Simple-Matrix-Bot-Lib

Release 2.8.0

KrazyKirby99999

Nov 08, 2022

GETTING STARTED

Simple-Matrix-Bot-Lib is a Python 3 library for quickly building Matrix bots. It is based on the matrix-nio Python library. To begin using Simple-Matrix-Bot-Lib, *check out the quickstart*.

CHAPTER

ONE

QUICKSTART

View on Github or View on PyPi

1.1 Install the simplematrixbotlib package

Simple-Matrix-Bot-Lib's package is simplematrixbotlib. It can be installed from pip or downloaded from github. Installation from pip:

```
python -m pip install simplematrixbotlib
```

Download from github:

```
git clone --branch master https://github.com/KrazyKirby99999/simple-matrix-bot-lib.git
```

1.2 Obtain Matrix login credentials

Go to https://app.element.io/#/register

If you are already using element web, then you may want to use a private session in your browser.

Change the homeserver if you prefer, and enter a new username, password, and/or email into the respective fields.

Save the homeserver, username, and password at a safe location, then complete the captcha.

Your bot's login credentials should resemble the following:

homeserver: https://example.com

username: example_bot

password: secretpassword

1.3 Create the bot

(Finished example code will be provided in full at the bottom)

Begin by importing the package.

```
import simplematrixbotlib as botlib
```

Create a Creds object with your login credentials.

```
creds = botlib.Creds("https://home.server", "user", "pass")
```

Create a bot object. This will be used as a handle throughout your project.

```
bot = botlib.Bot(creds)
```

If you want to use a prefix in the commands that your bot responds to, it may be useful to assign it to a variable.

```
PREFIX = '!'
```

Before creating a function handler for a command, it is necessary to add a listener.

```
@bot.listener.on_message_event
```

Create a command by defining a function. The function must be an "async" function with two arguments. Recommended argument names are (room, message) or (room, event)

```
async def echo(room, message):
    """
    Example command that "echoes" arguements.
    Usage:
    example_user- !echo say something
    echo_bot- say something
    """
```

Creating a MessageMatch object is optional, but useful for handling messages. The prefix argument is optional, but is needed when matching prefixes.

```
match = botlib.MessageMatch(room, message, bot, PREFIX)
```

This specific usage of the MessageMatch class will only allow the bot to react to messages that are not from the bot and also start with "!echo".

```
if match.is_not_from_this_bot() and match.prefix() and match.command("echo"):
```

This part of the handler is responsible for sending the response message. The rest of the message following "!echo" will be sent to the same room as the message.

```
await bot.api.send_text_message(room.room_id, " ".join(arg for arg in match.args()))
```

Finally run the bot.

```
bot.run()
```

This bot is an echo bot, which "echoes" the arguments of any message that starts with "!echo"(). As many handlers as needed can be added, each with its own handler function and a listener.

Full code of echo bot example

```
import simplematrixbotlib as botlib

creds = botlib.Creds("https://home.server", "user", "pass")
bot = botlib.Bot(creds)
PREFIX = '!'

@bot.listener.on_message_event
async def echo(room, message):
    match = botlib.MessageMatch(room, message, bot, PREFIX)

    if match.is_not_from_this_bot() and match.prefix() and match.command("echo"):

        await bot.api.send_text_message(
            room.room_id, " ".join(arg for arg in match.args())
            )

bot.run()
```

Other examples can be found here.

CHAPTER

TWO

EXAMPLES

2.1 A Basic Echo Bot

```
"""
Example Usage:

random_user
      !echo something

echo_bot
      something
"""

import simplematrixbotlib as botlib

creds = botlib.Creds("https://home.server", "user", "pass")
bot = botlib.Bot(creds)
PREFIX = '!'


@bot.listener.on_message_event
async def echo(room, message):
    match = botlib.MessageMatch(room, message, bot, PREFIX)

    if match.is_not_from_this_bot() and match.prefix() and match.command(
            "echo"):

        await bot.api.send_text_message(room.room_id,
                                        " ".join(arg for arg in match.args()))


bot.run()
```

2.2 An Echo Bot with Encryption and Verification Support

```
"""
Example Usage:

random_user
      *emoji verification or one-sided verification

random_user
      !echo something

echo_bot
      something
"""

import simplematrixbotlib as botlib

config = botlib.Config()
# config.encryption_enabled = True  # Automatically enabled by installing encryption␣
↪support
config.emoji_verify = True
config.ignore_unverified_devices = True

creds = botlib.Creds("https://home.server", "user", "pass")
bot = botlib.Bot(creds, config)
PREFIX = '!'


@bot.listener.on_message_event
async def echo(room, message):
    match = botlib.MessageMatch(room, message, bot, PREFIX)

    if match.is_not_from_this_bot()\
            and match.prefix()\
            and match.command("echo"):

        await bot.api.send_text_message(room.room_id,
                                        " ".join(arg for arg in match.args()))


bot.run()
```

2.3 A “High-Five” Bot

```
"""
Example Usage:

random_user
      !count
```

(continues on next page)

(continued from previous page)

```
echo_bot
      The bot has been high-fived 10 times!

random_user
      !high_five

echo_bot
      random_user high-fived the bot!"

random_user
      !count

echo_bot
      The bot has been high-fived 11 times!
"""

import simplematrixbotlib as botlib

creds = botlib.Creds("https://example.org", "hight_five_bot", "secretpassword")
bot = botlib.Bot(creds)

PREFIX = '!'

try:
    with open("high_fives.txt", "r") as f:
        bot.total_high_fives = int(f.read())
except FileNotFoundError:
    bot.total_high_fives = 0


@bot.listener.on_message_event
async def bot_help(room, message):
    bot_help_message = f"""
    Help Message:
        prefix: {PREFIX}
        commands:
            help:
                command: help, ?, h
                description: display help command
            give high fives:
                command: high_five, hf
                description: high-five the bot!
            count:
                command: count, how_many, c
                description: show amount of high fives
                """
    match = botlib.MessageMatch(room, message, bot, PREFIX)
    if match.is_not_from_this_bot() and match.prefix() and (
            match.command("help") or match.command("?") or match.command("h")):
        await bot.api.send_text_message(room.room_id, bot_help_message)
```

(continues on next page)

(continued from previous page)

```
@bot.listener.on_message_event
async def high_five(room, message):
    match = botlib.MessageMatch(room, message, bot, PREFIX)
    if match.is_not_from_this_bot() and match.prefix() and (
            match.command("high_five") or match.command("hf")):

        bot.total_high_fives += 1
        with open("high_fives.txt", "w") as f:
            f.write(str(bot.total_high_fives))

        await bot.api.send_text_message(
            room.room_id, f"{message.sender} high-fived the bot!")


@bot.listener.on_message_event
async def high_five_count(room, message):
    match = botlib.MessageMatch(room, message, bot, PREFIX)
    if match.is_not_from_this_bot() and match.prefix() and (
            match.command("count") or match.command("how_many")
            or match.command("c")):
        await bot.api.send_text_message(
            room.room_id,
            f"The bot has been high-fived {str(bot.total_high_fives)} times!")


bot.run()
```

2.4 A Rock Paper Scissors Bot

```
import simplematrixbotlib as botlib
import os
import random

creds = botlib.Creds("https://example.org", "echo_bot", "secretpassword")
bot = botlib.Bot(creds)

PREFIX = '!'


@bot.listener.on_message_event
async def help_message(room, message):
    match = botlib.MessageMatch(room, message, bot, PREFIX)
    if not (match.is_not_from_this_bot() and match.prefix()
            and match.command("help")):
        return

    message = (f"""
    Help
    ============================
    What is this bot?
```

(continues on next page)

(continued from previous page)

```
        Rock Paper Scissors Bot is a Matrix bot that plays rock paper scissors with room␣
↪members and is written in Python using the simplematrixbotlib package.
    Commands?
        {PREFIX}help - show this message
        {PREFIX}play <rock/paper/scissors> - play the game by making a choice
    """)

    await bot.api.send_text_message(room.room_id, message)


@bot.listener.on_message_event
async def make_choice(room, message):
    match = botlib.MessageMatch(room, message, bot, PREFIX)
    if not (match.is_not_from_this_bot() and match.prefix()
            and match.command("play")):
        return

    temp = True
    if not match.args():
        temp = False
    elif "rock" == match.args()[0]:
        choice = "rock"
    elif "paper" == match.args()[0]:
        choice = "paper"
    elif "scissors" == match.args()[0]:
        choice = "scissors"
    else:
        temp = False

    victory_table = {"rock": "scissors", "scissors": "paper", "paper": "rock"}

    if temp:
        bot_choice = random.choice(["rock", "paper", "scissors"])

        await bot.api.send_text_message(room.room_id, f"You choose {choice}.")
        await bot.api.send_text_message(room.room_id,
                                        f"The bot chose {bot_choice}.")

        if choice == bot_choice:
            await bot.api.send_text_message(room.room_id, "You Tied!")
        if bot_choice == victory_table[choice]:
            await bot.api.send_text_message(room.room_id, "You Won!")
        if choice == victory_table[bot_choice]:
            await bot.api.send_text_message(room.room_id, "You Lost!")

    else:
        await bot.api.send_text_message(
            room.room_id,
            "Invalid choice. Please choose \"rock\", \"paper\", or \"scissors\"."
        )
```

(continues on next page)

(continued from previous page)

```
bot.run()
```

2.5 A Reaction Echo Bot

```
"""
Example Usage:

random_user
      !echo something

random_user2
      *reacts with

echo_reaction_bot
      Reaction:
"""

import simplematrixbotlib as botlib

creds = botlib.Creds("https://example.com", "echo_reaction_bot", "password")
bot = botlib.Bot(creds)


@bot.listener.on_reaction_event
async def echo_reaction(room, event, reaction):
    resp_message = f"Reaction: {reaction}"
    await bot.api.send_text_message(room.room_id, resp_message)


bot.run()
```

2.6 Echo Bot within a Docker Container

```
FROM python:latest

RUN python -m pip install simplematrixbotlib

ADD echo.py echo.py

CMD [ "python", "echo.py" ]
```

2.7 Echo Bot using Config Files

```
"""
Example Usage:

random_user
      !echo something

echo_bot
      something
"""

import simplematrixbotlib as botlib

creds = botlib.Creds("https://home.server", "user", "pass")

config = botlib.Config()
config.load_toml("config.toml")

bot = botlib.Bot(creds, config)
PREFIX = '!'


@bot.listener.on_message_event
async def echo(room, message):
    match = botlib.MessageMatch(room, message, bot, PREFIX)

    if match.is_not_from_this_bot() and match.prefix() and match.command(
            "echo"):

        await bot.api.send_text_message(room.room_id,
                                        " ".join(arg for arg in match.args()))


bot.run()
```

2.7.1 Bot Config File in TOML format

```
[simplematrixbotlib.config]
join_on_invite = false
```

2.8 Echo Bot with Allow/Blocklist Config File

```
#Used with Bot Config File in TOML format
"""
Example Usage:

admin1
      !echo something

echo_bot
      something

admin3
      !echo something

user1
      !echo something
"""

import simplematrixbotlib as botlib

creds = botlib.Creds("https://home.server", "user", "pass")

config = botlib.Config()
config.load_toml("config.toml")

bot = botlib.Bot(creds, config)
PREFIX = '!'


@bot.listener.on_message_event
async def echo(room, message):
    match = botlib.MessageMatch(room, message, bot, PREFIX)

    if match.is_not_from_this_bot() \
       and match.is_from_allowed_user() \
       and match.prefix() \
       and match.command("echo"):

        await bot.api.send_text_message(room.room_id,
                                        " ".join(arg for arg in match.args()))


bot.run()
```

2.8.1 Bot Config File in TOML format

```
[simplematrixbotlib.config]
allowlist = ['@admin.*:example\.org']
blocklist = ['@admin3:example\.org']
```

2.9 Using Allow/Blocklist Interactively + Config File

```
"""
Example Usage:
note the escaped dot (\.)

user1
      !allow @user2:example\.org

admin1
      !allow @user1:example\.org, @admin2:example\.org

echo_bot
      allowing @user1:example\.org, @admin2:example\.org

user1
      !allow @user2:example\.org

echo_bot
      allowing @user2:example\.org

admin2
      !disallow @user1:example\.org
"""

import simplematrixbotlib as botlib

creds = botlib.Creds("https://home.server", "user", "pass")

config = botlib.Config()
config.load_toml("config_allow_interactive.toml")

bot = botlib.Bot(creds, config)
PREFIX = '!'


@bot.listener.on_message_event
async def echo(room, message):
    match = botlib.MessageMatch(room, message, bot, PREFIX)

    if match.is_not_from_this_bot() \
       and match.is_from_allowed_user() \
       and match.prefix():

        if match.command("allow"):
```

(continues on next page)

(continued from previous page)

```
            bot.config.add_allowlist(set(match.args()))
            await bot.api.send_text_message(
                room.room_id,
                f'allowing {", ".join(arg for arg in match.args())}')

        if match.command("disallow"):
            bot.config.remove_allowlist(set(match.args()))
            await bot.api.send_text_message(
                room.room_id,
                f'disallowing {", ".join(arg for arg in match.args())}')


bot.run()
```

2.9.1 Bot Config File in TOML format

```
[simplematrixbotlib.config]
allowlist = ['@admin1:example\.org']
```

2.10 Bot using Custom Option Config File

```
"""
Example Usage:

random_user
      !get

echo_bot
      something
"""

import simplematrixbotlib as botlib
from dataclasses import dataclass


@dataclass
class MyConfig(botlib.Config):
    _my_setting: str = "Hello"

    @property
    def my_setting(self) -> str:
        return self._my_setting

    @my_setting.setter
    def my_setting(self, value: str) -> None:
        self._my_setting = value
```

(continues on next page)

(continued from previous page)

```
creds = botlib.Creds("https://home.server", "user", "pass")
config = MyConfig()
config.load_toml('config_custom.toml')
bot = botlib.Bot(creds, config)
PREFIX = '!'


@bot.listener.on_message_event
async def get(room, message):
    match = botlib.MessageMatch(room, message, bot, PREFIX)

    if match.is_not_from_this_bot() and match.prefix() and match.command(
            "get"):

        await bot.api.send_text_message(room.room_id, config.my_value)


bot.run()
```

2.10.1 Bot Config File in TOML format

```
[simplematrixbotlib.config]
my_value = 'Hello World'
```

CHAPTER

THREE

USAGE WITH DOCKER

Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. By using docker, you can ensure that your bot's environment is the same regardless of whether your bot is self-hosted, cloud hosted, or hosted anywhere else that supports docker.

3.1 Install Docker

Start by installing Docker using the instructions provided on the docker website.

3.2 Prepare the bot code

After you have installed docker, you will next need to have the source code of your bot ready. For this example we will be using the echo bot from the quickstart.

3.3 Write the Dockerfile

Docker uses instructions located within files with the name of "Dockerfile" (no file extension). A completed dockerfile will be provided at the end.

Create a file with a name of "Dockerfile", and without any file extension. To begin the Dockerfile, we will need to set a base image. Docker will use this image as a starting point for our docker image during the build.

```
FROM python:latest
```

Python is preinstalled with this image, however the python packages needed will still need to be installed using the following line.

```
RUN python -m pip install simplematrixbotlib
```

Copy your bot's source code to the container using ADD. The first argument is the location in the host's filesystem, and the second argument is the location in the container's filesystem.

```
ADD echo.py echo.py
```

It will then be neccesary to set a command for docker to run when a docker image of the container is run. Use python as the command, and add any neccesary arguments using the same syntax as a python list of strings.

```
CMD [ "python", "echo.py" ]
```

3.4 Build the Docker container

Before you can run the bot, docker will need to build a container using the Dockerfile. The syntax for this command is "docker build -t container-name Directory-with-Dockerfile"

```
docker build -t echo-bot .
```

3.5 Create and run Docker image

Docker will automatically create a Docker image from the container and run it when you use the following command.

```
docker run echo-bot
```

If you want to be able to view print output in your terminal or cmd prompt, then use the following command instead.

```
docker run -e PYTHONBUFFERED=1 echo-bot
```

This concludes this guide. More information on Docker can be found at docker.com and more information on simplematrixbotlib can be found elsewhere in this documentation.

CHAPTER

FOUR

MANUAL

This is a manual for simplematrixbotlib that includes documentation, examples, and more.

4.1 Installation

The simplematrixbotlib package can be installed from pypi or from the git repository.

4.1.1 Installing from PyPi

Run the following command in your terminal or cmd prompt to install simplematrixbotlib from pypi

```
python -m pip install simplematrixbotlib
```

See *Encryption* to lean how to install E2E encryption support.

4.1.2 Installing from Git Repo

Run the following command in your terminal or cmd prompt to download the repository to your machine.

```
git clone --branch master https://github.com/KrazyKirby99999/simple-matrix-bot-lib.git
```

The package is located under (current directory)/simple-matrix-bot-lib as simplematrixbotlib.

4.2 Importing

Importing simplematrixbotlib requires *installation* of the package first.

4.2.1 How to import simplematrixbotlib

Importing the simplematrixbotlib package is done with the following python code.

```
import simplematrixbotlib as botlib
```

Referring to the package as "botlib" is optional, however this is how the simplematrixbotlib will be referred to throughout this manual and the rest of the documentation.

4.3 E2E Encryption

4.3.1 Requirements

End-to-end encryption support requires some additional dependencies to be installed, namely the `e2e` extra of `matrix-nio`. In turn, `matrix-nio[e2e]` requires libolm version 3.0.0 or newer. You can install it using you distribution's package manager or from source.

More information is available at matrix-nio.

Finally, install e2e support for matrix-nio by running:

```
python -m pip install "matrix-nio[e2e]"
```

If there are issues installing the e2e extra with pip from PyPI, additional packages may be required to build python-olm on your distribution, for example python3-devel on openSUSE.

4.3.2 Enabling

Encryption needs to be enabled in simplematrixbotlib's Config before calling `bot.run()`. When the dependencies are met, it will be enabled automatically but can be turned off if required.

```
config = botlib.Config()
config.encryption_enabled = True
config.emoji_verify = True
config.ignore_unverified_devices = False
config.store_path = './crypto_store/'
bot = botlib.Bot(creds, config)
bot.run()
```

4.3.3 Configuration Options

See *the Config class manual* to learn about settings regarding encryption provided by the Config class.

Additionally, you can manage trusted and distrusted devices using nio directly using the following methods. There are 4 states:

- default: Initially, devices are not trusted. Trying to send a message when such a device is present will cause an Exception, unless `ignore_unverified_devices` is enabled. This state resembles Element's setting "Never send encrypted messages to unverified sessions from this session".
- ignored: Nio will ignore that this device is not verified and send encrypted messages to it regardless. This resembles the default "gray shield" used by Element.

- verified: This is an explicitly trusted device and will receive messages. This resembles the "green shield" in Element.
- blacklisted: This device is explicitly untrusted and will not receive encrypted messages. This resembles the "red shield" in Element.

```
# set a device's trust state
# verifying a blacklisted or ignored device will automatically remove the former state
bot.async_client.olm.verify_device(device)
bot.async_client.olm.ignore_device(device)
bot.async_client.olm.blacklist_device(device)

# unset a device's trust state
bot.async_client.olm.unverify_device(device)
bot.async_client.olm.unignore_device(device)
bot.async_client.olm.unblacklist_device(device)

# check a device's trust state
bot.async_client.olm.is_device_verified(device)
bot.async_client.olm.is_device_ignored(device)
bot.async_client.olm.is_device_blacklisted(device)
```

4.3.4 Verification

The library supports 2 common types of verification.

Manual "Session key" fingerprint verification

Upon startup, when encryption is enabled, simplematrixbotlib will print some information about its device similar to this:

```
Connected to https://client.matrix.org as @simplematrixbotlib:matrix.org (ABCDEFGHIJKL)
This bot's public fingerprint ("Session key") for one-sided verification is: 0123 4567␣
↪89ab cdef ghij klmn opqr stuv wxyz ACBD EFG
```

1. Using the "Session ID" (e.g. `ABCDEFGHIJKL`) given in braces after the bot's Matrix ID and the fingerprint given in the next line, we can proceed to do verify our bot from our Matrix client.
2. In Element Web or Desktop, open the bot user's info and click on "X session(s)" - NOT on "Verify".
3. The bot's current sessions named "Bot Client using Simple-Matrix-Bot-Lib" will be listed with gray shields next to them.
4. Click the session with the correct Session ID, then select "Manually Verify by Text".
5. Confirm that Session ID and Session key shown in Element match those printed by your bot, then click "Verify session".

You have now verified your bot session one-sided from Element. This means, Element now knows that it really is your bot and be able to detect any attacks and show a red shield. However, since this is one-sided verification, your bot does not know the same about your Element session.

Interactive SAS verification using Emoji

The library is able to perform interactive to-device verification using the SAS method and Emoji. In-room verification is not supported by nio at this time, thus only single devices can be verified with each other individually. This method appears **not** to be supported by some clients, such as Element Android, at the time of writing.

Enable this method by the setting provided in the config class:

```
config.emoji_verify = True
```

Your bot now listens for incoming verification requests. **Because this method is interactive, you need interactive access to your bot's console!** Perform the following steps on Element Web/Desktop to verify your session and the bot's session with each other.

1. In Element Web or Desktop, open the bot user's info and click on "X session(s)" - NOT on "Verify".
2. The bot's current sessions named "Bot Client using Simple-Matrix-Bot-Lib" will be listed with gray shields next to them.
3. Click the session with the correct Session ID printed by your bot during startup, then select "Interactively verify by Emoji".
4. Compare the Emoji shown by Element and printed by your bot.
5. Select the appropriate button and enter the appropriate letter into the console depending on whether the Emoji match.

4.4 Usage of Creds class

The Creds class is a class that handles login credentials. The source is located at simplematrixbotlib/auth.py.

4.4.1 Creating an instance of the Creds class

An instance can be created using the following python code.

```
creds = botlib.Creds(
    homeserver="https://example.org",
    username="username",
    password="password",
    session_stored_file="session.txt"
    )
```

or

```
creds = botlib.Creds(
    homeserver="https://example.org",
    username="username",
    login_token="MDA..gZ2",
    session_stored_file="session.txt"
    )
```

or

```
creds = botlib.Creds(
    homeserver="https://example.org",
    username="username",
    access_token="syt_c2...DTJ",
    session_stored_file="session.txt"
    )
```

The homeserver and username arguments are always required. The `password` argument may be replaced by either the `login_token` argument or the `access_token` argument. The `login_token` is used with handling SSO logins (See the Matrix Docs) and can only be used to authenticate once. The `access_token` is generated by logging in using a different login method.

The optional `session_stored_file` argument is the location of a file used by the bot to store session information such as the generated access token and device name. When a `session_stored_file` is present, the Api class will prefer an existing `access_token` over a password or login token given in the Creds class automatically.

4.5 Usage of Config class

The Config class is a class that handles whether certain features are enabled or disabled. The source is located at simplematrixbotlib/config.py

4.5.1 Creating an instance of the Config class

An instance can be created using the following python code.

```
config = botlib.Config()
```

4.5.2 Built-in Values

The following Config values are may implement validation logic. They can be interacted with as if they were public member variables:

```
config.join_on_invite = True
print(config.join_on_invite)
```

See also: *Additional Methods*

`join_on_invite`

Boolean: whether the bot accepts all invites automatically.

encryption_enabled

Boolean: whether to enable encryption. Other settings depend on the value of this setting, e.g. setting encryption to false will also set `emoji_verify` to false. Encryption requires additional encryption-specific dependencies to be installed.

emoji_verify

Boolean: whether the bot's built-in emoji verification callback should be enabled. Requires encryption to be enabled. Learn more at *Interactive SAS verification using Emoji*.

ignore_unverified_devices

Boolean: whether to automatically ignore unverified devices in order to send encrypted messages to them without verifying. See *Encryption Configuration Options* to learn more about the different trust states, including ignoring. When encryption is not enabled, messages will always be sent to all devices.

store_path

String: path in the filesystem where the crypto-session gets stored. Can be relative (`./store/`) or absolute (`/home/example`). Needs to be readable and writable by the bot.

allowlist

List of strings: Regular expressions of matrix user IDs who are allowed to send commands to the bot. Defaults to allow everyone on the bot's homeserver. If the list is non-empty, user IDs that are not on it are blocked. Thus to allow anybody, set it to `[]`. You can check using `Match.is_from_allowed_user` if the sender of a command is allowed to use the bot and act accordingly. **IMPORTANT**: This only applies to `Match.is_from_allowed_user`!

blocklist

List of strings: Regular expressions of matrix user IDs who are not allowed to send commands to the bot. Defaults to empty, blocking nobody. Blocks user IDs on it if non-empty, even overriding the `allowlist`. For example: this way it is possible to allow all users from a homeserver, but block single ones. You can check using `Match.is_from_allowed_user` if the sender of a command is allowed to use the bot and act accordingly. **IMPORTANT**: This only applies to `Match.is_from_allowed_user`!

4.5.3 Additional methods

Configuration settings can additionally be manipulated in special ways using the following methods.

Method	Description
`add_allowlist(list)`	Merge this list into the list of users who are allowed to interact with the bot.
`remove_allowlist(list)`	Subtract this list from the list of users who are allowed to interact with the bot.
`add_blocklist(list)`	Merge this list into the list of users who are disallowed to interact with the bot.
`remove_blocklist(list)`	Subtract this list from the list of users who are disallowed to interact with the bot.

4.5.4 Loading and saving config values

Configuration settings can be set to values read from a file using the following python code.

```
config.load_toml("config.toml")
```

Depending on the file format, a specific method may be used for reading the file. A table of the appropriate method to use for each format is shown below.

Format	Method
TOML	`load_toml(file)`

Similarly, settings can be written to file after manipulating them at runtime.

```
config.save_toml("config.toml")
```

Format	Method
TOML	`save_toml(file)`

Example configuration files for each file format can be under the examples section of the documentation. An example of a toml config file can be found here.

4.5.5 Extending the Config class with custom settings

The Config class is designed to be easily extensible with any custom field you may need for your specific bot. This allows you to simply add your settings to the existing bot config file, right next to the other settings.

Extending the Config class is done by deriving your own Config class from it and adding your new values as well as functions if required.

First create your new class, called `MyConfig` for example, based on Config. Because Config is a dataclass, you need to add the dataclass decorator to your class as well. Then add your new custom field by adding an attribute to your class, and make sure to add a type annotation so it gets properly picked up as a dataclass field. When creating a simple attribute like that, its name may not start with an underscore _ in order to make it save and load properly.

```
import simplematrixbotlib as botlib
from dataclasses import dataclass


@dataclass
class MyConfig(botlib.Config):
    custom_setting: str = "My setting"
```

It is possible to add additional logic to your new setting by adding getter and setter methods. Most built-in settings are implemented this way similar to the example below.

Create your custom field by adding a "private" attribute to your class, i.e. its name starts with an underscore _. Then add a getter method by using the `@property` decorator, and a setter method using the setter decorator `@name-of-your-field-without-underscore.setter`. The name for each function is also the name of your field without the leading underscore. Your setting can then be accessed publicly by using the name without underscore, similar to the default Config settings. The functions for loading and saving your config file will automatically use the getter and setter methods and apply any logic in them.

If you wanted, you could add additional methods, e.g. to implement behavior like that of ``*add_allowlist()*` etc. <#additional-methods>`_ Take a look at the Config class source code if you are unsure how to do this.

```
import simplematrixbotlib as botlib
from dataclasses import dataclass


@dataclass
class MyConfig(botlib.Config):
    _my_setting: str = "Hello"

    @property
    def my_setting(self) -> str:
        return self._my_setting

    @my_setting.setter
    def my_setting(self, value: str) -> None:
        # validate(value)
        self._my_setting = value
```

Finally, use your custom Config class by instantiating it and passing the instance when creating your Bot instance.

```
config = MyConfig()
config.load_toml('config.toml')
bot = botlib.Bot(creds, config)
```

A complete example implementation of a custom Config class can be found here.

4.6 Usage of Bot class

The Bot class is a class that handles most of the functionality of a bot created with Simple-Matrix-Bot-Lib. The source is located at simplematrixbotlib/bot.py.

4.6.1 Creating an instance of the Bot class

An instance can be created using the following python code.

```
bot = botlib.Bot(
    creds=creds,
    config=config
    )
```

The creds argument is neccesary, and is an instance of the Creds class. The config argument is optional, and is an instance of the Config class.

4.6.2 Running the Bot

When the Bot is ready to be started, the run method can be used to run the Bot. An example is shown in the following python code.

```
bot.run()
```

4.7 Usage of Listener class

The Listener class is a class that is used to specify reactions to the events that occur in Matrix rooms. The source is located at simplematrixbotlib/listener.py

4.7.1 Accessing a Listener instance

An instance of the Listener class is automatically created when an instance of the Bot class is created. An example is shown in the following python code.

```
bot = botlib.Bot(creds)
bot.listener #Instance of the Listener class
```

4.7.2 Using the on_message_event decorator

The on_message_event method of the Listener class may be used to execute actions based on messages that are sent in rooms that the bot is a member of. Example usage of on_message_event is shown in the following python code.

```
@bot.listener.on_message_event
async def example(room, message):
    print(f"A message({message.content}) was sent in a room({room.room_id}).")
```

When any message is sent, the function will be called with room as a Room object representing each room that that the bot is a member of, and message as a RoomMessage object representing the message that was sent.

4.7.3 Using the on_reaction_event decorator

The on_reaction_event decorator method of the Listener class may be used to execute actions based on reactions that are sent in rooms that the bot is a member of. Example usage of on_reaction_event is shown in the following python code.

```
@bot.listener.on_reaction_event
async def example(room, event, reaction):
    print(f"User {event.source['sender']} reacted with {reaction} to message {event.
↪source['content']['m.relates_to']['event_id']}")
```

As of the time of writing, m.reaction events are not supported via matrix-nio. To work around this, it is recommended to use the event's source via event.source as a dictionary. An example m.reaction event source is provided for convenience below:

```
{
    "events": [
        {
            "content": {
                "m.relates_to": {
                    "event_id": "$FNP1EnwKRuzH38LjuYptDSkJpzomVt3tijlBy6yfc10",
                    "key": "",
                    "rel_type": "m.annotation"
                }
            },
            "origin_server_ts": 1641348447462,
            "sender": "@krazykirby99999:matrix.org",
            "type": "m.reaction",
            "unsigned": {
                "age": 341
            },
            "event_id": "$rGchfmQQmt2NxnlJ88HzWdVTIW-cfo-DGZFUYbqihBI"
        }
    ]
}
```


4.7.4 Using the on_custom_event decorator

The on_custom_event method of the Listener class may be used to execute actions based on any event that is sent in rooms that the bot is a member of. Example usage of on_custom_event is shown in the following python code.

```
import nio

@bot.listener.on_custom_event(nio.InviteMemberEvent)
async def example(room, event):
    if event.membership == "join":
        print(f"A user joined the room({room.room_id}).")
    if event.membership == "leave":
        print(f"A user left the room({room.room_id}).")
```

The on_custom_event method is almost identical to the on_message_event method. on_custom_event takes an argument that allows the developer to specify the event type for the Bot to respond to. Information on events can be found in the matrix-nio docs.

Using the on_startup decorator

The on_startup method of the Listener class may be used to execute actions upon the starting of the Bot. Example usage of the on_startup method is show in the following python code.

```
@bot.listener.on_startup
async def room_joined(room_id):
    print(f"This account is a member of a room with the id {room_id}")
```

When the bot is run, for each room that the Bot is a member of, the function will be called with room_id as a string that corresponds to the room_id of the room.

4.8 Usage of Match and MessageMatch classes

4.8.1 How to use the Match class

The Match class is a class that handles matching/filtering of the content of events. The source is located at simplematrixbotlib/match.py

Creating an instance of the Match class

An instance can be created using the following python code.

```
match = botlib.Match(
    room=room,
    event=event,
    bot=bot
)
```

The room, event, and bot arguments are neccesary. The room and event arguments should be the same as the arguments of the handler function. The bot argument should be the same as the instance of the Bot class. This class is intended to be used with non-message events, as the MessageMatch class is a child class of this class, and has message-specific methods. A list of methods for the Match class is shown below.

Method	Explanation
`Match.is_from_user_id(userid)`	Returns True if the userid argument matches the event's sender.
`Match.is_not_from_this_bot()`	Returns True if the event is not sent by this bot.

Example:

```
bot.listener.on_message_event
async def example(room, event):
    match = botlib.Match(room, event, bot)
    if match.is_not_from_this_bot():
        print(f"A user sent a message in room {room.room_id}")
```

4.8.2 How to use the MessageMatch class

The MessageMatch class is a class that handles matching/filtering of message events. It is a subclass of the Match class, and thus methods of the Match class can also be used with the MessageMatch class. The source is located at simplematrixbotlib/match.py

Creating an instance of the MessageMatch class

An instance can be created using the following python code.

```
match = botlib.MessageMatch(
    room=room,
    event=event,
    bot=bot,
    prefix="/"
)
```

The room, event, and bot arguments are necessary. The bot argument is an instance of the Bot class. The room and event arguments are the same as the arguments specified when creating a handler function to be used with the Listener.on_message_event method. The prefix argument is usually used as the beginning of messages that are intended to be commands, usually "!", "/" or another short string. An example handler function that uses MessageMatch is shown in the following python code.

```
bot.listener.on_message_event
async def example(room, message):
    match = botlib.MessageMatch(room, message, bot, "!")
    if match.command("help") and match.prefix(): # Matches any message that begins with
↪"!help "
        #Respond to help command
```

As said earlier, the prefix argument is optional. An example handler function without it is shown in the following python code.

```
bot.listener.on_message_event
async def example(room, message):
    match = botlib.MessageMatch(room, message, bot)
    if match.command("help"): # Matches any message that begins with "help "
        #Respond to help command
```

A list of methods for the Match class is shown below. *Methods from the Match class* can also be used with the MessageMatch class.

List of Methods:

Method	Explanation
`MessageMatch.command()` or `MessageMatch.command(command)`	The "command" is the beginning of messages that are intended to be commands, but after the prefix; e.g. "help". Returns the command if the command argument is empty. Returns True if the command argument is equivalent to the command.
`MessageMatch.prefix()`	Returns True if the message begins with the prefix specified during the initialization of the instance of the MessageMatch class. Returns True if no prefix was specified during the initialization.
`MessageMatch.args()`	Returns a list of strings; each string is part of the message separated by a space, with the exception of the part of the message before the first space (the prefix and command). Returns an empty list if it is a single-word command.
`MessageMatch.contains(string)`	Returns True if the message contains the value specified in the string argument.

4.9 Usage of Api class

The Api class is a class that is used to simplify interaction with the matrix-nio library that Simple-Matrix-Bot-Lib is built upon. The source is located at simplematrixbotlib/api.py

4.9.1 Accessing an Api instance

An instance of the Api class is automatically created when an instance of the Bot class is created. An example is shown in the following python code.

```
bot = botlib.Bot(creds)
bot.api #Instance of the Api class
```

4.9.2 Using the send_text_message method

The send_text_message method of the Api class can be used to send text messages in Matrix rooms. An example is shown in the following python code.

```
async def example(room, message):
    match = botlib.MessageMatch(room, message, bot)
    example_message = "Hello World"
    if match.is_not_from_this_bot():
        await bot.api.send_text_message(
            room_id=room.room_id,
            message=example_message,
            msgtype="m.notice")
```

The first two arguments are required. The room_id argument is the id of the destination room. The message argument is the string that is to be sent as a message. The msgtype argument can be “m.text” (default) or “m.notice”.

4.9.3 Using the send_image_message method

The send_image_message method of the Api class can be used to send image messages in Matrix rooms. An example is shown in the following python code.

```
async def example(room, message):
    match = botlib.MessageMatch(room, message, bot)
    example_image="./img/example.png"
    if match.is_not_from_this_bot():
        await bot.api.send_image_message(
            room_id=room.room_id,
            image_filepath=example_image)
```

Both arguments are required. The room_id argument is the id of the destination room. The image_filepath argument is a string that is the path to the image file that is to be sent as a message.

4.9.4 Using the send_video_message method

The send_video_message method of the Api class can be used to send video messages in Matrix rooms. An example is shown in the following python code.

```
async def example(room, message):
    match = botlib.MessageMatch(room, message, bot)
    example_video="./videos/example.mp4"
    if match.is_not_from_this_bot():
        await bot.api.send_video_message(
            room_id=room.room_id,
            video_filepath=example_video)
```

Both arguments are required. The room_id argument is the id of the destination room. The video_filepath argument is a string that is the path to the video file that is to be sent as a message.

4.9.5 Using the send_markdown_message method

The send_markdown_message method of the Api class can be used to send markdown messages in Matrix rooms. An example is shown in the following python code.

```
async def example(room, message):
    match = botlib.MessageMatch(room, message, bot)
    example_markdown = "# Hello World from [simplematrixbotlib](https://github.com/
↪KrazyKirby99999/simple-matrix-bot-lib)!"
    if match.is_not_from_this_bot():
        await bot.api.send_markdown_message(
            room_id=room.room_id,
            message=example_markdown,
            msgtype="m.notice")
```

The first two arguments are required. The room_id argument is the id of the destination room. The message argument is the string with markdown syntax that is to be sent as a message. The msgtype argument can be “m.text” (default) or “m.notice”.