mmpy$_{b}otDocumentation$

Release 1.3.9

Sep 11, 2023

Contents

A chat bot for Mattermost

CHAPTER 1

Contents

1.1 Getting Started

1.1.1 Compatibility

- Python: 2.6, 3.5+ (for version <= 1.3)
- Python: 3.8+ (for version >= 2.0)
- Mattermost: 4.0+

1.1.2 Installation

PyPi (pip)

The recommended method to install *mmpy_bot* is via pip:

```
pip install -U mmpy_bot
```

Git Repo

1. Clone the git repository:

   ```
   $ git clone git@github.com:attzonko/mmpy_bot.git
   ```

2. Install requirements:

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

1.1.3 Running the bot

We recommend creating an *entrypoint* file for executing the bot, which will look something like this:

```
#!/usr/bin/env python

from mmpy_bot import Bot, Settings, ExamplePlugin, WebHookExample
# from my_plugin import MyPlugin  <== Example of importing your own plugin, don't forget to add it to the plugins list.

bot = Bot(
    settings=Settings(
        MATTERMOST_URL = "http://<mattermost_server_url>",
        MATTERMOST_PORT = 443,
        MATTERMOST_API_PATH = '/api/v4',
        BOT_TOKEN = "<your_bot_token>",
        BOT_TEAM = "<team_name>",
        SSL_VERIFY = True,
    ),
    plugins=[ExamplePlugin(), WebHookExample()],
)
bot.run()
```

You can then simply launch the bot with *python entrypoint.py*. For more information on configuring bot settings and plugins, please see settings.py and the plugins page.

Container

A container image is available at jneeven/mmpy_bot. Using your preferred container management software (Docker/Podman), you can pull the image and run it using the following steps:

1. Pull the image from the Docker repository:

```
$ podman pull docker.io/jneeven/mmpy_bot
```

2. Run the container with defined environment variables:

```
$ podman run -d --name=mmpy_bot --network=host -e MATTERMOST_URL=<mattermost_server_url> -e MATTERMOST_PORT=<mattermost_server_port> -e BOT_TOKEN=<bot_token> docker.io/jneeven/mmpy_bot
```

You can also find an example *docker-compose.yml* file here.

Customizing your bot

Getting your bot running is only the beginning. The real fun begins with writing plugins to get it functioning exactly how you want it! Head on over to the plugins page to get started.

Fetch mmpy_bot version

To check your installed version of *mmpy_bot*, simply open a Python interpreter and run the following commands:

```
import mmpy_bot
print(mmpy_bot.__version__)
```

1.2 Plugins

A chat bot is meaningless unless you can extend/customize it to fit your own use cases, which can be achieved through custom plugins.

1.2.1 Writing your first plugin

1. To demonstrate how easy it is to create a plugin for *mmpy_bot*, let's write a basic plugin and run it. Start with an empty Python file and import these three *mmpy_bot* modules:

```
from mmpy_bot import Plugin, listen_to
from mmpy_bot import Message
```

2. Now create a Class with the name of your plugin, subclassing *Plugin*:

```
class MyPlugin(Plugin):
```

3. Now we can write the methods that control how the bot will respond to certain messages. Let's start with a basic one that will simply trigger a response from the bot:

```
@listen_to("wake up")
async def wake_up(self, message: Message):
    self.driver.reply_to(message, "I'm awake!")
```

   In the above code block, the `@listen_to` decorator tells the bot to listen on any channel for the string "wake up", and respond with "I'm awake!".

4. Save your plugin file and open a fresh Python file which will be the entrypoint to start the bot and include your plugin:

```
#!/usr/bin/env python

from mmpy_bot import Bot, Settings
from my_plugin import MyPlugin

bot = Bot(
    settings=Settings(
        MATTERMOST_URL = "http://127.0.0.1",
        MATTERMOST_PORT = 8065,
        BOT_TOKEN = "<your_bot_token>",
        BOT_TEAM = "<team_name>",
        SSL_VERIFY = False,
    ),  # Either specify your settings here or as environment variables.
    plugins=[MyPlugin()],  # Add your own plugins here.
)
bot.run()
```

   The above code assumes your plugin is in the same directory as the entrypoint file. Also be sure to set the correct settings for your Mattermost server and bot account.

5. Save your entrypoint file and run it from the command prompt:

```
$ ./my_bot.py
```

If everything went as planned you can now start your bot, send the message "wake up" and expect the appropriate reply.

1.2.2 Further configuration

The below code snippets provide an insight into the functionality that can be added to the bot. For more in-depth examples, please refer to ./plugins/example.py and ./plugins/webhook_example.py.

1.2.3 Implementing regular expression

```
import re

@listen_to('hi', re.IGNORECASE)
def hi(message):
    message.reply('I can understand hi or HI!')

@listen_to('Give me (.*)')
async def give_me(self, message, something):
    self.driver.reply_to(message, 'Here is %s' % something)
```

1.2.4 Only accept messages that mention the bot

If you want the bot to only respond to messages containing a mention (e.g. "hey @bot_name !"), you can use the *needs_mention* flag. Note that this will also trigger if you send the bot a direct message without mentioning its name!

```
@listen_to("hey", needs_mention=True)
async def hey(self, message: Message):
    self.driver.reply_to(message, "Hi! You mentioned me?")
```

1.2.5 Only accept direct messages

Using *direct_only=True*, the bot will only respond if you send it a direct message.

```
@listen_to("hey", direct_only=True)
async def hey(self, message: Message):
    self.driver.reply_to(message, "Hi! This is a private conversation.")
```

1.2.6 Restrict messages to specific users

```
@listen_to("^admin$", direct_only=True, allowed_users=["admin", "root"])
async def users_access(self, message: Message):
    """Will only trigger if the username of the sender is 'admin' or 'root'."
↪""
    self.driver.reply_to(message, "Access allowed!")
```

1.2.7 Restrict messages to specific channels

```
@listen_to("^poke$", allowed_channels=["#staff", "#town-square"])
async def poke(self, message: Message):
    """Will only trigger if the message has been send in '#staff' or '#town-
↪square'."""
    self.driver.reply_to(message, "Access allowed!")
```

1.2.8 Extra listener metadata

In some cases, it is helpful to add extra metadata to listeners. The example below shows `category` and `human_description`.

`category` is used by `HelpPlugin` to group listeners from the same category while `human_description` is displayed instead of the listener regular expression. The latter is particularly useful if users of the bot are not familiar with or don't know how to read regular expressions.

```
import re

@listen_to(
    "^reply at (.*)$",
    re.IGNORECASE,
    needs_mention=True,
    category="schedule",
    human_description="reply at TIMESTAMP",
)
def schedule_once(self, message: Message, trigger_time: str):
    """Schedules a reply to be sent at the given time."""
    (...)
```

You can also pass arbitrary keywords in the `listen_to` decorator. These will be made available through `FunctionInfo.metadata` instances, described in more detail below.

1.2.9 Click support

mmpy_bot now supports click commands, so you can build a robust CLI-like experience if you need it. The example below registers a *hello_click* command that takes a positional argument, a keyword argument and a toggleable flag, which are automatically converted to the correct type. For example, it can be called with *hello_click my_argument –keyword-arg=3 -f* and will parse the arguments accordingly. A nice benefit of *click* commands is that they also automatically generate nicely formatted help strings. Try sending "help" to the *ExamplePlugin* to see what it looks like!

```
@listen_to("hello_click", needs_mention=True)
@click.command(help="An example click command with various arguments.")
@click.argument("POSITIONAL_ARG", type=str)
@click.option("--keyword-arg", type=float, default=5.0, help="A keyword arg.")
@click.option("-f", "--flag", is_flag=True, help="Can be toggled.")
def hello_click(
    self, message: Message, positional_arg: str, keyword_arg: float, flag: bool
):
    """A click function documented via docstring"""
    response = (
        "Received the following arguments:\n"
        f"- positional_arg: {positional_arg}\n"
        f"- keyword_arg: {keyword_arg}\n"
        f"- flag: {flag}\n"
    )
    self.driver.reply_to(message, response)
```

1.2.10 Custom help messages

mmpy_bot defaults to responding to *@botname help* or *help* in a direct message if the *HelpPlugin* is enabled. If you wish to customize the way help is displayed you can subclass *HelpPlugin* and override *get_help_string*. If instead you

want to customize which help functions are displayed but not the format of the help text you can override *get_help*. To access information about active plugins call *self.plugin_manager.get_help()* which will return *FunctionInfo* instances. See below for an example.

```
from mmpy_bot.plugins import HelpPlugin

class MyAdminOnlyHelpPlugin(HelpPlugin):
    def get_help(self, message):
        """Show admin plugins only to admin_user."""
        function_info = super().get_help(message)

        if message.sender_name != "admin_user":
            return [x for x in function_info if x.metadata.get("category") != "admin"]
        else:
            return function_info

    def get_help_string(self, message):
        list_of_help_info = self.get_help(message)
        return f"This is all the help I can share {list_of_help_info}"
```

FunctionInfo provides the following attributes:

- *help_type* - a string *message* or *webhook*
- *location* - name of the plugin class
- *function* - function object decorated with *listen_to*
- *pattern* - regular expression or pattern that triggers the function
- *docheader* - first line of docstring of decorated function
- *docfull* - docstring of decorated function - for *click* functions, includes formatted help text
- *direct* - *True* if function can only be used via direct message
- *mention* - *True* if function can only be triggered by prefixing with *@botname*
- *metadata* - a dictionary with additional keyword arguments passed to *listen_to* or *listen_webhook*.

If necessary, you can also access the docstring of the plugin class with:

```
from mmpy_bot.utils import split_docstring

head, full = split_docstring(FunctionInfo.function.plugin.__doc__)
```

You should then enable your custom plugin by adding it to the list of enabled plugins:

```
from mmpy_bot import Bot, Settings
from my_help_plugin import MyHelpPlugin

bot = Bot(
    settings=Settings(
        ...,
    ),
    plugins=[
        MyHelpPlugin()
        ...,
    ],
)
bot.run()
```

If you wish to have the bot respond to *!help* in any channel, you can set the *RESPOND_CHANNEL_HELP* setting to *True*.

1.2.11 File upload

```
@listen_to("^hello_file$", re.IGNORECASE, needs_mention=True)
async def hello_file(self, message: Message):
    """Responds by uploading a text file."""
    file = Path("/tmp/hello.txt")
    file.write_text("Hello from this file!")
    self.driver.reply_to(message, "Here you go", file_paths=[file])
```

1.2.12 Plugin startup and shutdown

The *Plugin* class comes with an *on_start* and *on_stop* function, which will be called when the bot starts up or shuts down. They can be used as follows:

```
def on_start(self):
    """Notifies some channel that the bot is now running."""
    self.driver.create_post(channel_id="some_channel_id", message="The bot just␣
↪started running!")

def on_stop(self):
    """Notifies some channel that the bot is shutting down."""
    self.driver.create_post(channel_id="some_channel_id", message="I'll be right back!
↪")
```

1.2.13 Webhook listener

If you want to interact with your bot not only through chat messages but also through web requests (for example to implement an interactive dialog), you can use enable the built-in *WebHookServer*. In your *Settings*, make sure to set *WEBHOOK_HOST_ENABLED=True* and provide a value for *WEBHOOK_HOST_URL* and *WEBHOOK_HOST_PORT* (see settings.py for more info). Then, on your custom *Plugin* you can create a function like this:

```
from mmpy_bot import listen_webhook

@listen_webhook("ping")
async def ping_listener(self, event: WebHookEvent):
    """Listens to post requests to '<server_url>/hooks/ping' and posts a message in
    the channel specified in the request body."""

    self.driver.create_post(
        event.body["channel_id"], f"Webhook {event.webhook_id} triggered!"
    )
```

And if you want to send a web response back to the incoming HTTP POST request, you can use *Driver.respond_to_web*:

```
@listen_webhook("ping")
async def ping_listener(self, event: WebHookEvent):
    # Respond to the web request rather than posting a message.
    self.driver.respond_to_web(
```

(continues on next page)

(continued from previous page)

```
        event,
        {
            # You can add any kind of JSON-serializable data here
            "message": "hello!",
        },
    )
```

For more information about the *WebHookServer* and its possibilities, take a look at the WebHookExample plugin.

1.2.14 Job scheduling

mmpy_bot integrates schedule to provide in-process job scheduling.

With schedule, we can put periodic jobs into waiting queue like this:

```
@listen_to("^schedule every ([0-9]+)$", re.IGNORECASE, needs_mention=True)
def schedule_every(self, message: Message, seconds: int):
    """Schedules a reply every x seconds. Use the `cancel jobs` command to stop.

    Arguments:
    - seconds (int): number of seconds between each reply.
    """
    schedule.every(int(seconds)).seconds.do(
        self.driver.reply_to, message, f"Scheduled message every {seconds} seconds!"
    )

@listen_to('cancel jobs', re.IGNORECASE)
def cancel_jobs(message):
    schedule.clear()
    self.driver.reply_to('All jobs cancelled.')
```

The schedule package provides human-readable APIs to schedule jobs. Check out schedule.readthedocs.io for more usage examples.

schedule is designed for periodic jobs. In order to support one-time-only jobs, mmpy_bot has a monkey-patching on integrated schedule package.

We can schedule a one-time-only job by *schedule.once* method. You should notice that this method takes a datetime object, which is different from *schedule.every* methods.

The following code example uses *schedule.once* to schedule a job. This job will be trigger at *t_time*.

1.3 Contributing

1.3.1 Setup your environment

We recommend using venv to keep your development environment isolated from your base Python environment. It is part of Python by default.

1. Clone the mmpy_bot repository, setup your virtual environment and install the requirements:

```
$ git clone https://github.com/attzonko/mmpy_bot.git
$ cd mmpy_bot
$ python3 -m venv venv
```

(continues on next page)

(continued from previous page)

```
$ source venv/bin/activate
$ pip install -e ".[dev]"
```

2. Spin up the Mattermost container (Podman or Docker required):

```
$ podman-compose -f tests/integration_tests/docker-compose.yml up -d
```

3. In order to run the bot, it is advised to use an entrypoint Python file which defines your Mattermost server and bot account settings, as well as importing any custom plugins you may create. See the provided entrypoint.py as a reference. To run your bot inside a docker container (not necessary, you can also just run *python entrypoint.py*) you can use the provided docker-compose file.

1.4 Testing

mmpy_bot develops all tests based on pytest. If you need to add your own tests and run tests, please install the dev requirements.

```
$ pip install -r dev-requirements.txt
```

All the tests are put in `mmpy_bot\tests`. There are two test packages: `unit_tests` and `integration_tests`.

Tests which can be performed by a single bot without requiring a server or interaction with other bots should be kept in the `unit_tests` package. Tests that require interactions between bots on a mattermost server belong to the `integration_tests` package.

1.4.1 Adding unit tests

There are multiple test modules inside unit_tests package, one for each module in the code. The naming convention of these modules is *modulename_test*. Inside each module, there will be several test functions with naming convention *test_methodname*, grouped into classes for each corresponding class in the code. If you need to add more unit tests, please consider following these conventions.

1.4.2 Running the unit tests

To run the unit tests (in parallel), simply execute:

```
$ pytest -n auto tests\unit_tests
```

1.4.3 Adding integration tests

The integration tests are run on the `jneeven:mattermost-bot-test` docker image, for which dockerfiles are provided in the `tests/intergration_tests` folder. The tests are defined as interactions between a bot (the responder) and a driver (the one sending test messages), which live inside the docker image. Their respective tokens are available in `tests/integration_tests/utils.py`, and the two bots are available as pytest fixtures so they can be easily re-used. Note that while the bot is also a fixture, it should not be used in any functions. It will simply be started whenever the integration tests are executed.

An integration test might look like this (also have a look at the actual code in `tests/integration_tests/test_example_plugin.py`):

```
from tests.integration_tests.utils import start_bot  # noqa, only imported␣
↪so that the bot is started
from tests.integration_tests.utils import MAIN_BOT_ID, OFF_TOPIC_ID,␣
↪RESPONSE_TIMEOUT, TEAM_ID
from tests.integration_tests.utils import driver as driver_fixture
from tests.integration_tests.utils import expect_reply

# Hacky workaround to import the fixture without linting errors
driver = driver_fixture

# Verifies that the bot is running and listening to this non-targeted message
def test_start(driver):
    post = driver.create_post(OFF_TOPIC_ID, "starting integration tests!")
    # Checks whether the bot has sent us the expected reply
    assert expect_reply(driver, post)["message"] == "Bring it on!"
```

In this test, the driver sends a message in the "off-topic" channel, and waits for the bot to reply 'Bring it on!'. If no reply occurs within a default response timeout (15 seconds by default, but this can be passed as an argument to `expect_reply`), an exception will be raised. The driver fixture is imported from the utils and can be re-used in every test function simply by adding it as a function argument.

1.4.4 Running the integration_tests

Running the integration_tests is easy: simply `cd` into `tests/integration_tests`, and run `docker-compose up -d` to start a local mattermost server. Then run `pytest -n auto .` to start the tests! For more info about the integration tests an the docker server, have a look at *tests/integration_tests/README.md*.

1.4.5 Test coverage:

Install pytest-cov:

```
$ pip install pytest-cov
```

Set necessary configuration as described above, and run:

```
$ py.test --cov=mmpy_bot tests\
```

It automatically runs tests and measures code coverage of modules under mmpy_bot root dir. Using "–cov-report" parameter to write report into "cov_html" folder by html format.

```
py.test --cov-report html:logs\cov_html --cov=mmpy_bot tests\
```

1.5 Credits

1.5.1 Active Contributors

- Alex Tzonkov <alex.tzonkov@gmail.com>
- Jelmer Neeven <jelmer@neeven.tech>
- Renato Alves <alves.rjc@gmail.com>
- Thomas Tuffin <ttuffin@redhat.com>

1.5.2 Past Contributors

- Victor Hu <selain@nature.ee.ncku.edu.tw>
- tgly307 <tgly307@163.com>
- GoTLiuM InSPiRiT <gotlium@gmail.com>
- Grokzen <grokzen@gmail.com>
- Philipp Schmitt <philipp@schmitt.co>
- Tom Horlock <thomashorlock1993@gmail.com>
- Evgeniy Poturaev <gig177@ya.ru>

CHAPTER 2

Contributing

You can grab latest code on `main` branch at Github.

Feel free to submit issues, pull requests are also welcome.

Good contributions follow *simple guidelines*