new poll vote is received.

Parameters

vote (group.PollVote) – New vote.

on_group_officers_edit(edit)

Called when group officers are edited.

Parameters

edit (group.OfficersEdit) – New edit.

on_unknown(payload)

Called when unknown event is received.

Parameters

payload (dict) – Json payload of the event.

1.5.4 Cogs

Cog

CogMeta

1.5.5 Abstract Base Classes

An `py:abstract` base class (also known as an `abc`) is a class that models can inherit to get their behaviour. The Python implementation of an `abc` is slightly different in that you can register them at run-time. **Abstract base classes cannot be instantiated.** They are mainly there for usage with `py:isinstance()` and `py:issubclass()`.

This library has a module related to abstract base classes, some of which are actually from the `abc` standard module, others which are not.

1.5.6 Utility Classes

Attachment

Keyboard

Cooldown

1.5.7 VK Models

Models are classes that are received from VK and are not meant to be created by the user of the library.

Danger: The classes listed below are **not intended to be created by users** and are also **read-only**.

For example, this means that you should not make your own `User` instances nor should you modify the `User` instance yourself.

User

Group

Message

MessageEvent

1.5.8 Exceptions

1.5.9 Additional Classes

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

B

built-in function

- on_audio_new(), 12
- on_board_post_delete(), 14
- on_board_post_edit(), 14
- on_board_post_new(), 14
- on_board_post_restore(), 14
- on_chat_create(), 12
- on_chat_invite_user(), 11
- on_chat_invite_user_by_link(), 11
- on_chat_kick_user(), 11
- on_chat_photo_remove(), 11
- on_chat_photo_update(), 11
- on_chat_pin_message(), 12
- on_chat_title_update(), 12
- on_chat_unpin_message(), 12
- on_command(), 10
- on_command_completion(), 10
- on_command_error(), 10
- on_conversation_start(), 11
- on_error(), 10
- on_group_join(), 15
- on_group_leave(), 15
- on_group_officers_edit(), 15
- on_market_comment_delete(), 14
- on_market_comment_edit(), 13
- on_market_comment_new(), 13
- on_market_comment_restore(), 13
- on_message_allow(), 12
- on_message_deny(), 12
- on_message_edit(), 11
- on_message_event(), 11
- on_message_new(), 10
- on_message_reply(), 11
- on_message_typing_state(), 11
- on_photo_comment_delete(), 13
- on_photo_comment_edit(), 13
- on_photo_comment_new(), 12
- on_photo_comment_restore(), 13
- on_photo_new(), 12
- on_poll_vote_new(), 15
- on_ready(), 10

- on_unknown(), 15
- on_user_block(), 15
- on_user_unblock(), 15
- on_video_comment_delete(), 13
- on_video_comment_edit(), 13
- on_video_comment_new(), 13
- on_video_comment_restore(), 13
- on_video_new(), 12
- on_wall_post_new(), 14
- on_wall_reply_delete(), 15
- on_wall_reply_edit(), 14
- on_wall_reply_new(), 14
- on_wall_reply_restore(), 14
- on_wall_repost(), 14

O

- on_audio_new()
 - built-in function, 12
- on_board_post_delete()
 - built-in function, 14
- on_board_post_edit()
 - built-in function, 14
- on_board_post_new()
 - built-in function, 14
- on_board_post_restore()
 - built-in function, 14
- on_chat_create()
 - built-in function, 12
- on_chat_invite_user()
 - built-in function, 11
- on_chat_invite_user_by_link()
 - built-in function, 11
- on_chat_kick_user()
 - built-in function, 11
- on_chat_photo_remove()
 - built-in function, 11
- on_chat_photo_update()
 - built-in function, 11
- on_chat_pin_message()
 - built-in function, 12
- on_chat_title_update()
 - built-in function, 12

`on_chat_unpin_message()`
 built-in function, 12

`on_command()`
 built-in function, 10

`on_command_completion()`
 built-in function, 10

`on_command_error()`
 built-in function, 10

`on_conversation_start()`
 built-in function, 11

`on_error()`
 built-in function, 10

`on_group_join()`
 built-in function, 15

`on_group_leave()`
 built-in function, 15

`on_group_officers_edit()`
 built-in function, 15

`on_market_comment_delete()`
 built-in function, 14

`on_market_comment_edit()`
 built-in function, 13

`on_market_comment_new()`
 built-in function, 13

`on_market_comment_restore()`
 built-in function, 13

`on_message_allow()`
 built-in function, 12

`on_message_deny()`
 built-in function, 12

`on_message_edit()`
 built-in function, 11

`on_message_event()`
 built-in function, 11

`on_message_new()`
 built-in function, 10

`on_message_reply()`
 built-in function, 11

`on_message_typing_state()`
 built-in function, 11

`on_photo_comment_delete()`
 built-in function, 13

`on_photo_comment_edit()`
 built-in function, 13

`on_photo_comment_new()`
 built-in function, 12

`on_photo_comment_restore()`
 built-in function, 13

`on_photo_new()`
 built-in function, 12

`on_poll_vote_new()`
 built-in function, 15

`on_ready()`
 built-in function, 10

`on_unknown()`
 built-in function, 15

`on_user_block()`
 built-in function, 15

`on_user_unblock()`
 built-in function, 15

`on_video_comment_delete()`
 built-in function, 13

`on_video_comment_edit()`
 built-in function, 13

`on_video_comment_new()`
 built-in function, 13

`on_video_comment_restore()`
 built-in function, 13

`on_video_new()`
 built-in function, 12

`on_wall_post_new()`
 built-in function, 14

`on_wall_reply_delete()`
 built-in function, 15

`on_wall_reply_edit()`
 built-in function, 14

`on_wall_reply_new()`
 built-in function, 14

`on_wall_reply_restore()`
 built-in function, 14

`on_wall_repost()`
 built-in function, 14