Err Documentation

Release 6.1.1

Guillaume Binet, Tali Davidovich Petrover and Nick Groenen

Jul 07, 2019

Contents

Errbot is a chatbot, a daemon that connects to your favorite chat service and brings your tools into the conversation.

The goal of the project is to make it easy for you to write your own plugins so you can make it do whatever you want: a deployment, retrieving some information online, trigger a tool via an API, troll a co-worker,. . .

Errbot is being used in a lot of different contexts: chatops (tools for devops), online gaming chatrooms like EVE, video streaming chatrooms like livecoding.tv, home security, etc.

Screenshots

Simple to build upon

Extending Errbot and adding your own commands can be done by creating a plugin, which is simply a class derived from *BotPlugin*. The docstrings will be automatically reused by the *!help* command:

```python
from errbot import BotPlugin, botcmd

class HelloWorld(BotPlugin):
    """Example 'Hello, world!' plugin for Errbot."""

    @botcmd
    def hello(self, msg, args):
        """Say hello to the world."""
        return "Hello, world!"
```

Once you said "!hello" in your chatroom, the bot will answer "Hello, world!".

Batteries included

We aim to give you all the tools you need to build a customized bot safely, without having to worry about basic functionality. As such, Errbot comes with a wealth of features out of the box.

3.1 Multiple server backends

Errbot has support for a number of different networks and is architectured in a way that makes it easy to write new backends in order to support more. Currently, the following networks are supported:

- XMPP *(Any standards-compliant XMPP/Jabber server should work - Google Talk/Hangouts included)*

- Hipchat

- IRC

- Slack

- Telegram

- Bot Framework (maintained separately)

- CampFire (maintained separately)

- **'Cisco Webex Teams'_** (maintained separately)

- Discord (maintained separately)

- Gitter (maintained separately)

- Matrix (maintained separately)

- Mattermost (maintained separately)

- Skype (maintained separately)

- Tox (maintained separately)

- VK (maintained separately)

• Zulip (maintained separately)

3.2 Core features

• Multi User Chatroom (MUC) support

• A dynamic plugin architecture: Bot admins can install/uninstall/update/enable/disable plugins dynamically just by chatting with the bot

• Advanced security/access control features (see below)

• A *!help* command that dynamically generates documentation for commands using the docstrings in the plugin source code

• A per-user command history system where users can recall previous commands

• The ability to proxy and route one-to-one messages to MUC so it can enable simpler XMPP notifiers to be MUC compatible (for example the Jira XMPP notifier)

3.3 Built-in administration and security

• Can be setup so a restricted list of people have administrative rights

• Fine-grained *access controls* may be defined which allow all or just specific commands to be limited to specific users and/or rooms

• Plugins may be hosted publicly or privately and dynamically installed (by admins) via their Git url

• Plugins can be configured directly from chat (no need to change setup files for every plugin)

• Configs can be exported and imported again with two commands (!export and !import respectively)

• Technical logs can be logged to file, inspected from the chat or optionally *logged to Sentry*

3.4 Extensive plugin framework

• Hooks and callbacks for various types of events, such as `callback_connect()` for when the bot has connected or `callback_message()` for when a message is received.

• Local text and graphical consoles for easy testing and development

• Plugins get out of the box support for subcommands

• We provide an automatic persistence store per plugin

• There's really simple webhooks integration

• As well as a polling framework for plugins

• An easy configuration framework

• A test backend for unittests for plugins which can make assertions about issued commands and their responses

• And a templating framework to display fancy HTML messages. Automatic conversion from HTML to plaintext when the backend doesn't support HTML means you don't have to make separate text and HTML versions of your command output yourself

Sharing

One of the main goals of Errbot is to make it easy to share your plugin with others as well.

Errbot features a built-in *repositories command* (*!repos*) which can be used to install, uninstall and update plugins made available by the community. Making your plugin available through this command only requires you to publish it as a publicly available Git repository.

You may also discover plugins from the community on our plugin list that we update from plugins found on github.

Community

Errbot has a Google plus community, which is the best place to discuss anything related to Errbot as well as promote your own creations ! This is also the place where you will find announcements of new versions and other news related to the project.

You can also interact directly with the community online from the "Open Chat" button at the bottom of this page. Don't be shy and feel free to ask any question there, we are more than happy to help you.

If you think you hit a bug or the documentation is not clear enough, you can open an issue or even better, open a pull request.

User guide

6.1 Setup

6.1.1 Prerequisites

Errbot runs under Python 3.3+ on Linux, Windows and Mac.

6.1.2 Installation

Option 1: Use the package manager of your distribution (if available)

On some distributions, Errbot is also available as a package via your usual package manager. In these cases, it is generally recommended to use your distribution's package instead of installing from PyPi but note that the version packaged with your distribution may be a few versions behind.

Example of packaged versions of Errbot:

Gentoo: https://gpo.zugaina.org/net-im/errbot Arch: https://aur.archlinux.org/packages/python-err/ Docker: https://hub.docker.com/r/rroemhild/errbot/ Juju: https://jujucharms.com/u/onlineservices-charmers/errbot

Option 2: Installing Errbot in a virtualenv (preferred)

Installing into a virtualenv is **strongly** recommended. If you have virtualenv installed, you can do for example:

```
virtualenv --python `which python3` ~/.errbot-ve
~/.errbot-ve/bin/pip install errbot
```

If you have virtualenvwrapper installed it is even simpler:

```
mkvirtualenv -p `which python3` errbot-ve
pip install errbot
```

Option 3: Installing Errbot at the system level (not recommended)

Errbot may be installed directly from PyPi using pip by issuing:

```
pip3 install errbot
```

Note: Some of errbot's dependencies need to build C extensions which means you need to have development headers for some libraries installed. On Debian/Ubuntu these may be installed with *apt-get install python3-dev libssl-dev libffi-dev* Package names may differ on other OS's.

First run

You can quickly configure Errbot by first creating a working directory and calling *errbot –init*:

```
mkdir ~/errbot-root
cd ~/errbot-root
errbot --init
```

This will create a minimally working errbot in text (development) mode. You can try it right away:

```
errbot
[...]
>>>
```

>>> is a prompt, you can talk to errbot directly. You can try:

```
!tryme
!help
!about
```

6.1.3 Configuration

Once you have installed errbot and did *errbot –init*, you will have to tweak the generated *config.py* to connect to your desired chat network.

You can use `config-template.py` as a base for your *config.py*.

We'll go through the options that you absolutely must check now so that you can quickly get started and make further tweaks to the configuration later on.

Open *config.py* in your favorite editor. The first setting to check or change if *BOT_DATA_DIR* if correct. This is the directory where the bot will store configuration data.

The first setting to check or change *BOT_LOG_FILE* to be sure it point to a writeable directory on your system.

The final configuration we absolutely must do is setting up a correct *BACKEND* which is set to *Text* by *errbot –init* but you can change to the name of the chat system you want to connect to (see the template above for valid values).

You absolutely need a *BOT_IDENTITY* entry to set the credentials Errbot will use to connect to the chat system.

You can find here more details about configuring Errbot for some specific chat systems:

XMPP backend configuration

This backend lets you connect to any Jabber/XMPP server. To select this backend, set *BACKEND = 'XMPP'*.

Extra Dependencies

You need to install this dependency before using Errbot with XMPP:

```
pip install sleekxmpp pyasn1 pyasn1-modules
```

Account setup

You must manually register an XMPP account for the bot on the server you wish to use. Errbot does not support XMPP registration itself.

Configure the account by setting up *BOT_IDENTITY* as follows:

```
BOT_IDENTITY = {
    'username': 'err@server.tld',  # The JID of the user you have created for the bot
    'password': 'changeme',        # The corresponding password for this user
    # 'server': ('host.domain.tld',5222), # server override
}
```

By default errbot will query SRV records for the correct XMPP server and port, which should work with a properly configured server.

If your chosen XMPP server does not have correct SRV records setup, you can also set the *server* key to override this.

A random resource ID is assigned when errbot starts up. You may fix the resource by appending it to the user name:

```
BOT_IDENTITY = {
    'username': 'err@server.tld/resource',
...
```

Bot admins

You can set *BOT_ADMINS* to configure which XMPP users are bot administrators. For example: *BOT_ADMINS = ('gbin@someplace.com', 'zoni@somewhere.else.com')*

MUC rooms

If you want the bot to join a certain chatroom when it starts up then set *CHATROOM_PRESENCE* with a list of MUCs to join. For example: *CHATROOM_PRESENCE = ('err@conference.server.tld',)*

Note: don't omit the comma under any circumstance!

You can configure the username errbot should use in chatrooms by setting *CHATROOM_FN*.

IRC backend configuration

This backend lets you connect to any IRC server. To select this backend, set *BACKEND = 'IRC'*.

Extra Dependencies

You need to install this dependency before using Errbot with IRC:

```
pip install irc
```

Account setup

Configure the account by setting up *BOT_IDENTITY* as follows:

```
BOT_IDENTITY = {
    'nickname' : 'err-chatbot',
    # 'username' : 'err-chatbot',    # optional, defaults to nickname if omitted
    # 'password' : None,             # optional
    'server' : 'irc.freenode.net',
    # 'port': 6667,                  # optional
    # 'ssl': False,                  # optional
    # 'ipv6': False,                 # optional
    # 'nickserv_password': None,     # optional

    ## Optional: Specify an IP address or hostname (vhost), and a
    ## port, to use when making the connection. Leave port at 0
    ## if you have no source port preference.
    ##    example: 'bind_address': ('my-errbot.io', 0)
    # 'bind_address': ('localhost', 0),
}
```

You will at a minimum need to set the correct values for *nickname* and *server* above. The rest of the options can be left commented, but you may wish to set some of them.

Bot admins

You can set *BOT_ADMINS* to configure which IRC users are bot administrators. For example: *BOT_ADMINS =* *('gbin!gbin@*', '*!*@trusted.host.com')*

Note: The default syntax for users on IRC is *{nick}!{user}@{host}* but this can be changed by adjusting the *IRC_ACL_PATTERN* setting.

Channels

If you want the bot to join a certain channel when it starts up then set *CHATROOM_PRESENCE* with a list of channels to join. For example: *CHATROOM_PRESENCE = ('#errbotio',)*

Note: You may leave the value for *CHATROOM_FN* at its default as it is ignored by this backend.

Flood protection

Many IRC servers have flood protection enabled, which means the bot will get kicked out of a channel when sending too many messages in too short a time.

Errbot has a built-in message ratelimiter to avoid this situation. You can enable it by setting *IRC_CHANNEL_RATE* and *IRC_PRIVATE_RATE* to ratelimit channel and private messages, respectively.

The value for these options is a (floating-point) number of seconds to wait between each message it sends.

Rejoin on kick/disconnect

Errbot won't rejoin a channel by default when getting kicked out of one. If you want the bot to rejoin channels on kick, you can set *IRC_RECONNECT_ON_KICK = 5* (to join again after waiting 5 seconds).

Similarly, to rejoin channels after being disconnected from the server you may set *IRC_RECONNECT_ON_DISCONNECT = 5*.

HipChat backend configuration

This backend lets you connect to the HipChat messaging service. To select this backend, set *BACKEND = 'Hipchat'*.

Extra Dependencies

You need to install this dependency before using Errbot with Hipchat:

```
pip install sleekxmpp pyasn1 pyasn1-modules hypchat
```

Account setup

You will first need to create a regular user account for the bot to use. Once you have an account for errbot to use, login at HipChat and go into the account settings for the user.

You will need to create an API token under **API access**. Make sure it has all available scopes otherwise some functionality will be unavailable, which may prevent the bot from working correctly at all.

With the API token created, continue on to **XMPP/Jabber info**. You will be needing the *Jabber ID* which is listed here.

You can now configure the account by setting up *BOT_IDENTITY* as follows:

```
BOT_IDENTITY = {
    'username' : '12345_123456@chat.hipchat.com',
    'password' : 'changeme',
    # Group admins can create/view tokens on the settings page after logging
    # in on HipChat's website
    'token'    : 'ed4b74d62833267d98aa99f312ff04',
    # If you're using HipChat server (self-hosted HipChat) then you should set
    # the endpoint below. If you don't use HipChat server but use the hosted version
    # of HipChat then you may leave this commented out.
    # 'endpoint' : 'https://api.hipchat.com',
    # If your self-hosted Hipchat server is using SSL, and your certificate
    # is self-signed, set verify to False or hypchat will fail
    # 'verify': False,
```

Bot admins

You can set *BOT_ADMINS* to configure which Hipchat users are bot administrators. Make sure to include the @ sign. For example: *BOT_ADMINS = ('@gbin', '@zoni')*

Rooms

You can let the bot join rooms (that it has access to) by setting up *CHATROOM_PRESENCE*. For example: *CHAT-ROOM_PRESENCE = ('General', 'Another room')*

You must also set the correct value for *CHATROOM_FN*. This **must** be set to the value of *Room nickname* which can be found in the HipChat account settings under **XMPP/Jabber info**.

@mentions

To make the bot respond when it is mentioned (such as with *"@errbot status"*) we recommend also setting *BOT_ALT_PREFIXES = ('@errbot',)* (assuming *errbot* is the username of the account you're using for the bot).

Slack backend configuration

This backend lets you connect to the Slack messaging service. To select this backend, set *BACKEND = 'Slack'*.

Extra Dependencies

You need to install this dependency before using Errbot with Slack:

```
pip install slackclient
```

Account setup

You will need to have an account at Slack for the bot to use, either a bot account (recommended) or a regular user account.

We will assume you're using a bot account for errbot, which may be created here. Make note of the **API Token** you receive as you will need it next.

With the bot account created on Slack, you may configure the account in errbot by setting up *BOT_IDENTITY* as follows:

```
BOT_IDENTITY = {
    'token': 'xoxb-4426949411-aEM7...',
}
```

Proxy setup

In case you need to use a Proxy to connect to Slack, you can set the proxies with the token config.

> **BOT_IDENTITY = {** 'token': 'xoxb-4426949411-aEM7...', 'proxies': {'http': 'some-http-proxy',
> 'https': 'some-https-proxy'}
>
> }

Bot admins

You can set *BOT_ADMINS* to configure which Slack users are bot administrators. Make sure to include the @ sign:

```
BOT_ADMINS = ('@gbin', '@zoni')
```

Bot mentions using @

To enable using the bot's name in *BOT_ALT_PREFIXES* for @mentions in Slack, simply add the bot's name as follows:

```
BOT_ALT_PREFIXES = ('@botname',)
```

Channels/groups

If you're using a bot account you should set *CHATROOM_PRESENCE = ()*. Bot accounts on Slack are not allowed to join/leave channels on their own (they must be invited by a user instead) so having any rooms setup in *CHATROOM_PRESENCE* will result in an error.

If you are using a regular user account for the bot then you can set *CHATROOM_PRESENCE* to a list of channels and groups to join.

Note: You may leave the value for *CHATROOM_FN* at its default as it is ignored by this backend.

Telegram backend configuration

This backend lets you connect to Telegram Messenger. To select this backend, set *BACKEND = 'Telegram'*.

Extra Dependencies

You need to install this dependency before using Errbot with Telegram:

```
pip install python-telegram-bot
```

Account setup

You will first need to create a bot account on Telegram for errbot to use. You can do this by talking to @BotFather (see also: BotFather). Make sure you take note of the token you receive, you'll need it later.

Once you have created a bot account on Telegram you may configure the account in errbot by setting up *BOT_IDENTITY* as follows:

```
BOT_IDENTITY = {
    'token': '103419016:AAbcd1234...',
}
```

Bot admins

You can setup *BOT_ADMINS* to designate which users are bot admins, but on Telegram this is a little more difficult to do. In order to configure a user here you will have to obtain their user ID.

The easiest way to do this is to start the bot with no *BOT_ADMINS* defined. Then, have the user for which you want to obtain the user ID message the bot and send it the *!whoami* command.

This will print some info about the user, including the following: *string representation is '123669037'*. It is this number that needs to be filled in for *BOT_ADMINS*. For example: *BOT_ADMINS = (123669037,)*

Rooms

Telegram does not expose any room management to bots. As a group admin, you will have to add a bot to a groupchat at which point it will automatically join.

By default the bot will not receive any messages which makes interacting with it in a groupchat difficult.

To give the bot access to all messages in a groupchat, you can use the */setprivacy* command when talking to @BotFather.

Note: Because Telegram does not support room management, you must set *CHATROOM_PRESENCE = ()* otherwise you will see errors.

Slash commands

Telegram treats messages which start with a / differently, which is designed specifically for interacting with bots.

We therefor suggest setting *BOT_PREFIX = '/'* to take advantage of this.

6.1.4 Starting the daemon

The first time you start Errbot, it is recommended to run it in foreground mode. This can be done with:

```
errbot
```

If you installed errbot into a virtualenv (as recommended), call it by prefixing the virtualenv *bin/* directory:

```
/path/to/my/virtualenv/bin/errbot
```

Please pass -h or –help to errbot to get a list of supported parameters. Depending on your situation, you may need to pass –config (or -c) pointing to the directory holding your *config.py* when starting Errbot.

If all that worked out, you can now use the -d (or –daemon) parameter to run it in a detached mode:

```
errbot --daemon
```

If you are going to run your bot all the time then using some process control system such as supervisor is highly recommended. Installing and configuring such a system is outside the scope of this document, however, we do provide some sample daemon configurations below.

Note: There are two ways to gracefully shut down a running bot.

You can use the `!shutdown` command to do so via chat or you can send a *SIGINT* signal to the errbot process to do so from the commandline

If you're running errbot in the foreground then pressing Ctrl+C is equivalent to sending *SIGINT*.

Daemon Configurations

These are a few example configurations using common init daemons:

supervisord (*/etc/supervisor/conf.d/errbot.conf*)

```
[program:errbot]
command = /path/to/errbot/virtualenv/bin/errbot --config /path/to/errbot/config.py
user = errbot
stdout_logfile = /var/log/supervisor/errbot.log
stderr_logfile = NONE
redirect_stderr = true
directory = /path/to/errbot/
startsecs = 3
stopsignal = INT
environment = LC_ALL="en_US.UTF-8"
```

systemd (*/etc/systemd/system/errbot.service*)

```
[Unit]
Description=Start Errbot chatbot
After=network.service

[Service]
Environment="LC_ALL=en_US.UTF-8"
Environment="PATH=/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/path/to/errbot/
↪virtualenv/bin"
ExecStart=/path/to/errbot/virtualenv/bin/errbot --config /path/to/errbot/config.py
WorkingDirectory=/path/to/errbot/
User=errbot
Restart=always
KillSignal=SIGINT

[Install]
WantedBy=multi-user.target
```

Note: Running errbot within a daemon process can have security implications if the daemon is started with an account containing elevated privileges. We encourage errbot **not** be run under a *root* or *administrator* account but under a non-privileged account. The command below creates a non-privileged *errbot* account on Linux:

```
$ useradd --no-create-home --no-user-group -g nogroup -s /bin/false errbot
```

6.1.5 Upgrading

Errbot comes bundled with a plugin which automatically performs a periodic update check. Whenever there is a new release on PyPI, this plugin will notify the users set in *BOT_ADMINS* about the new version.

Assuming you originally installed errbot using pip (see *installation*), you can upgrade errbot in much the same way. If you used a virtualenv:

```
/path/to/my/virtualenv/bin/pip install --upgrade errbot
```

Or if you used pip without virtualenv:

```
pip install --upgrade errbot
```

It's recommended that you review the changelog before performing an upgrade in case backwards-incompatible changes have been introduced in the new version. The changelog for the release you will be installing can always be found on PyPI.

6.1.6 Provisioning (advanced)

See the provisioning documentation

6.2 Administration

This document describes how to configure, administer and interact with errbot.

6.2.1 Configuration

There is a split between two types of configuration within errbot. On the one hand there is "setup" information, such as the (chat network) backend to use, storage selection and other settings related to how errbot should run. These settings are all configured through the *config.py* configuration file as explained in *configuration*.

The other type of configuration is the "runtime" configuration such as the plugin settings. Plugins can be dynamically configured through chatting with the bot by using the `!plugin config <plugin name>` command.

There are a few other commands which adjust the runtime configuration, such as the `!plugin blacklist <plugin>` command to unload and blacklist a specific plugin.

You can view a list of all these commands and their help documentation by using the built-in help function.

The built-in help function

To get a list of all available commands, you can issue:

```
!help
```

If you just wish to know more about a specific command you can issue:

```
!help <command>
```

6.2.2 Installing plugins

Errbot plugins are typically published to and installed from GitHub. We periodically crawl GitHub for errbot plugin repositories and publish the results for people to browse.

You can have your bot display the same list of repos by issuing:

```
!repos
```

Searching can be done by specifying one or more keywords, for example:

```
!repos search hello
```

To install a plugin from the list, issue:

```
!repos install <name of plugin>
```

You aren't limited to installing public plugins though. You can install plugins from any git repository you have access to, whether public or private, hosted on GitHub, BitBucket or elsewhere. The *!repos install* command can take any git URI as argument.

If you're unhappy with a plugin and no longer want it, you can always uninstall a plugin again with:

```
!repos uninstall <plugin>
```

You will probably also want to update your plugins periodically. This can be done with:

```
!repos update all
```

Dependencies

Please pay attention when you install a plugin as it may have additional dependencies. If the plugin contains a *requirements.txt* file then Errbot will automatically check the requirements listed within and warn you when you are missing any.

Additionally, if you set `AUTOINSTALL_DEPS` to `True` in your **config.py**, Errbot will use pip to install any missing dependencies automatically. If you have installed Errbot in a virtualenv, this will run the equivalent of `pip install -r requirements.txt`. If no virtualenv is detected, the equivalent of `pip install --user -r requirements.txt` is used to ensure the package(s) is/are only installed for the user running Err.

Extra plugin directory

Plugins installed via the `!repos` command are managed by errbot itself and stored inside the *BOT_DATA_DIR* you set in *config.py*. If you want to manage your plugins manually for any reason then errbot allows you to load additional plugins from a directory you specify. You can do so by specifying the setting *BOT_EXTRA_PLUGIN_DIR* in your *config.py* file. See the `config-template.py` file for more details.

6.2.3 Disabling plugins

You have a number of options available to you if you need to disable a plugin for any reason. Plugins can be temporarily disabled by using the `!plugin deactivate <plugin name>` command, which deactivates the plugin until the bot is restarted (or activated again via `!plugin activate <plugin name>`.

If you want to prevent a plugin from being loaded at all during bot startup, the `!plugin blacklist <plugin name>` command may be used.

It's also possible to strip errbot down even further by disabling some of its core plugins which are otherwise activated by default. You may for example want to this if you're building a very specialized bot for a specific purpose.

Disabling core plugins can be done by setting the *CORE_PLUGINS* setting in *config.py*. For example, setting *CORE_PLUGINS = ()* would disable all of the core plugins which even removes the plugin and repository management commands described above.

6.2.4 Restricting access

Errbot features a number of options to limit and restrict access to commands of your bot. All of these are configured through the *config.py* file as explained in *configuration*.

The first of these is *BOT_ADMINS*, which sets up the administrators for your bot. Some commands are hardcoded to be admin-only so the people listed here will be given access to those commands (the users listed here will also receive warning messages generated by the `warn_admins()` plugin function).

More advanced access controls can be set up using the *ACCESS_CONTROLS* and *ACCESS_CONTROLS_DEFAULT* options which allow you to set up sophisticated rules. The example `config.py` file contains more information about the format of these options.

If you don't like encoding access controls into the config file, a member of the errbot community has also created a dynamic ACL module which can be administered through chat commands instead.

Note: Different backends have different formats to identify users. Refer to the backend-specific notes at the end of the *configuration* chapter to see which format you should use.

Command filters

If our built-in access controls don't fit your needs, you can always create your own easily using *command filters*. Command filters are functions which are called automatically by errbot whenever a user executes a command. They allow the command to be allowed, blocked or even modified based on logic you implement yourself. In fact, the restrictions enforced by *BOT_ADMINS* and *ACCESS_CONTROLS* above are implemented using a command filter themselves so they can serve as a good `example` (be sure to view the module source).

You can add command filters to your bot by including them as part of any regular errbot plugin, it will find and register them automatically when your plugin is loaded. Any method in your plugin which is decorated by `cmdfilter()` will then act as a command filter.

6.3 Plugin development

Plugins form the heart of Errbot. From the ground up, it is designed to be extended entirely through plugins. In this guide we will explain the basics of writing simple plugins, which we then follow up on further with sets of recipes on a range of topics describing how to handle more advanced use-cases.

6.3.1 Intro

Before we get started I would like to make sure you have all the necessary requirements installed and give you an idea of which knowledge you should already possess in order to follow along without difficulty.

Requirements

This guide assumes that you've already installed and configured Errbot and have successfully managed to connect it to a chatting service server. See *Setup* if you have not yet managed to install or start Errbot.

Prior knowledge

You can most definitely work with Errbot if you only have basic Python knowledge, but you should know about data structures such as dictionaries, tuples and lists, know what docstrings are and have a basic understanding of decorators.

6.3.2 Development environment

Before we dive in and start writing our very first plugin, I'd like to take a moment to show you some tools and features which help facilitate the development process.

Loading plugins from a local directory

Normally, you manage and install plugins through the built-in *!repos* command. This installs plugins by cloning them via git, and allows updating of them through the *!repos update* command.

During development however, it would be easier if you could load your plugin(s) directly, without having to commit them to a Git repository and instructing Errbot to pull them down.
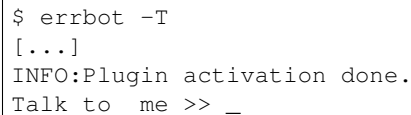
This can be achieved through the *BOT_EXTRA_PLUGIN_DIR* setting in the *config.py* configuration file. If you set a path here pointing to a directory on your local machine, Errbot will (recursively) scan that directory for plugins and attempt to load any it may find.

Local test mode

You can run Errbot in a local single-user mode that does not require any server connection by passing in the `--text` (or `-T`) option flag when starting the bot.

In this mode, a very minimal back-end is used which you can interact with directly on the command-line. It looks like this:

```
$ errbot -T
[...]
INFO:Plugin activation done.
Talk to  me >> _
```

If you have PySide installed, you can also run this same mode in a separate window using `--graphic` (or `-G`) instead of `--text`. The advantage of this is that you do not have the bot's responses and log information mixed up together in the same window.

Plugin scaffolding

Plugins consist of two parts, a special *.plug* file and one or more Python (*.py*) files containing the actual code of your plugin (both of these are explained in-depth in the next section). Errbot can automatically generate these files for you so that you do not have to write boilerplate code by hand.

To create a new plugin, run *errbot –new-plugin* (optionally specifying a directory where to create the new plugin - it will use the current directory by default). It will ask you a few questions such as the name for your plugin, a description and which versions of errbot it will work with and generate a plugin skeleton from this with all the information filled out automatically for you.

6.3.3 Hello, world!

On the *homepage*, we showed you the following *"Hello world!"* plugin as an example:

```python
from errbot import BotPlugin, botcmd


class HelloWorld(BotPlugin):
    """Example 'Hello, world!' plugin for Errbot"""

    @botcmd
    def hello(self, msg, args):
        """Say hello to the world"""
        return "Hello, world!"
```

In this chapter, you will learn exactly how this plugin works.

I will assume you've configured the *BOT_EXTRA_PLUGIN_DIR* as described in the previous chapter. To get started, create a new, empty directory named *HelloWorld* inside this directory.

Create a new file called *helloworld.py* inside the *HelloWorld* directory you just created. This file contains all the logic for your plugin, so copy and paste the above example code into it.

Anatomy of a BotPlugin

Although this plugin is only 9 lines long, there is already a lot of interesting stuff going on here. Lets go through it step by step.

```python
from errbot import BotPlugin, botcmd
```

This should be pretty self-explanatory. Here we import the `BotPlugin` class and the `botcmd()` decorator. These let us build a class that can be loaded as a plugin and allow us to mark methods of that class as bot commands.

```python
class HelloWorld(BotPlugin):
    """Example 'Hello, world!' plugin for Errbot"""
```

Here we define the class that makes up our plugin. The name of your class, *HelloWorld* in this case, is what will make up the name of your plugin. This name will be used in commands such as *!status*, *!plugin load* and *!plugin unload*

The class' docstring is used to automatically populate the built-in command documentation. It will be visible when issuing the *!help* command.

> **Warning:** A plugin should only ever contain a single class inheriting from `BotPlugin`

```python
@botcmd
def hello(self, msg, args):
    """Say hello to the world"""
    return "Hello, world!"
```

This method, *hello*, is turned into a bot command which can be executed because it is decorated with the `botcmd()` decorator. Just as with the class docstring above, the docstring here is used to populate the *!help* command.

The name of the method, *hello* in this case, will be used as the name of the command. That means this method creates the *!hello* command.

Note: The method name must comply with the usual Python naming conventions for identifiers , that is, they may not begin with a digit (like `911` but only with a letter or underscore, so `_911` would work) and cannot be any of the reserved keywords such as `pass` (instead use `password`) etc.

Note: Should multiple plugins define the same command, they will be dynamically renamed (by prefixing them with the plugin name) so that they no longer clash with each other.

If we look at the function definition, we see it takes two parameters, *msg* and *args*. The first is a `Message` object, which represents the full message object received by Errbot. The second is a string (or a list, if using the *split_args_with* parameter of `botcmd()`) with the arguments passed to the command.

For example, if a user were to say *!hello Mister Errbot*, *args* would be the string *"Mister Errbot"*.

Finally, you can see we return with the string *Hello, world!*. This defines the response that Errbot should give. In this case, it makes all executions of the *!hello* command return the message *Hello, world!*.

Note: If you return *None*, Errbot will not respond with any kind of message when executing the command.

Plugin metadata

We have our plugin itself ready, but if you start the bot now, you'll see it still won't load your plugin. What gives?

As it turns out, you need to supply a file with some meta-data alongside your actual plugin file. This is a file that ends with the extension *.plug* and it is used by Errbot to identify and load plugins.

Lets go ahead and create ours. Place the following in a file called *helloworld.plug*:

```
[Core]
Name = HelloWorld
Module = helloworld

[Python]
Version = 2+

[Documentation]
Description = Example "Hello, world!" plugin
```

Note: This INI-style file is parsed using the Python configparser class. Make sure to use a valid file structure.

Lets look at what this does. We see two sections, *[Core]* , and *[Documentation]*. The *[Core]* section is what tells Errbot where it can actually find the code for this plugin.

The key *Module* should point to a module that Python can find and import. Typically, this is the name of the file you placed your code in with the *.py* suffix removed.

The key *Name* should be identical to the name you gave to the class in your plugin file, which in our case was *HelloWorld*. While these names can differ, doing so is not recommended.

Note: If you're wondering why you have to specify it when it should be the same as the class name anyway, this has to do with technical limitations that we won't go into here.

The *[Documentation]* section will be explained in more detail further on in this guide, but you should make sure to at least have the *Description* item here with a short description of your plugin.

Wrapping up

If you've followed along so far, you should now have a working *Hello, world!* plugin for Errbot. If you start your bot, it should load your plugin automatically.

You can verify this by giving the *!status* command, which should respond with something like the following:

```
Yes I am alive...
With these plugins (A=Activated, D=Deactivated, B=Blacklisted, C=Needs to be
→configured):
[A] ChatRoom
[A] HelloWorld
[A] VersionChecker
[A] Webserver
```

If you don't see your plugin listed or it shows up as unloaded, make sure to start your bot with *DEBUG*-level logging enabled and pay close attention to what it reports. You will most likely see an error being reported somewhere along the way while Errbot starts up.

Next steps

You now know enough to create very simple plugins, but we have barely scratched the surface of what Errbot can do. The rest of this guide will be a recipe-style set of topics that cover all the advanced features Errbot has to offer.

6.3.4 Advanced bot commands

Automatic argument splitting

With the *split_args_with* argument to `botcmd()`, you can specify a delimiter of the arguments and it will give you an array of strings instead of a string:

```python
@botcmd(split_args_with=None)
def action(self, mess, args):
    # if you send it !action one two three
    # args will be ['one', 'two', 'three']
```

Note: *split_args_with* behaves exactly like `str.split()`, therefore the value *None* can be used to split on any type of whitespace, such as multiple spaces, tabs, etc. This is recommended over ' ' for general use cases but you're free to use whatever argument you see fit.

Subcommands

If you put an _ in the name of the function, Errbot will create what looks like a subcommand for you. This is useful to group commands that belong to each other together.

```python
@botcmd
def basket_add(self, mess, args):
    # Will respond to !basket add
    pass


@botcmd
def basket_remove(self, mess, args):
    # Will respond to !basket remove
    pass
```

Note: It will still respond to !basket_add and !basket_remove as well.

Argparse argument splitting

With the `arg_botcmd()` decorator you can specify a command's arguments in argparse format. The decorator can be used multiple times, and each use adds a new argument to the command. The decorator can be passed any valid add_arguments() parameters.

```python
@arg_botcmd('first_name', type=str)
@arg_botcmd('--last-name', dest='last_name', type=str)
@arg_botcmd('--favorite', dest='favorite_number', type=int, default=42)
def hello(self, mess, first_name=None, last_name=None, favorite_number=None):
    # if you send it !hello Err --last-name Bot
    # first_name will be 'Err'
    # last_name will be 'Bot'
    # favorite_number will be 42
```

Note:

- An argument's *dest* parameter is used as its kwargs key when your command is called.

- *favorite_number* would be *None* if we removed *default=42* from the `arg_botcmd()` call.

Commands using regular expressions

In addition to the fixed commands created with the `botcmd()` decorator, Errbot supports an alternative type of bot function which can be triggered based on a regular expression. These are created using the `re_botcmd()` decorator. There are two forms these can be used, with and without the usual bot prefix.

In both cases, your method will receive the message object same as with a regular `botcmd()`, but instead of an *args* parameter, it takes a *match* parameter which will receive an `re.MatchObject`.

Note: By default, only the first occurrence of a match is returned, even if it can match multiple parts of the message. If you specify *matchall=True*, you will instead get a list of `re.MatchObject` items, containing all the non-overlapping matches that were found in the message.

With a bot prefix

You can define commands that trigger based on a regular expression, but still require a bot prefix at the beginning of the line, in order to create more flexible bot commands. Here's an example of a bot command that lets people ask for cookies:

```python
from errbot import BotPlugin, re_botcmd

class CookieBot(BotPlugin):
    """A cookiemonster bot"""

    @re_botcmd(pattern=r"^(([Cc]an|[Mm]ay) I have a )?cookie please\?$")
    def hand_out_cookies(self, msg, match):
        """
        Gives cookies to people who ask me nicely.

        This command works especially nice if you have the following in
        your `config.py`:

        BOT_ALT_PREFIXES = ('Err',)
        BOT_ALT_PREFIX_SEPARATORS = (':', ',', ';')

        People are then able to say one of the following:

        Err, can I have a cookie please?
        Err: May I have a cookie please?
        Err; cookie please?
        """
        yield "Here's a cookie for you, {}".format(msg.frm)
        yield "/me hands out a cookie."
```

Without a bot prefix

It's also possible to trigger commands even when no bot prefix is specified, by passing *prefixed=False* to the `re_botcmd()` decorator. This is especially useful if you want to trigger on specific keywords that could show up anywhere in a conversation:

```python
import re
from errbot import BotPlugin, re_botcmd

class CookieBot(BotPlugin):
    """A cookiemonster bot"""

    @re_botcmd(pattern=r"(^| )cookies?( |$)", prefixed=False, flags=re.IGNORECASE)
    def listen_for_talk_of_cookies(self, msg, match):
        """Talk of cookies gives Errbot a craving..."""
        return "Somebody mentioned cookies? Om nom nom!"
```

6.3.5 Messaging

Returning multiple responses

Often, with commands that take a long time to run, you may want to be able to send some feedback to the user that the command is progressing. Instead of using a single *return* statement you can use *yield* statements for every line of

output you wish to send to the user.

In the following example, the output will be "Going to sleep", followed by a 10 second wait period and "Waking up" in the end.

```python
from errbot import BotPlugin, botcmd
from time import sleep

class PluginExample(BotPlugin):
    @botcmd
    def longcompute(self, mess, args):
        yield "Going to sleep"
        sleep(10)
        yield "Waking up"
```

Sending a message to a specific user or room

Sometimes, you may wish to send a message to a specific user or a groupchat, for example from pollers or on webhook events. You can do this with *send()*:

```python
self.send(
    self.build_identifier("user@host.tld/resource"),
    "Boo! Bet you weren't expecting me, were you?",
)
```

send() requires a valid *Identifier* instance to send to. *build_identifier()* can be used to build such an identifier. The format(s) supported by *build_identifier* will differ depending on which backend you are using. For example, on Slack it may support *#channel* and *@user*, for XMPP it includes *user@host.tld/resource*, etc.

Templating

It's possible to send Markdown responses using Jinja2 templates.

To do this, first create a directory called *templates* in the directory that also holds your plugin's *.plug* file.

Inside this directory, you can place Markdown templates (with a *.md* extension) in place of the content you wish to show. For example this *hello.md*:

```
Hello, {{name}}!
```

Note: See the Jinja2 Template Designer Documentation for more information on the available template syntax.

Next, tell Errbot which template to use by specifying the *template* parameter to botcmd() (leaving off the *.md* suffix).

Finally, instead of returning a string, return a dictionary where the keys refer to the variables you're substituting inside the template (*{{name}}* in the above template example):

```python
from errbot import BotPlugin, botcmd

class Hello(BotPlugin):
    @botcmd(template="hello")
    def hello(self, msg, args):
        """Say hello to someone"""
        return {'name': args}
```

It's also possible to use templates when using *self.send()*, but in this case you will have to do the template rendering step yourself, like so:

```python
from errbot import BotPlugin, botcmd
from errbot.templating import tenv


class Hello(BotPlugin):
    @botcmd(template="hello")
    def hello(self, msg, args):
        """Say hello to someone"""
        response = tenv().get_template('hello.md').render(name=args)
        self.send(msg.frm, response)
```

Cards

Errbot cards are a canned format for notifications. It is possible to use this format to map to some native format in backends like Slack (Attachment) or Hipchat (Cards).

Similar to a *self.send()* you can use *send_card()* to send a card.

The following code demonstrate the various available fields.

```python
from errbot import BotPlugin, botcmd


class Travel(BotPlugin):
    @botcmd
    def hello_card(self, msg, args):
        """Say a card in the chatroom."""
        self.send_card(title='Title + Body',
                       body='text body to put in the card',
                       thumbnail='https://raw.githubusercontent.com/errbotio/errbot/
→master/docs/_static/errbot.png',
                       image='https://www.google.com/images/branding/googlelogo/2x/
→googlelogo_color_272x92dp.png',
                       link='http://www.google.com',
                       fields=(('First Key','Value1'), ('Second Key','Value2')),
                       color='red',
                       in_reply_to=msg)
```

Trigger a callback with every message received

It's possible to add a callback that will be called on every message sent either directly to the bot, or to a chatroom that the bot is in:

```python
from errbot import BotPlugin


class PluginExample(BotPlugin):
    def callback_message(self, mess):
        if mess.body.find('cookie') != -1:
            self.send(
                mess.frm,
                "What what somebody said cookie!?",
            )
```

6.3.6 Presence

Presence describes the concept of a person's availability state, such as *online* or *away*, possibly with an optional message.

Callbacks for presence changes

Plugins may override *callback_presence()* in order to receive notifications of presence changes. You will receive a *Presence* object for every presence change received by Errbot.

Here's an example which simply logs each presence change to the log when it includes a status message:

```python
from errbot import BotPlugin


class PluginExample(BotPlugin):
    def callback_presence(self, presence):
        if presence.get_message() is not None:
            self.log.info(presence)
```

Change the presence or status of the bot

You can also, depending on the backend you use, change the current status of the bot. This allows you to make a moody bot that leaves the room when it is in a bad mood ;)

```python
from errbot import BotPlugin, botcmd, ONLINE, AWAY


class PluginExample(BotPlugin):
    @botcmd
    def grumpy(self, mess, args):
        self.change_presence(AWAY, 'I am tired of you all!')

    @botcmd
    def happy(self, mess, args):
        self.change_presence(ONLINE, 'I am back and so happy to see you!')
```

6.3.7 Mentions

Depending on the backend used, users can mention and notify other users by using a special syntax like *@gbin*. With this feature, a plugin can listen to the mentioned users in the chat.

How to use it

Here is an example to listen to every mention and report them back on the chat.

```python
from errbot import BotPlugin


class PluginExample(BotPlugin):

    def callback_mention(self, message, mentioned_people):
        for identifier in mentioned_people:
            self.send(message.frm, 'User %s has been mentioned' % identifier)
```

Identifying if the bot itself has been mentioned

Simply test the presence of the bot identifier within the *mentioned_people*:

```python
from errbot import BotPlugin


class PluginExample(BotPlugin):

    def callback_mention(self, message, mentioned_people):
        if self.bot_identifier in mentioned_people:
            self.send(message.frm, 'Errbot has been mentioned !')
```

6.3.8 Persistence

Persistence describes the ability for the plugins to persist data even if Errbot is restarted.

How to use it

Your plugin *is* the store, simply use self as a dictionary.

Here is a simple example for storing and retreiving a value from the store.

```python
from errbot import BotPlugin, botcmd


class PluginExample(BotPlugin):
    @botcmd
    def remember(self, msg, args):
        self['TODO'] = args

    @botcmd
    def recall(self, msg, args):
        return self['TODO']
```

Caveats

The storing occurs when you *assign the key*:

```python
# THIS WON'T WORK
d = {}
self['FOO'] = d
d['subkey'] = 'NONONONONONO'
```

What you need to do instead: (manual method)

```python
# THIS WORKS
d = {}
self['FOO'] = d

# later ...
d['subkey'] = 'NONONONONONO'
self['FOO'] = d  # restore the full key if something changed in memory.
```

Or use the mutable contex manager:

```
# THIS WORKS AND IS CLEANER
d = {}
self['FOO'] = d

# later ...

with self.mutable('FOO') as d:
    d['subkey'] = 'NONONONONONO'
# it will save automatically the key
```

6.3.9 Configuration

Plugin configuration through the built-in *!config* command

Errbot can keep a simple python object for the configuration of your plugin. This avoids the need for admins to configure settings in some kind of configuration file, instead allowing configuration to happen directly through chat commands.

In order to enable this feature, you need to provide a configuration template (ie. a config example) by overriding the *get_configuration_template()* method. For example, a plugin might request a dictionary with 2 entries:

```
from errbot import BotPlugin

class PluginExample(BotPlugin):
    def get_configuration_template(self):
        return {'ID_TOKEN': '00112233445566778899aabbccddeeff',
                'USERNAME':'changeme'}
```

With this in place, an admin will be able to request the default configuration template with *!plugin config PluginExample*. He or she could then give the command *!plugin config PluginExample {'ID_TOKEN' : '00112233445566778899aabbccddeeff', 'USERNAME':'changeme'}* to enable that configuration.

It will also be possible to recall the configuration template, as well as the config that is actually set, by issuing *!plugin config PluginExample* again.

Within your code, the config that is set will be in *self.config*:

```
@botcmd
def mycommand(self, mess, args):
    # oh I need my TOKEN !
    token = self.config['ID_TOKEN']
```

> **Warning:** If there is no configuration set yet, *self.config* will be *None*.

Supplying partial configuration

Sometimes you want to allow users to only supply a part of the configuration they wish to override from the template instead of having to copy-paste and modify the complete entry.

This can be achieved by overriding *configure()*:

```python
from itertools import chain

CONFIG_TEMPLATE = {'ID_TOKEN': '00112233445566778899aabbccddeeff',
                   'USERNAME':'changeme'}


def configure(self, configuration):
    if configuration is not None and configuration != {}:
        config = dict(chain(CONFIG_TEMPLATE.items(),
                            configuration.items()))
    else:
        config = CONFIG_TEMPLATE
    super(PluginExample, self).configure(config)
```

What this achieves is that it creates a Python dictionary object which contains all the values from our *CON-FIG_TEMPLATE* and then updates that dictionary with the configuration received when calling the *!config* command. *!config* must be passed a dictionary for this to work.

If you wish to reset the configuration to its defaults all you need to do is pass an empty dictionary to *!config*.

You'll now also need to override *get_configuration_template()* and return the *CONFIG_TEMPLATE* in that function:

```python
def get_configuration_template(self):
    return CONFIG_TEMPLATE
```

Using custom configuration checks

By default, Errbot will check the supplied configuration against the configuration template, and raise an error if the structure of the two doesn't match.

You need to override the *check_configuration()* method if you wish do some other form of configuration validation. This method will be called automatically when an admin configures your plugin with the *!config* command.

> **Warning:** If there is no configuration set yet, it will pass *None* as parameter. Be mindful of this situation.

Using the partial configuration trick as shown above requires you to override *check_configuration()*, so at a minimum you'll need this:

```python
def check_configuration(self, configuration):
    pass
```

We suggest that you at least do some validation instead of nothing but that is up to you.

6.3.10 Streams

Streams are file transfers. It can be used to store documents, index them, send generated content on the fly etc.

Waiting for incoming file transfers

The bot can be sent files from the users. You only have to implement the *callback_stream()* method on your plugin to be notified for new incoming file transfer requests.

Note: not all backends supports this, check if it has been correctly implemented from the backend itself.

For example, getting the initiator of the transfer and the content, see *Stream* for more info about the various fields.

```python
from errbot import BotPlugin, botcmd


class PluginExample(BotPlugin):

    def callback_stream(self, stream):
        self.send(stream.identifier, "File request from :" + str(stream.identifier))
        stream.accept()
        self.send(stream.identifier, "Content:" + str(stream.fsource.read()))
```

Sending a file to a user or a room

You can use *send_stream_request()* to initiate a transfer:

```python
stream = self.send_stream_request(msg.frm, open('/tmp/myfile.zip', 'r'), name='bills.
↪zip', stream_type='application/zip')
```

The returned stream object can be used to monitor the progress of the transfer with *stream.status*, *stream.transfered* etc... See *Stream* for more details.

6.3.11 Plugin Dependencies

Sometimes you need to be able to share a plugin feature with another. For example imagine you have a series of plugin configured the same way, you might want to make them depend on a central plugin taking care of the configuration that would share it with all the others.

Declaring dependencies

If you want to be able to use a plugin from another, the later needs to be activated before the former. You can ask Errbot to do so by adding a comma separated name list of the plugins your plugin is depending on in the **Core** section of your plug file like this:

```ini
[Core]
Name = MyPlugin
Module = myplugin
DependsOn = OtherPlugin1, OtherPlugin2
```

Using dependencies

Once a dependent plugin has been declared, you can use it as soon as your plugin is activated.

```python
from errbot import BotPlugin, botcmd


class OtherPlugin1(BotPlugin):

    def activate(self):
        self.my_variable = 'hello'
        super().activate()
```

If you want to use it from MyPlugin:

```python
from errbot import BotPlugin, botcmd


class MyPlugin(BotPlugin):

    @botcmd
    def hello(self, msg, args):
        return self.get_plugin('OtherPlugin1').my_variable
```

Important to note: if you want to use a dependent plugin from within activate, you need to be in activated state, for example:

```python
from errbot import BotPlugin, botcmd


class MyPlugin(BotPlugin):

    def activate(self):
        super().activate()  # <-- needs to be *before* get_plugin
        self.other = self.get_plugin('OtherPlugin1')

    @botcmd
    def hello(self, msg, args):
        return self.other.my_variable
```

6.3.12 Dynamic plugins (advanced)

Sometimes the list of commands the bot wants to expose is not known at plugin development time.

For example, you have a remote service with commands that can be set externally.

This feature allows you to define and update on the fly plugins and their available commands.

Defining new commands

You can create a commands from scratch with *Command*. By default it will be a *botcmd()*.

```python
# from a lambda
my_command1 = Command(lambda plugin, msg, args: 'received %s' % msg, name='my_command
→', doc='documentation of my_command')

# or from a function
def my_command(plugin, msg, args):
    """
    documentation of my_command.
    """
    return 'received %s' % msg

my_command2 = Command(my_command)
```

Note: the function will by annotated by a border effect, be sure to use a local function if you want to derive commands for the same underlying function.

Registering the new plugin

Once you have your series of Commands defined, you can package them in a plugin and expose them on errbot with
`create_dynamic_plugin()`.

```
# from activate, another bot command, poll etc.
self.create_dynamic_plugin('my_plugin', (my_command1, my_command2))
```

Refreshing a plugin

You need to detroy and recreate the plugin to refresh its commands.

```
self.destroy_dynamic_plugin('my_plugin')
self.create_dynamic_plugin('my_plugin', (my_command1, my_command2, my_command3))
```

Customizing the type of commands and parameters

You can use other type of commands by specifying cmd_type and pass them parameters with cmd_args and
cmd_kwargs.

```
# for example a botmatch
re1 = Command(lambda plugin, msg, match: 'fffound',
              name='ffound',
              cmd_type=botmatch,
              cmd_args=(r'^.*cheese.*$',))

# or a split_args_with
saw = Command(lambda plugin, msg, args: '+'.join(args),
              name='splitme',
              cmd_kwargs={'split_args_with': ','})
```

6.3.13 Scheduling

Calling a function at a regular interval

It's possible to automatically call functions at regular intervals, using the *start_poller()* and *stop_poller()*
methods.

For example, you could schedule a callback to be executed once every minute when your plugin gets activated:

```
from errbot import BotPlugin

class PluginExample(BotPlugin):
    def my_callback(self):
        self.log.debug('I am called every minute')

    def activate(self):
        super().activate()
        self.start_poller(60, self.my_callback)
```

It is also possible to specify the *times* parameter, which denotes how many times the function should be called, for
instance:

```python
from errbot import BotPlugin

class PluginExample(BotPlugin):
    def my_callback(self):
        self.log.debug('I got called after a minute (and just once)')

    def activate(self):
        super().activate()
        self.start_poller(60, self.my_callback, times=1)
```

6.3.14 Webhooks

Errbot has a small integrated webserver that is capable of hooking up endpoints to methods inside your plugins.

You must configure the *Webserver* plugin before this functionality can be used. You can get the configuration template using *!plugin config Webserver*, from where it's just a simple matter of plugging in the desired settings.

Note: There is a *!generate certificate* command to generate a self-signed certificate in case you want to enable SSL connections and do not have a certificate.

Warning: It is not recommended to expose Errbot's webserver directly to the network. Instead, we recommend placing it behind a webserver such as nginx or Apache.

Simple webhooks

All you need to do for a plugin of yours to listen to a specific URI is to apply the `webhook()` decorator to your method. Whatever it returns will be returned in response to the request:

```python
from errbot import BotPlugin, webhook

class PluginExample(BotPlugin):
    @webhook
    def test(self, request):
        self.log.debug(repr(request))
        return "OK"
```

This will listen for POST requests on http://yourserver.tld:yourport/test/, and return *"OK"* as the response body.

Note: If you return *None*, an empty 200 response will be sent.

You can also set a custom URI pattern by providing the *uri_rule* parameter:

```python
from errbot import BotPlugin, webhook

class PluginExample(BotPlugin):
    @webhook('/example/<name>/<action>/')
    def test(self, request, name, action):
        return "User %s is performing %s" % (name, action)
```

Refer to the documentation on Flask's route for details on the supported syntax (Errbot uses Flask internally).

Handling JSON request

If an incoming request has the MIME media type set to *application/json* the request will automatically be decoded as JSON. You will receive the result of calling *json.loads()* on *request* automatically so that you won't have to do this yourself.

Handling form-encoded requests

Form-encoded requests (those with an *application/x-www-form-urlencoded* mimetype) are very simple to handle as well, you just need to specify the *form_param* parameter.

A good example for this is the GitHub format which posts a form with a *payload* parameter:

```python
from errbot import BotPlugin, webhook


class Github(BotPlugin):
    @webhook('/github/', form_param = 'payload')
    def notification(self, payload):
        for room in self.bot_config.CHATROOM_PRESENCE:
            self.send(
                self.build_identifier(room),
                'Commit on %s!' % payload['repository']['name'],
            )
```

The raw request

The above webhooks are convenient for simple tasks, but sometimes you might wish to have more power and have access to the actual request itself. By setting the *raw* parameter of the webhook() decorator to *True*, you will be able to get the bottle.BaseRequest which contains all the details about the actual request:

```python
from errbot import BotPlugin, webhook


class PluginExample(BotPlugin):
    @webhook(raw=True)
    def test(self, request):
        user_agent = request.get_header("user-agent", "Unknown")
        return "Your user-agent is {}".format(user_agent)
```

Returning custom headers and status codes

Adjusting the response headers, setting cookies or returning a different status code can all be done by manipulating the flask response object. The Flask docs on the response object explain this in more detail. Here's an example of setting a custom header:

```python
from errbot import BotPlugin, webhook
from flask import after_this_request


class PluginExample(BotPlugin):
    @webhook
    def example(self, incoming_request):
        @after_this_request
        def add_header(response):
            response.headers['X-Powered-By'] = 'Errbot'
        return "OK"
```

Flask also has various helpers such as the *abort()* method. Using this method we could, for example, return a 403 forbidden response like so:

```python
from errbot import BotPlugin, webhook
from flask import abort


class PluginExample(BotPlugin):
    @webhook
    def example(self, incoming_request):
        abort(403, "Forbidden")
```

Testing a webhook through chat

You can use the *!webhook* command to test webhooks without making an actual HTTP request, using the following format: *!webhook test [endpoint] [post_content]*

For example:

```
!webhook test github payload=%7B%22pusher%22%3A%7B%22name%22%3A%22gbin%22%2C%22email
→%22%3A%22gbin%40gootz.net%22%7D%2C%22repository%22%3A%7B%22name%22%3A%22test%22%2C
→%22created_at%22%3A%222012-08-12T16%3A09%3A43-07%3A00%22%2C%22has_wiki%22%3Atrue%2C
→%22size%22%3A128%2C%22private%22%3Afalse%2C%22watchers%22%3A0%2C%22url%22%3A%22https
→%3A%2F%2Fgithub.com%2Fgbin%2Ftest%22%2C%22fork%22%3Afalse%2C%22pushed_at%22%3A
→%222012-08-12T16%3A26%3A35-07%3A00%22%2C%22has_downloads%22%3Atrue%2C%22open_issues
→%22%3A0%2C%22has_issues%22%3Atrue%2C%22stargazers%22%3A0%2C%22forks%22%3A0%2C
→%22description%22%3A%22ignore%20this%2C%20this%20is%20for%20testing%20the%20new
→%20err%20github%20integration%22%2C%22owner%22%3A%7B%22name%22%3A%22gbin%22%2C
→%22email%22%3A%22gbin%40gootz.net%22%7D%7D%2C%22forced%22%3Afalse%2C%22after%22%3A
→%22b3cd9e66e52e4783c1a0b98fbaaad6258669275f%22%2C%22head_commit%22%3A%7B%22added%22
→%3A%5B%5D%2C%22modified%22%3A%5B%22README.md%22%5D%2C%22timestamp%22%3A%222012-08-
→12T16%3A24%3A25-07%3A00%22%2C%22removed%22%3A%5B%5D%2C%22author%22%3A%7B%22name%22
→%3A%22Guillaume%20BINET%22%2C%22username%22%3A%22gbin%22%2C%22email%22%3A%22gbin
→%40gootz.net%22%7D%2C%22url%22%3A%22https%3A%2F%2Fgithub.com%2Fgbin%2Ftest%2Fcommit
→%2Fb3cd9e66e52e4783c1a0b98fbaaad6258669275f%22%2C%22id%22%3A
→%22b3cd9e66e52e4783c1a0b98fbaaad6258669275f%22%2C%22distinct%22%3Atrue%2C%22message
→%22%3A%22voila%22%2C%22committer%22%3A%7B%22name%22%3A%22Guillaume%20BINET%22%2C
→%22username%22%3A%22gbin%22%2C%22email%22%3A%22gbin%40gootz.net%22%7D%7D%2C
→%22deleted%22%3Afalse%2C%22commits%22%3A%5B%7B%22added%22%3A%5B%5D%2C%22modified%22
→%3A%5B%22README.md%22%5D%2C%22timestamp%22%3A%222012-08-12T16%3A24%3A25-07%3A00%22
→%2C%22removed%22%3A%5B%5D%2C%22author%22%3A%7B%22name%22%3A%22Guillaume%20BINET%22
→%2C%22username%22%3A%22gbin%22%2C%22email%22%3A%22gbin%40gootz.net%22%7D%2C%22url%22
→%3A%22https%3A%2F%2Fgithub.com%2Fgbin%2Ftest%2Fcommit
→%2Fb3cd9e66e52e4783c1a0b98fbaaad6258669275f%22%2C%22id%22%3A
→%22b3cd9e66e52e4783c1a0b98fbaaad6258669275f%22%2C%22distinct%22%3Atrue%2C%22message
→%22%3A%22voila%22%2C%22committer%22%3A%7B%22name%22%3A%22Guillaume%20BINET%22%2C
→%22username%22%3A%22gbin%22%2C%22email%22%3A%22gbin%40gootz.net%22%7D%7D%5D%2C%22ref
→%22%3A%22refs%2Fheads%2Fmaster%22%2C%22before%22%3A
→%2229b1f5e59b7799073b6d792ce76076c200987265%22%2C%22compare%22%3A%22https%3A%2F
→%2Fgithub.com%2Fgbin%2Ftest%2Fcompare%2F29b1f5e59b77...b3cd9e66e52e%22%2C%22created
→%22%3Afalse%7D
```

Note: You can get a list of all the endpoints with the *!webstatus* command.

6.3.15 Testing your plugins

Just as Errbot has tests that validates that it behaves correctly so should your plugin. Errbot is tested using Python's py.test module and because we already provide some utilities for that we highly advise you to use *py.test* too.

We're going to write a simple plugin named *myplugin.py* with a *MyPlugin* class. It's tests will be stored in *test_myplugin.py* in the same directory.

Interacting with the bot

Lets go for an example, *myplugin.py*:

```python
from errbot import BotPlugin, botcmd

class MyPlugin(BotPlugin):
    @botcmd
    def mycommand(self, message, args):
        return "This is my awesome command"
```

And *myplugin.plug*:

```ini
[Core]
Name = MyPlugin
Module = myplugin

[Documentation]
Description = my plugin
```

This does absolutely nothing shocking, but how do you test it? We need to interact with the bot somehow, send it *!mycommand* and validate the reply. Fortunatly Errbot provides some help.

Our test, *test_myplugin.py*:

```python
pytest_plugins = ["errbot.backends.test"]

extra_plugin_dir = '.'

def test_command(testbot):
    testbot.push_message('!mycommand')
    assert 'This is my awesome command' in testbot.pop_message()
```

Lets walk through this line for line. First of all, we specify our pytest fixture location `test` in the backends tests, to allow us to spin up a bot for testing purposes and interact with the message queue. To avoid specifying the module in every test module, you can simply place this line in your conftest.py.

Then we set *extra_plugin_dir* to ., the current directory so that the test bot will pick up on your plugin.

After that we define our first *test_* method which simply sends a command to the bot using `push_message()` and then asserts that the response we expect, *"This is my awesome command"* is in the message we receive from the bot which we get by calling `pop_message()`.

You can assert the response of a command using the method assertCommand of the testbot. *testbot.assertCommand('!mycommand', 'This is my awesome command')* to achieve the equivalent of pushing message and asserting the response in the popped message.'

Helper methods

Often enough you'll have methods in your plugins that do things for you that are not decorated with *@botcmd* since the user never calls out to these methods directly.

Such helper methods can be either instance methods, methods that take *self* as the first argument because they need access to data stored on the bot or class or static methods, decorated with either *@classmethod* or *@staticmethod*:

```python
class MyPlugin(BotPlugin):
    @botcmd
    def mycommand(self, message, args):
        return self.mycommand_helper()

    @staticmethod
    def mycommand_helper():
        return "This is my awesome command"
```

The *mycommand_helper* method does not need any information stored on the bot whatsoever or any other bot state. It can function standalone but it makes sense organisation-wise to have it be a member of the *MyPlugin* class.

Such methods can be tested very easily, without needing a bot:

```python
import myplugin

def test_mycommand_helper():
    expected = "This is my awesome command"
    result = myplugin.MyPlugin.mycommand_helper()
    assert result == expected
```

Here we simply import *myplugin* and since it's a *@staticmethod* we can directly access it through *myplugin.MyPlugin.method()*.

Sometimes however a helper method needs information stored on the bot or manipulate some of that so you declare an instance method instead:

```python
class MyPlugin(BotPlugin):
    @botcmd
    def mycommand(self, message, args):
        return self.mycommand_helper()

    def mycommand_helper(self):
        return "This is my awesome command"
```

Now what? We can't access the method directly anymore because we need an instance of the bot and the plugin and we can't just send *!mycommand_helper* to the bot, it's not a bot command (and if it were it would be *!mycommand helper* anyway).

What we need now is get access to the instance of our plugin itself. Fortunately for us, there's a method that can help us do just that:

```python
extra_plugin_dir = '.'

def test_mycommand_helper(testbot):
    plugin = testbot._bot.plugin_manager.get_plugin_obj_by_name('MyPlugin')
    expected = "This is my awesome command"
    result = plugin.mycommand_helper()
    assert result == expected
```

There we go, we first grab our plugin using a helper method on *plugin_manager* and then simply execute the method and compare the result with the expected result. You can also access *@classmethod* or *@staticmethod* methods this way, but you don't have to.

Sometimes a helper method will be making HTTP or API requests which might not be possible to test directly. In that case, we need to mock that particular method and make it return the expected value without actually making the request.

```python
URL = 'http://errbot.io'


class MyPlugin(BotPlugin):
    @botcmd
    def mycommand(self, message, args):
        return self.mycommand_helper()


    def mycommand_helper(self):
        return (requests.get(URL).status_code)
```

What we need now is to somehow replace the method making the request with our mock object and *inject_mocks* method comes in handy.

Refer unittest.mock for more information about mock.

```python
from unittest.mock import MagicMock

extra_plugin_dir = '.'


def test_mycommand_helper(testbot):
    helper_mock = MagicMock(return_value='200')
    mock_dict = {'mycommand_helper': helper_mock}
    testbot.inject_mocks('MyPlugin', mock_dict)
    testbot.push_message('!mycommand')
    expected = '200'
    result = testbot.pop_message()
    assert result == expected
```

Pattern

It's a good idea to split up your plugin in two types of methods, those that directly interact with the user and those that do extra stuff you need.

If you do this the *@botcmd* methods should only concern themselves with giving output back to the user and calling different other functions it needs in order to fulfill the user's request.

Try to keep as many helper methods simple, there's nothing wrong with having an extra helper or two to avoid having to nest fifteen if-statements. It becomes more legible, easier to maintain and easier to test.

If you can, try to make your helper methods *@staticmethod* decorated functions, it's easier to test and you don't need a full running bot for those tests.

All together now

myplugin.py:

```python
from errbot import BotPlugin, botcmd
```

```python
class MyPlugin(BotPlugin):
    @botcmd
    def mycommand(self, message, args):
        return self.mycommand_helper()

    @botcmd
    def mycommand_another(self, message, args):
        return self.mycommand_another_helper()

    @staticmethod
    def mycommand_helper():
        return "This is my awesome command"

    def mycommand_another_helper(self):
        return "This is another awesome command"
```

myplugin.plug:

```ini
[Core]
Name = MyPlugin
Module = myplugin

[Documentation]
Description = my plugin
```

test_myplugin.py:

```python
import myplugin

extra_plugin_dir = '.'

def test_mycommand(testbot):
    testbot.push_message('!mycommand')
    assert 'This is my awesome command' in testbot.pop_message()

def test_mycommand_another(testbot):
    testbot.push_message('!mycommand another')
    assert 'This is another awesome command' in testbot.pop_message()

def test_mycommand_helper():
    expected = "This is my awesome command"
    result = myplugin.MyPlugin.mycommand_helper()
    assert result == expected

def test_mycommand_another_helper():
    plugin = testbot._bot.plugin_manager.get_plugin_obj_by_name('MyPlugin')
    expected = "This is another awesome command"
    result = plugin.mycommand_another_helper()
    assert result == expected
```

You can now simply run `py.test` to execute the tests.

PEP-8 and code coverage

If you feel like it you can also add syntax checkers like *pep8* into the mix to validate your code behaves to certain stylistic best practices set out in PEP-8.

First, install the pep8 for py.test: `pip install pytest-pep8`.

Then, simply add *–pep8* to the test invocation command: *py.test –pep8*.

You also want to know how well your tests cover you code.

To that end, install coverage: `pip install coverage` and then run your tests like this: `coverage run --source myplugin -m py.test --pep8`.

You can now have a look at coverage statistics through `coverage report`:

```
Name          Stmts   Miss  Cover
----------------------------------
myplugin         49      0   100%
```

It's also possible to generate an HTML report with `coverage html` and opening the resulting *htmlcov/index.html*.

Travis and Coveralls

Last but not least, you can run your tests on Travis-CI so when you update code or others submit pull requests the tests will automatically run confirming everything still works.

In order to do that you'll need a *.travis.yml* similar to this:

```yaml
language: python
python:
  - 2.7
  - 3.3
  - 3.4
install:
  - pip install -q errbot pytest pytest-pep8 --use-wheel
  - pip install -q coverage coveralls --use-wheel
script:
  - coverage run --source myplugin -m py.test --pep8
after_success:
  - coveralls
notifications:
  email: false
```

Most of it is self-explanatory, except for perhaps the *after_success*. The author of this plugin uses Coveralls.io to keep track of code coverage so after a successful build we call out to coveralls and upload the statistics. It's for this reason that we *pip install [..] coveralls [..]* in the *.travis.yml*.

The *-q* flag causes pip to be a lot more quiet and *–use-wheel* will cause pip to use wheels if available, speeding up your builds if you happen to depend on something that builds a C-extension.

Both Travis-CI and Coveralls easily integrate with Github hosted code.

6.3.16 Logging

Logging information on what your plugin is doing can be a tremendous asset when managing your bot in production, especially when something is going wrong.

Errbot uses the standard Python logging library to log messages internally and provides a logger for your own plugins to use as well as *self.log*. You can use this logger to log status messages to the log like this:

```python
from errbot import BotPlugin


class PluginExample(BotPlugin):
    def callback_message(self, message):
        self.log.info("I just received a message!")
```

6.3.17 Exception Handling

Properly handling exceptions helps you build plugins that don't crash or produce unintended side-effects when the user or your code does something you did not expect. Combined with logging, exceptions also allow you to get visibility of areas in which your bot is failing and ultimately address problems to improve user experience.

Exceptions in Errbot plugins should be handled slightly differently from how exceptions are normally used in Python. When an unhandled exception is raised during the execution of a command, Errbot sends a message like this:

```
Computer says nooo. See logs for details:
<exception message here>
```

The above is neither helpful nor user-friendly, as the exception message may be too technical or brief (notice there is no traceback) for the user to understand. Even if you were to provide your own exception message, the "Computer says nooo . . . " part is neither particularly attractive or informative.

When handling exceptions, follow these steps:

- trap the exception as you usually would

- log the exception inside of the `except` block

    - `self.log.exception('Descriptive message here')`

    - import and use the logging module directly if you don't have access to `self`

    - `self.log` is just a convenience wrapper for the standard Python `logging` module

- send a message describing what the user did wrong and recommend a solution for them to try their command again

- do not re-raise your exception in the `except` block as you normally would. This is usually done in order to produce an entry in the error logs, but we've already logged the exception, and by not re-raising it, we prevent that automatic "Computer says nooo. . . . " message from being sent

Also, note that there is a `errbot.ValidationException` class which you can use inside your helper methods to raise meaningful errors and handle them accordingly.

Here's an example:

```python
from errbot import BotPlugin, arg_botcmd, ValidationException


class FooBot(BotPlugin):
    """An example bot"""

    @arg_botcmd('first_name', type=str)
    def add_first_name(self, message, first_name):
        """Add your first name if it doesn't contain any digits"""
        try:
            FooBot.validate_first_name(first_name)
        except ValidationException as exc:
            self.log.exception(
                'first_name=%s contained a digit' % first_name
```

(continues on next page)

```
        )
        return 'Your first name cannot contain a digit.'

    # Add some code here to add the given name to your database

    return "Your name has been added."

@staticmethod
def validate_first_name(first_name):
    if any(char.isdigit() for char in first_name):
        raise ValidationException(
            "first_name=%s contained a digit" % first_name
        )
```

6.3.18 Plugin compatibility settings

Errbot compatibility

Sometimes when your plugin breaks under a specific version of Errbot, you might want to warn the user of your plugin and not load it.

You can do it by adding an **Errbot** section to your plug file like this:

```
[Core]
Name = MyPlugin
Module = myplugin

[Documentation]
Description = my plugin

[Errbot]
Min=2.4.0
Max=2.6.0
```

If the **Errbot** section is omitted, it defaults to "compatible with any version".

If the **Min** option is omitted, there is no minimum version enforced.

If the **Max** option is omitted, there is no maximum version enforced.

Versions need to be a 3 dotted one (ie 2.4 is not allowed but 2.4.0 is). And it understands those suffixes:

- "-beta"
- "-rc1"
- "-rc2"
- etc.

For example: 2.4.0-rc1

note: -beta1 or -rc are illegal. Only rc can get a numerical suffix.

6.3.19 Backend-specifics

Errbot uses external libraries for most backends, which may offer additional functionality not exposed by Errbot in a generic, backend-agnostic fashion.

It is possible to access the underlying client used by the backend you are using in order to provide functionality that isn't otherwise available. Additionally, interacting directly with the bot internals gives you the freedom to control Errbot in highly specific ways that may not be officially supported.

> **Warning:** The following instructions describe how to interface directly with the underlying bot object and clients of backends. We offer no guarantees that these internal APIs are stable or that a given backend will continue to use a given client in the future. The following information is provided **as-is** without any official support. We can give **no** guarantees about API stability on the topics described below.

Getting to the bot object

From within a plugin, you may access *self._bot* in order to get to the instance of the currently running bot class. For example, with the Telegram backend this would be an instance of `TelegramBackend`:

```
>>> type(self._bot)
<class 'errbot.backends.TelegramBackend'>
```

To find out what methods each bot backend has, you can take a look at the documentation of the various backends in the `errbot.backends` package.

Plugins may use the *self._bot* object to offer tailored, backend-specific functionality on specific backends. To determine which backend is being used, a plugin can inspect the *self._bot.mode* property. The following table lists all the values for *mode* for the official backends:

Backend	Mode value
hipchat	hipchat
irc	irc
slack	slack
telegram_messenger	telegram
test	test
text	text
xmpp	xmpp

Here's an example of using a backend-specific feature. In Slack, emoji reactions can be added to messages the bot receives using the *add_reaction* and *remove_reaction* methods. For example, you could add an hourglass to messages that will take a long time to reply fully to.

```python
from errbot import BotPlugin, botcmd

class PluginExample(BotPlugin):
    @botcmd
    def longcompute(self, mess, args):
        if self._bot.mode == "slack":
            self._bot.add_reaction(mess, "hourglass")
        else:
            yield "Finding the answer..."

        time.sleep(10)

        yield "The answer is: 42"
        if self._bot.mode == "slack":
            self._bot.remove_reaction(mess, "hourglass")
```

Getting to the underlying client library

Most of the backends use a third-party library in order to connect to their respective network. These libraries often support additional features which Errbot doesn't expose in a generic way so you may wish to make use of these in order to access advanced functionality.

Backends set their own attribute(s) to point to the underlying libraries' client instance(s). The following table lists these attributes for the official backends, along with the library used by the backend:

Backend	Library	Attribute(s)
hipchat	sleekxmpp + hypchat	`self._bot.conn self._bot.conn.hypchat`
irc	irc	`self._bot.conn self._bot.conn.connection`
slack	slackclient	`self._bot.sc`
telegram_messenger	telegram-python-bot	`self._bot.telegram`
xmpp	sleekxmpp	`self._bot.conn`

6.4 Flow development

Flows are a feature in Errbot to enable plugin designers to chain several plugin commands together into a "conversation".

For example, imagine interacting with a bot that needs more that one command, like setting up a poll in a chatroom:

```
User: !poll new Where do we go for lunch ?

Bot: Flow poll_setup started, you can continue with:
     !poll newoption <your option>

User: !poll newoption Greek

Bot: Option added, current options:
     - Greek

Bot: You can continue with:
     !poll newoption <your option>
     !poll start

User: !poll newoption French

Bot: Option added, current options:
     - Greek
     - French

Bot: You can continue with:
     !poll newoption <your option>
     !poll start

User: !poll start
[...]
```

In this guide we will explain the underlying concepts and basics of writing flows. Prerequesite: you need to be familiar with the normal errbot plugin development.

6.4.1 Flows Concepts

Static structure

Flows are represented as graphs. Those graphs have a root (FlowRoot), which is basically their entry point, and are composed of nodes (FlowNodes). Every node represents an Errbot command.

Fig. 1: Example of a flow construction.

This defines a simple flow where for example this sequence of commands is possible:

```
!command1 !command2 !command3 !command1 !command2 !command3 !last_command
```

On the connections of those nodes (), you can attach **predicates**, predicates are simple conditions to allow the flow to continue without any user intervention.

Execution

At execution time, Errbot will keep track of who started the flow, and at what step (node) it is currently. On top of that, Errbot will initialize a context for the entire conversation. The context is a simple Python dictionary and it is attached to only one conversation. Think of this like the persistence for plugins, but linked to a conversation only.

If you don't specify any predicate when you build your flow, every single step is "manual". It means that Errbot will wait for the user to execute one of the possible commands at every step to advance along the graph.

Predicates can be used to trigger a command automatically. Predicates are simple functions saying to Errbot, "this command has enough in the context to be able to execute without any user intervention". At any time if a predicate is verified after a step is executed, Errbot will proceed and execute the next step.

6.4.2 Basic Flow Definition

Flows are like plugins

They are defined by a `.flow` file, similar to the plugin ones:

```
[Core]
Name = MyFlows.
Module = myflows.py

[Documentation]
Description = my documentation.

[Python]
Version = 2+
```

Now in the `myflows.py` file you will have pretty familiar structure with a `BotFlow` as type and @botflow as flow decorator:

```python
from errbot import botflow, FlowRoot, BotFlow


class MyFlows(BotFlow):
    """ Conversation flows for Errbot"""
```

(continues on next page)

```
    @botflow
    def example(self, flow: FlowRoot):
        """ Docs for the flow example comes here """
        # [...]
```

Errbot will pass the root of the flow as the only parameter to your flow definition so you can build your graph from there.

Making a simple graph

Within your flow, you can connect commands together. For example, to make a simple linear flow between `!first`, `!second` and `!third`:

```
@botflow
def example(self, flow: FlowRoot):
    first_step = flow.connect('first')              # first is a command name from any
→loaded plugin.
    second_step = first_step.connect('second')
    third_step = second_step.connect('third')
```

You can represent this flow like this:

O is the state "not started" for the flow `example`.

You can start this flow manually by doing `!flows start example`.

The bot will tell you that it expects a `!first` command:

Once you have executed `!first`, you will be in that state:

The bot will tell you that it expects `!second`, etc.

Making a flow start automatically

Now, usually flows are linked to a first action your users want to do. For example: `!poll new`, `!vm create`, `!report init` or first commands like that that suggests that you will have a follow-up.

To trigger a flow automatically on those first commands, you can use `auto_trigger`.

```
@botflow
def example(self, flow: FlowRoot):
    first_step = flow.connect('first', auto_trigger=True)
    second_step = first_step.connect('second')
    third_step = second_step.connect('third')
```

You can still represent this flow like this:

BUT, when a user will execute a `!first` command, the bot will instantly instantiate a Flow in this state:

And tell the user that `!second` is the follow-up.

Flow ending

If a node has no more children and a user passed it, it will automatically end the flow.

Sometimes, with loops etc., you might want to explicitly mark an END FlowNode with a predicate. You can do it like this, for example for a guessing game plugin:

In the flow code. . .

```python
from errbot import botflow, FlowRoot, BotFlow, FLOW_END


class GuessFlows(BotFlow):
    """ Conversation flows related to polls"""

    @botflow
    def guess(self, flow: FlowRoot):
        """ This is a flow that can set a guessing game."""
        # setup Flow
        game_created = flow.connect('tryme', auto_trigger=True)
        one_guess = game_created.connect('guessing')
        one_guess.connect(one_guess)  # loop on itself
        one_guess.connect(FLOW_END, predicate=lambda ctx: ctx['ended'])
```

6.4.3 Advanced Flow Definitions

Storing something in the flow context

Flows have a state the plugins can use to store some contextual information. Let's take back out simple linear flow:

```python
@botflow
def example(self, flow: FlowRoot):
    first_step = flow.connect('first')
    second_step = first_step.connect('second')
    third_step = second_step.connect('third')
```

You can represent this flow like this:

You can store something in the context, for example in `!first` and retrieve it in `!second`. Like this:

```python
@botcmd
def first(self, msg, args):
    msg.ctx['mydata'] = 'Hello'
    return 'First done!'

@botcmd
def second(self, msg, args):
    return msg.ctx['mydata'] + ' World!'
```

`msg.ctx` is a dictionary created every time a flow starts.

Making a step execute automatically

In our previous example, if `msg.ctx['mydata']` is populated, we can arguably expect that Errbot should not wait for the user to enter `!second` to execute it and just proceed by itself.

You can do that by defining a **predicate**, which is a simple function that returns `True` if the conditions to proceed to the next step are met. The function takes only one parameter, the context, the same one you get from `msg.ctx`.

```python
@botflow
def example(self, flow: FlowRoot):
    first_step = flow.connect('first')
    second_step = first_step.connect('second',
                                     predicate=lambda ctx: 'mydata' in ctx)
    third_step = second_step.connect('third')
```

Now, after starting the flow with `!flows start example`, the state will be:

Errbot will wait for `!first`. But then, once the user executes `!first`, it will see that the predicate between `!first` and `!second` is verified, so will go on and execute `!second`, displaying 'Hello World' and proceed to wait for `!third`:

Branching in the graph

It is perfectly possible to branch out to several possibilities (possibly with different predicates).

```python
@botflow
def example(self, flow: FlowRoot):
    first_step = flow.connect('first')
    second_step = first_step.connect('second',
                                     predicate=lambda ctx: 'mydata' in ctx)
    other = first_step.connect('other_second',
                               predicate= lambda ctx: 'otherdata' in ctx)
```

This will do something like that:

In manual mode, the bot will tell the user about his 2 possible options to continue.

Making a looping graph

You can also perfectly reexecute a part of a graph in a "loop". You can branch directly the node object instead of the command name in that case.

```python
@botflow
def example(self, flow: FlowRoot):
    first_step = flow.connect('first')
    second_step = first_step.connect('second')
    third_step = second_step.connect(first_step, predicate=...)
    final_step = third_step.connect('final', predicate=...)
```

You can represent this flow like this:

The typical use case is to repeatedly ask something to the user.

6.5 [Advanced] Backend development

A backend is the glue code to connect Errbot to a chatting service. Starting with Errbot 2.3.0, backends can be developed out of the main repository. This documentation is there to guide you making a new backend for a chatting service but is also interesting to understand more core concepts of Errbot.

It is important to understand the core concepts of Errbot before starting to work on a backend.

6.5.1 Architecture

Backends are just a specialization of the bot, they are what is instanciated as the bot and are the entry point of the bot.

Following this logic a backend must inherit from `ErrBot`.

`ErrBot` inherits itself from *Backend*. This is where you can find what Errbot is expecting from backend.

You'll see a series of methods simply throwing the exception `NotImplementedError`. Those are the one you need to implement to fill up the blanks.

6.5.2 Identifiers

Every backend have a very specific way of addressing who, where and how to speak to somebody.

Lifecycle: identifiers are either created internally by the backend or externally by the plugins from *build_identifier()*.

There are 2 types of identifiers:

- a person
- a person in a chatroom

6.5.3 Identifier for a person

It is important to note that for some backends you can infer what a person is from what a person in a chatroom is, but for privacy reason you cannot on some backends ie. you can send a private message to a person in a chatroom but if the person leaves the room you have no way of knowing how to contect her/him personally.

Backends must implement a specific Identifier class that matches their way of identifying those.

For example Slack has a notion of userid and channeid you can find in the `SlackIdentifier` which is completely opaque to ErrBot itself.

But you need to implement a mapping from those private parameters to those properties:

- person: this needs to be a string that identifies a person. This should be enough info for the backend to contact this person. This should be a *secure* and sure way to identify somebody.

- client: this will identify optionally as a string additional information or channel from where this person is sending a message. For example, some backends open a one to one room to chat, or some backends identifies the current peripheral from which the person is sending a message from (mobile, web, . . .)

Some of those strings are completely unreadable for humans ie. *U00234FBE* for a person. So you need to provide more human readable info:

- nick: this would be the short name refering to that person. ie. *gbin*

- displayName: (optionally) this would give for example a full name ie. *Guillaume Binet*. This is often found in professional chatting services.

6.5.4 Identifier for a person in a chatroom

This is simply an Identifier with an added property: room. The string representation of room should give a charoom identifier (see below).

See for example `SlackMUCIdentifier`

6.5.5 Chatrooms / MUCRooms

In order to implement the various MUC related APIs you'll find from `Backend`, you'll need to implement a Room class. To help guide you, you can inherit from `MUCRoom` and fill up the blanks from the NotImplementedError.

Lifecycle: Those are created either internally by the backend or externally through `join_room()` from a string identifier.

6.6 [Advanced] Storage Plugin development

A storage plugin is the glue code that tells Errbot how to store the persistent data the plugins and the bot itself are producing. Starting with Errbot 3.3.0, storage plugins can be developed out of the main repository. This documentation is there to guide you making a new storage plugin so you can connect Errbot to your favorite database.

6.6.1 Architecture

Storage plugins are instantiated in 2 stages.

The first stage is storage plugin discovery and is similar to normal bot plugins:

- Errbot scans *errbot/storage* and *config.BOT_EXTRA_STORAGE_PLUGINS_DIR* for *.plug* files pointing to plugins implementing *StoragePluginBase*.

- Once the correct plugin from *config.STORAGE* is found, it is built with the bot config as its *__init__* parameter.

- By calling *super().__init__* on *StoragePluginBase*, Errbot will populate *self._storage_config* from *config.STORAGE_CONFIG*. This configuration should contain the custom parameters needed by your plugin to be able to connect to your database/storage ie. url, port, path, credentials ... You need to document them clearly so your users can set *config.STORAGE_CONFIG* correctly.

- As you can see in *StoragePluginBase*, you just have to implement the *open* method there.

The second stage is opening the storage, which is done using the *open* method:

- Various parts of Errbot may need separate key/value storage, the *open* method has a namespace to track those. For example, the internal *BotPluginManager* will open the namespace *core* to store the bot plugins and their config, the installed repos, etc.

- *open* needs to return a *StorageBase*, which exposes the various actions that Errbot can call on the storage (set, get, ...).

- You don't need to track the lifecycle of the storage, it will be enforced externally: no double-close, double-open, *get* after close, etc.

Plugins are `collections.MutableMapping` and use *StoreMixin* as an adapter from the mapping accessors to the *StorageBase* implementation.

6.6.2 Testing

Storage plugins are completely independent from Errbot itself. It should be easy to instantiate and test them externally.

6.6.3 Example

You can have a look at the internal shelf implementation *ShelfStorage*

6.7 Logging to Sentry

According to the official website. . .

> Sentry is an event logging platform primarily focused on capturing and aggregating exceptions.
>
> It was originally conceived at DISQUS in early 2010 to solve exception logging within a Django application. Since then it has grown to support many popular languages and platforms, including Python, PHP, Java, Ruby, Node.js, and even JavaScript.

6.7.1 Come again? Just what is Sentry, exactly?

The official documentation explains it better:

> Sentry is a realtime event logging and aggregation platform. At its core it specializes in monitoring errors and extracting all the information needed to do a proper post-mortem without any of the hassle of the standard user feedback loop.

If that sounds like something you'd want to gift your precious Errbot instance with, then do keep on reading :)

6.7.2 Setting up Sentry itself

Installing and configuring sentry is beyond the scope of this document. However, there are two options available to you. You can either get a hosted account, or grab the code and run your own server instead.

6.7.3 Configuring Errbot to use Sentry

Once you have an instance of Sentry available, you'll probably want to create a team specifically for Errbot first.

When you have, you should be able to access a page called "Client configuration". There, you will be presented with a so-called DSN value, which has the following format:

> http://000000000000000:00000000000000000@sentry.domain.tld/0

To setup Errbot with Sentry:

- Open up your bot's config.py

- Set **BOT_LOG_SENTRY** to *True* and fill in **SENTRY_DSN** with the DSN value obtained previously

- Optionally adjust **SENTRY_LOGLEVEL** to the desired level
- Optionally adjust **SENTRY_TRANSPORT** to the desired transport
- Restart Errbot

You can find a list of Sentry transport classes here.

You should now see Exceptions and log messages show up in your Sentry stream.

Getting involved

7.1 Contributing

If you would like to contribute to the project, please do not hesitate to get involved! Here you can find how best to get started.

7.1.1 Contributing to Errbot itself

Clone Errbot

All development on Errbot happens on GitHub. If you'd like to get involved, just fork the repository and make changes in your own repo. When you are satisfied with your changes, just open a pull request with us and we'll get it reviewed as soon as we can! Depending on our thoughts, we might decide to merge it in right away, or we may ask you to change certain parts before we will accept the change.

Run Errbot from source

Clone you github fork repo locally and install errbot in development mode from the root of the repo with:

```
pip install -e .
```

From there, anytime you execute *errbot* it will run from the checked out version of Errbot with all your local changes taken into account.

Preparing your pull request

In order to make the process easy for everyone involved, please follow these guidelines as you open a pull request.

- Make your changes on a separate branch, preferably giving it a descriptive name.

- Split your work up into smaller commits if possible, while making sure each commit can still function on its own. Do not commit work-in-progress code - commit it once it's working.

- Run tox before opening your pull request, and make sure all tests pass. You can install tox with **pip install tox**

- If you can, please add tests for your code. We know large parts of our codebase are missing tests, so we won't reject your code if it lacks tests, though.

7.1.2 Contributing documentation & making changes to the website

errbot.io is created using Sphinx, which also doubles as a generator for our (API) documentation. The code for it is in the same repository as Errbot itself, inside the docs folder. To make changes to the documentation or the website, you can build the HTML locally as follows:

```
# Change directory into the docs folder
cd docs/
# Install the required extra dependencies
pip install -r requirements.txt
# Generate the static HTML
make html
# Then, open the generated _build/html/index.html in a browser
```

To submit your changes back to us, please make your change in a separate branch as described in the previous section, then open a pull request with us.

Note: You must do this with Python 3, Python 2 is unsupported.

7.2 Issues and feature requests

Please report issues or feature requests on the issue tracker on GitHub.

When reporting issues, please be as specific as possible. Include things such as your Python version, platform, debug logs, and a description of what is happening. If you can tell us how to reproduce the issue ourselves, this makes it a lot easier for us to figure out what is going on, as well.

7.3 Getting help

The best place to get help if you get stuck with anything is to ask for advice on our Gitter chat room. If nobody is around to help you, opening an issue on the issue tracker is your next best option.

If you have a code-related question concerning (plugin) development it's best to ask your question on Stack Overflow, tagged errbot.

API documentation

8.1 errbot package

8.1.1 Subpackages

errbot.backends package

Submodules

errbot.backends.base module

class errbot.backends.base.**Backend**(_)

> Bases: abc.ABC

> Implements the basic Bot logic (logic independent from the backend) and leaves you to implement the missing parts.

> **MSG_ERROR_OCCURRED = 'Sorry for your inconvenience. An unexpected error occurred.'**

> **__init__**(_)
>> Those arguments will be directly those put in BOT_IDENTITY

> **build_identifier**(*text_representation: str*) → errbot.backends.base.Identifier

> **build_message**(*text: str*) → errbot.backends.base.Message
>> You might want to override this one depending on your backend

> **build_reply**(*msg: errbot.backends.base.Message, text: str = None, private: bool = False, threaded: bool = False*)
>> Should be implemented by the backend

> **callback_presence**(*presence: errbot.backends.base.Presence*) → None
>> Implemented by errBot.

`callback_room_joined`(*room: errbot.backends.base.Room*) → None
　　See `ErrBot`

`callback_room_left`(*room: errbot.backends.base.Room*) → None
　　See `ErrBot`

`callback_room_topic`(*room: errbot.backends.base.Room*) → None
　　See `ErrBot`

`change_presence`(*status: str = 'online'*, *message: str = ''*) → None
　　Signal a presence change for the bot. Should be overridden by backends with a super().send_message()
　　call.

`cmd_history = {}`

`connect`() → Any
　　Connects the bot to server or returns current connection

`connect_callback`() → None

`disconnect_callback`() → None

`is_from_self`(*msg: errbot.backends.base.Message*) → bool
　　Needs to be overridden to check if the incoming message is from the bot itself.

　　　　Return type `bool`

　　　　Parameters msg (*[Message](#)*) – The incoming message.

　　　　Returns True if the message is coming from the bot.

`mode`

`prefix_groupchat_reply`(*message: errbot.backends.base.Message*, *identifier: errbot.backends.base.Identifier*)
　　Patches message with the conventional prefix to ping the specific contact For example: @gbin, you forgot
　　the milk !

`query_room`(*room: str*) → errbot.backends.base.Room
　　Query a room for information.

　　　　Return type *[Room](#)*

　　　　Parameters room (`str`) – The room to query for.

　　　　Returns An instance of *[Room](#)*.

`reset_reconnection_count`() → None
　　Reset the reconnection count. Back-ends should call this after successfully connecting.

`rooms`
　　Return a list of rooms the bot is currently in.

　　　　Returns A list of *[Room](#)* instances.

`send_message`(*msg: errbot.backends.base.Message*) → None
　　Should be overridden by backends with a super().send_message() call.

`serve_forever`() → None
　　Connect the back-end to the server and serve forever.

　　Back-ends MAY choose to re-implement this method, in which case they are responsible for implementing
　　reconnection logic themselves.

　　Back-ends SHOULD trigger *[connect_callback()](#)* and *[disconnect_callback()](#)* themselves
　　after connection/disconnection.

serve_once() → None
> Connect the back-end to the server and serve a connection once (meaning until disconnected for any reason).
>
> Back-ends MAY choose not to implement this method, IF they implement a custom *serve_forever()*.
>
> This function SHOULD raise an exception or return a value that evaluates to False in order to signal something went wrong. A return value that evaluates to True will signal the bot that serving is done and a shut-down is requested.

class errbot.backends.base.**Card**(*body: str = ''*, *frm: errbot.backends.base.Identifier = None*, *to: errbot.backends.base.Identifier = None*, *parent: errbot.backends.base.Message = None*, *summary: str = None*, *title: str = ''*, *link: str = None*, *image: str = None*, *thumbnail: str = None*, *color: str = None*, *fields: Tuple[Tuple[str, str]] = ()*)
> Bases: *errbot.backends.base.Message*

Card is a special type of preformatted message. If it matches with a backend similar concept like on Slack or Hipchat it will be rendered natively, otherwise it will be sent as a regular message formatted with the card.md template.

> **__init__**(*body: str = ''*, *frm: errbot.backends.base.Identifier = None*, *to: errbot.backends.base.Identifier = None*, *parent: errbot.backends.base.Message = None*, *summary: str = None*, *title: str = ''*, *link: str = None*, *image: str = None*, *thumbnail: str = None*, *color: str = None*, *fields: Tuple[Tuple[str, str]] = ()*)
> > Creates a Card. :type color: `str` :type thumbnail: `str` :type image: `str` :type link: `str` :type title: `str` :type summary: `str` :type parent: *Message* :type to: *Identifier* :type frm: *Identifier* :type body: `str` :param body: main text of the card in markdown. :param frm: the card is sent from this identifier. :param to: the card is sent to this identifier (Room, RoomOccupant, Person. . .). :param parent: the parent message this card replies to. (threads the message if the backend supports it). :param summary: (optional) One liner summary of the card, possibly collapsed to it. :param title: (optional) Title possibly linking. :param link: (optional) url the title link is pointing to. :param image: (optional) link to the main image of the card. :param thumbnail: (optional) link to an icon / thumbnail. :param color: (optional) background color or color indicator. :param fields: (optional) a tuple of (key, value) pairs.

> **color**

> **fields**

> **image**

> **link**

> **summary**

> **text_color**

> **thumbnail**

> **title**

class errbot.backends.base.**Identifier**
> Bases: `abc.ABC`

> This is just use for type hinting representing the Identifier contract, NEVER TRY TO SUBCLASS IT OUTSIDE OF A BACKEND, it is just here to show you what you can expect from an Identifier. To get an instance of a real identifier, always use the properties from Message (to, from) or self.build_identifier

> > to make an identifier from a String.

> > The semantics is anything you can talk to: Person, Room, RoomOccupant etc.

class errbot.backends.base.**Message**(*body: str = ''*, *frm: errbot.backends.base.Identifier =*
None, *to: errbot.backends.base.Identifier = None*, *par-*
ent: Optional[errbot.backends.base.Message] = None, *de-*
layed: bool = False, *partial: bool = False*, *extras: Map-*
ping[KT, VT_co] = None, *flow=None*)

Bases: `object`

A chat message.

This class represents chat messages that are sent or received by the bot.

__init__(*body: str = ''*, *frm: errbot.backends.base.Identifier = None*, *to: er-*
rbot.backends.base.Identifier = None, *parent: Optional[errbot.backends.base.Message] =*
None, *delayed: bool = False*, *partial: bool = False*, *extras: Mapping[KT, VT_co] = None*,
flow=None)

Parameters

- **body** (`str`) – The markdown body of the message.

- **extras** – Extra data attached by a backend

- **flow** – The flow in which this message has been triggered.

- **parent** – The parent message of this message in a thread. (Not supported by all back-ends)

- **partial** (`bool`) – Indicates whether the message was obtained by breaking down the message to fit the `MESSAGE_SIZE_LIMIT`.

body

Get the plaintext body of the message.

Returns The body as a string.

clone()

delayed

extras

flow

Get the conversation flow for this message.

Returns A *Flow*

frm

Get the sender of the message.

Returns An *Identifier* identifying the sender.

is_direct

is_group

is_threaded

parent

partial

to

Get the recipient of the message.

Returns A backend specific identifier representing the recipient.

---

class errbot.backends.base.**Person**

Bases: *errbot.backends.base.Identifier*

This is just use for type hinting representing the Identifier contract, NEVER TRY TO SUBCLASS IT OUTSIDE OF A BACKEND, it is just here to show you what you can expect from an Identifier. To get an instance of a real identifier, always use the properties from Message (to, from) or self.build_identifier

to make an identifier from a String.

aclattr

> **Returns** returns the unique identifier that will be used for ACL matches.

client

> **Returns** a backend specific unique identifier representing the device or client the person is using to talk.

fullname

Some backends have the full name of a user.

> **Returns** the fullname of this user if available.

nick

> **Returns** a backend specific nick returning the nickname of this person if available.

person

> **Returns** a backend specific unique identifier representing the person you are talking to.

class errbot.backends.base.**Presence**(*identifier: errbot.backends.base.Identifier*, *status: str = None*, *message: str = None*)

Bases: object

This class represents a presence change for a user or a user in a chatroom.

Instances of this class are passed to *callback_presence()* when the presence of people changes.

__init__(*identifier: errbot.backends.base.Identifier*, *status: str = None*, *message: str = None*)

Initialize self. See help(type(self)) for accurate signature.

identifier

Identifier for whom its status changed. It can be a RoomOccupant or a Person. :return: the person or roomOccupant

message

Returns a human readable message associated with the status if any. like : "BRB, washing the dishes" It can be None if it is only a general status update (see get_status)

status

Returns the status of the presence change. It can be one of the constants ONLINE, OFFLINE, AWAY, DND, but can also be custom statuses depending on backends. It can be None if it is just an update of the status message (see get_message)

class errbot.backends.base.**Room**

Bases: *errbot.backends.base.Identifier*

This class represents a Multi-User Chatroom.

create() → None

Create the room.

Calling this on an already existing room is a no-op.

destroy() → None
    Destroy the room.

    Calling this on a non-existing room is a no-op.

exists
    Boolean indicating whether this room already exists or not.

        Getter Returns *True* if the room exists, *False* otherwise.

invite(**args*) → None
    Invite one or more people into the room.

        *argsargs One or more identifiers to invite into the room.

join(*username: str = None*, *password: str = None*) → None
    Join the room.

    If the room does not exist yet, this will automatically call [*create()*](#) on it first.

joined
    Boolean indicating whether this room has already been joined.

        Getter Returns *True* if the room has been joined, *False* otherwise.

leave(*reason: str = None*) → None
    Leave the room.

        Parameters **reason** (`str`) – An optional string explaining the reason for leaving the room.

occupants
    The room's occupants.

        Getter Returns a list of occupant identities.

        Raises `MUCNotJoinedError` if the room has not yet been joined.

topic
    The room topic.

        Getter Returns the topic (a string) if one is set, *None* if no topic has been set at all.

        Note: Back-ends may return an empty string rather than *None* when no topic has been set as a network may not differentiate between no topic and an empty topic.

        Raises `MUCNotJoinedError` if the room has not yet been joined.

exception errbot.backends.base.**RoomDoesNotExistError**
    Bases: [*errbot.backends.base.RoomError*](#)

    Exception that is raised when performing an operation on a room that doesn't exist

exception errbot.backends.base.**RoomError**
    Bases: `Exception`

    General exception class for MUC-related errors

exception errbot.backends.base.**RoomNotJoinedError**
    Bases: [*errbot.backends.base.RoomError*](#)

    Exception raised when performing MUC operations that require the bot to have joined the room

class errbot.backends.base.**RoomOccupant**
    Bases: [*errbot.backends.base.Identifier*](#)

> **room**
>> Some backends have the full name of a user.
>>
>>> **Returns** the fullname of this user if available.

class errbot.backends.base.**Stream**(*identifier: errbot.backends.base.Identifier*, *fsource: BinaryIO*, *name: str = None*, *size: int = None*, *stream_type: str = None*)

> Bases: _io.BufferedReader

> This class represents a stream request.

> Instances of this class are passed to *callback_stream()* when an incoming stream is requested.

> **__init__**(*identifier: errbot.backends.base.Identifier*, *fsource: BinaryIO*, *name: str = None*, *size: int = None*, *stream_type: str = None*)
>> Initialize self. See help(type(self)) for accurate signature.

> **accept**() → None
>> Signal that the stream has been accepted.

> **ack_data**(*length: int*) → None
>> Acknowledge data has been transfered.

> **clone**(*new_fsource: BinaryIO*) → errbot.backends.base.Stream
>> Creates a clone and with an alternative stream

> **error**(*reason='unknown'*) → None
>> An internal plugin error prevented the transfer.

> **identifier**
>> The identity the stream is coming from if it is an incoming request or to if it is an outgoing request.

> **name**
>> The name of the stream/file if it has one or None otherwise. !! Be carefull of injections if you are using this name directly as a filename.

> **reject**() → None
>> Signal that the stream has been rejected.

> **size**
>> The expected size in bytes of the stream if it is known or None.

> **status**
>> The status for this stream.

> **stream_type**
>> The mimetype of the stream if it is known or None.

> **success**() → None
>> The streaming finished normally.

> **transfered**
>> The currently transfered size.

exception errbot.backends.base.**UserDoesNotExistError**
> Bases: Exception

> Exception that is raised when performing an operation on a user that doesn't exist

errbot.backends.hipchat module

class errbot.backends.hipchat.**HipChatRoom**(*name*, *bot*)

    Bases: *errbot.backends.base.Room*

    This class represents a Multi-User Chatroom.

    __init__(*name*, *bot*)

        Parameters **name** – The name of the room

    create(*privacy='public'*, *guest_access=False*)

        Create the room.

        Calling this on an already existing room is a no-op.

        Parameters

            • **privacy** (str) – Whether the room is available for access by other users or not. Valid values are "public" and "private".

            • **guest_access** (bool) – Whether or not to enable guest access for this room.

    destroy()

        Destroy the room.

        Calling this on a non-existing room is a no-op.

    domain

    exists

        Boolean indicating whether this room already exists or not.

        Getter Returns *True* if the room exists, *False* otherwise.

    invite(**args*)

        Invite one or more people into the room.

        Parameters **args** – One or more people to invite into the room. May be the mention name (beginning with an @) or "FirstName LastName" of the user you wish to invite.

    jid

    join(*username=None*, *password=None*)

        Join the room.

        If the room does not exist yet, this will automatically call *create()* on it first.

    joined

        Boolean indicating whether this room has already been joined or not.

        Getter Returns *True* if the room has been joined, *False* otherwise.

    leave(*reason=None*)

        Leave the room.

        Parameters **reason** – An optional string explaining the reason for leaving the room

    name

        The name of this room

    node

    notify(*message*, *color=None*, *notify=False*, *message_format=None*)

        Send a notification to a room.

        See the HipChat API documentation for more info.

occupants
   The room's occupants.

      Getter Returns a list of `HipChatMUCOccupant` instances.

resource

room
   Return room information from the HipChat API

topic
   The room topic.

      Getter Returns the topic (a string) if one is set, *None* if no topic has been set at all.

class errbot.backends.hipchat.**HipChatRoomOccupant**(*node=None*, *domain=None*, *re-source=None*, *room=None*, *hipchat_user=None*, *aclattr=None*)

   Bases: [`errbot.backends.xmpp.XMPPRoomOccupant`](#)

   An occupant of a Multi-User Chatroom.

   This class has all the attributes that are returned by a call to https://www.hipchat.com/docs/apiv2/method/get_all_participants with the link to self expanded.

   __init__(*node=None*, *domain=None*, *resource=None*, *room=None*, *hipchat_user=None*, *aclattr=None*)

      Parameters hipchat_user – A user object as returned by https://www.hipchat.com/docs/apiv2/method/get_all_participants with the link to self expanded.

   aclattr

      Returns returns the unique identifier that will be used for ACL matches.

class errbot.backends.hipchat.**HipchatBackend**(*config*)
   Bases: [`errbot.backends.xmpp.XMPPBackend`](#)

   __init__(*config*)
      Those arguments will be directly those put in BOT_IDENTITY

   build_reply(*msg*, *text=None*, *private=False*, *threaded=False*)
      Should be implemented by the backend

   callback_message(*msg*)
      Processes for commands and dispatches the message to all the plugins.

   create_connection()

   mode

   prefix_groupchat_reply(*message*, *identifier*)
      Patches message with the conventional prefix to ping the specific contact For example: @gbin, you forgot the milk !

   query_room
      Query a room for information.

      Parameters room – The name (preferred) or XMPP JID of the room to query for.

      Returns An instance of [`HipChatRoom`](#).

   room_factory
      alias of [`HipChatRoom`](#)

roomoccupant_factory
   alias of *HipChatRoomOccupant*

rooms()
   Return a list of rooms the bot is currently in.

> **Returns** A list of *HipChatRoom* instances.

send_card(*card*)
   Sends a card, this can be overriden by the backends *without* a super() call.

> **Parameters** **card** – the card to send.

> **Returns** None

send_stream_request(*identifier*, *fsource*, *name='file.txt'*, *size=None*, *stream_type=None*)
   Starts a file transfer. note, fsource used to make the stream needs to be in open/rb state

class errbot.backends.hipchat.**HipchatClient**(**args*, ***kwargs*)
   Bases: *errbot.backends.xmpp.XMPPConnection*

__init__(**args*, ***kwargs*)
   Initialize self. See help(type(self)) for accurate signature.

users
   A list of all the users.

   See also: https://www.hipchat.com/docs/apiv2/method/get_all_users

class errbot.backends.hipchat.**HipchatExtension**(**args*, ***kwargs*)
   Bases: markdown.extensions.Extension

   Removes the unsupported html tags from hipchat

extendMarkdown(*md*, *md_globals*)
   Add the various proccesors and patterns to the Markdown Instance.

   This method must be overriden by every extension.

   Keyword arguments:

   • md: The Markdown instance.

   • md_globals: Global variables in the markdown module namespace.

class errbot.backends.hipchat.**HipchatTreeprocessor**(*markdown_instance=None*)
   Bases: markdown.treeprocessors.Treeprocessor

run(*root*)
   Subclasses of Treeprocessor should implement a *run* method, which takes a root ElementTree. This method can return another ElementTree object, and the existing root ElementTree will be replaced, or it can modify the current tree and return None.

errbot.backends.hipchat.**hipchat_html**()

errbot.backends.irc module

class errbot.backends.irc.**IRCBackend**(*config*)
   Bases: *errbot.core.ErrBot*

__init__(*config*)
   Those arguments will be directly those put in BOT_IDENTITY

aclpattern = '{nick}!{user}@{host}'

build_identifier(*txtrep*)

build_message(*text*)
> You might want to override this one depending on your backend

build_reply(*msg*, *text=None*, *private=False*, *threaded=False*)
> Should be implemented by the backend

change_presence(*status: str = 'online'*, *message: str = ''*) → None
> Signal a presence change for the bot. Should be overridden by backends with a super().send_message() call.

connect()
> Connects the bot to server or returns current connection

mode

prefix_groupchat_reply(*message*, *identifier*)
> Patches message with the conventional prefix to ping the specific contact For example: @gbin, you forgot the milk !

query_room(*room*)
> Query a room for information.
>
>> Parameters **room** – The channel name to query for.
>>
>> Returns An instance of `IRCMUCRoom`.

rooms()
> Return a list of rooms the bot is currently in.
>
>> Returns A list of `IRCMUCRoom` instances.

send_message(*msg*)
> This needs to be overridden by the backends with a super() call.
>
>> Parameters **msg** – the message to send.
>>
>> Returns None

send_stream_request(*identifier*, *fsource*, *name=None*, *size=None*, *stream_type=None*)

serve_forever()
> Connect the back-end to the server and serve forever.
>
> Back-ends MAY choose to re-implement this method, in which case they are responsible for implementing reconnection logic themselves.
>
> Back-ends SHOULD trigger `connect_callback()` and `disconnect_callback()` themselves after connection/disconnection.

shutdown()

class errbot.backends.irc.**IRCConnection**(*bot*, *nickname*, *server*, *port=6667*, *ssl=False*, *bind_address=None*, *ipv6=False*, *password=None*, *username=None*, *nickserv_password=None*, *private_rate=1*, *channel_rate=1*, *reconnect_on_kick=5*, *reconnect_on_disconnect=5*)
> Bases: `irc.bot.SingleServerIRCBot`

> **__init__**(*bot*, *nickname*, *server*, *port=6667*, *ssl=False*, *bind_address=None*, *ipv6=False*, *password=None*, *username=None*, *nickserv_password=None*, *private_rate=1*, *channel_rate=1*, *reconnect_on_kick=5*, *reconnect_on_disconnect=5*)
>> Initialize self. See help(type(self)) for accurate signature.

away (*message=''*)

> Extend the original implementation to support AWAY. To set an away message, set message to something. To cancel an away message, leave message at empty string.

connect (**args*, ***kwargs*)

> Connect using the underlying connection

on_currenttopic (*connection*, *event*)

> When you Join a room with a topic set this event fires up to with the topic information. If the room that you join don't have a topic set, nothing happens. Here is NOT the place to fire the `callback_room_topic()` event for that case exist on_topic.
>
> > **Parameters**
> >
> > - **connection** – Is an 'irc.client.ServerConnection' object
> >
> > - **event** – Is an 'irc.client.Event' object The event.arguments[0] contains the room name The event.arguments[1] contains the topic of the room.

on_dcc_connect (*dcc*, *event*)

on_dcc_disconnect (*dcc*, *event*)

on_dccmsg (*dcc*, *event*)

on_disconnect (*connection*, *event*)

on_endofnames (*connection*, *event*)

> Handler of the enfofnames IRC message/event.
>
> The endofnames message is sent to the client when the server finish to send the list of names of the room ocuppants. This usually happens when you join to the room. So in this case, we use this event to determine that our bot is finally joined to the room.
>
> > **Parameters**
> >
> > - **connection** – Is an 'irc.client.ServerConnection' object
> >
> > - **event** – Is an 'irc.client.Event' object the event.arguments[0] contains the channel name

on_join (*connection*, *event*)

> Handler of the join IRC message/event. Is in response of a /JOIN client message.
>
> > **Parameters**
> >
> > - **connection** – Is an 'irc.client.ServerConnection' object
> >
> > - **event** – Is an 'irc.client.Event' object the event.target contains the channel name

on_kick (*_*, *e*)

on_notopic (*connection*, *event*)

> This event fires ip when there is no topic set on a room
>
> > **Parameters**
> >
> > - **connection** – Is an 'irc.client.ServerConnection' object
> >
> > - **event** – Is an 'irc.client.Event' object The event.arguments[0] contains the room name

on_part (*connection*, *event*)

> Handler of the part IRC Message/event.
>
> The part message is sent to the client as a confirmation of a /PART command sent by someone in the room/channel. If the event.source contains the bot nickname then we need to fire the `callback_room_left()` event on the bot.

Parameters

- **connection** – Is an 'irc.client.ServerConnection' object

- **event** – Is an 'irc.client.Event' object The event.source contains the nickmask of the user that leave the room The event.target contains the channel name

on_privmsg(_, *e*)

on_privnotice(_, *e*)

on_pubmsg(_, *e*)

on_pubnotice(_, *e*)

on_topic(*connection*, *event*)

> On response to the /TOPIC command if the room have a topic. If the room don't have a topic the event fired is on_notopic :param connection: Is an 'irc.client.ServerConnection' object
>
> > Parameters **event** – Is an 'irc.client.Event' object The event.target contains the room name. The event.arguments[0] contains the topic name

on_welcome(_, *e*)

static send_chunk(*stream*, *dcc*)

send_private_message(*to*, *line*)

send_public_message(*to*, *line*)

send_stream_request(*identifier*, *fsource*, *name=None*, *size=None*, *stream_type=None*)

class errbot.backends.irc.**IRCPerson**(*mask*)

> Bases: *errbot.backends.base.Person*
>
> **__init__**(*mask*)
>
> > Initialize self. See help(type(self)) for accurate signature.
>
> **aclattr**
>
> > Returns returns the unique identifier that will be used for ACL matches.
>
> **client**
>
> > Returns a backend specific unique identifier representing the device or client the person is using to talk.
>
> **fullname**
>
> > Some backends have the full name of a user.
> >
> > Returns the fullname of this user if available.
>
> **host**
>
> **nick**
>
> > Returns a backend specific nick returning the nickname of this person if available.
>
> **person**
>
> > Returns a backend specific nick returning the nickname of this person if available.
>
> **user**

class errbot.backends.irc.**IRCRoom**(*room*, *bot*)

> Bases: *errbot.backends.base.Room*

Represent the specifics of a IRC Room/Channel.

This lifecycle of this object is:

- Created in IRCConnection.on_join
- The joined status change in IRCConnection on_join/on_part
- Deleted/destroyed in IRCConnection.on_disconnect

__init__(*room*, *bot*)
> Initialize self. See help(type(self)) for accurate signature.

cb_set_topic(*current_topic*)
> Store the current topic for this room.
>
> This method is called by the IRC backend when a *currenttopic*, *topic* or *notopic* IRC event is received to store the topic set for this channel.
>
> This function is not meant to be executed by regular plugins. To get or set

create()
> Not supported on this back-end. Will join the room to ensure it exists, instead.

destroy()
> Not supported on IRC, will raise *RoomError*.

exists
> Boolean indicating whether this room already exists or not.
>
> > **Getter** Returns *True* if the room exists, *False* otherwise.

invite(**args*)
> Invite one or more people into the room.
>
> > ***argsargs** One or more nicks to invite into the room.

join(*username=None*, *password=None*)
> Join the room.
>
> If the room does not exist yet, this will automatically call *create()* on it first.

joined
> Boolean indicating whether this room has already been joined.
>
> > **Getter** Returns *True* if the room has been joined, *False* otherwise.

leave(*reason=None*)
> Leave the room.
>
> > **Parameters** **reason** – An optional string explaining the reason for leaving the room

occupants
> The room's occupants.
>
> > **Getter** Returns a list of occupants. :raises: MUCNotJoinedError if the room has not yet been joined.

topic
> The room topic.
>
> > **Getter** Returns the topic (a string) if one is set, *None* if no topic has been set at all.

class errbot.backends.irc.**IRCRoomOccupant**(*mask*, *room*)
> Bases: *errbot.backends.irc.IRCPerson*, *errbot.backends.base.RoomOccupant*
>
> **__init__**(*mask*, *room*)
> > Initialize self. See help(type(self)) for accurate signature.

---

room
> Some backends have the full name of a user.

> > **Returns** the fullname of this user if available.

errbot.backends.irc.**irc_md**()
> This makes a converter from markdown to mirc color format.

errbot.backends.null module

errbot.backends.slack module

errbot.backends.telegram_messenger module

exception errbot.backends.telegram_messenger.**RoomsNotSupportedError**(*message=None*)
> Bases: *errbot.backends.base.RoomError*

> **__init__**(*message=None*)
> > Initialize self. See help(type(self)) for accurate signature.

class errbot.backends.telegram_messenger.**TelegramBackend**(*config*)
> Bases: *errbot.core.ErrBot*

> **__init__**(*config*)
> > Those arguments will be directly those put in BOT_IDENTITY

> **build_identifier**(*txtrep*)
> > Convert a textual representation into a *TelegramPerson* or *TelegramRoom*.

> **build_reply**(*msg*, *text=None*, *private=False*, *threaded=False*)
> > Should be implemented by the backend

> **change_presence**(*status: str = 'online'*, *message: str = ''*) → None
> > Signal a presence change for the bot. Should be overridden by backends with a super().send_message() call.

> **mode**

> **prefix_groupchat_reply**(*message*, *identifier*)
> > Patches message with the conventional prefix to ping the specific contact For example: @gbin, you forgot the milk !

> **query_room**(*room*)
> > Not supported on Telegram.

> > **Raises** *RoomsNotSupportedError*

> **rooms**()
> > Not supported on Telegram.

> > **Raises** *RoomsNotSupportedError*

> **send_message**(*msg*)
> > This needs to be overridden by the backends with a super() call.

> > **Parameters** **msg** – the message to send.

> > **Returns** None

> **send_stream_request**(*identifier*, *fsource*, *name='file'*, *size=None*, *stream_type=None*)
> > Starts a file transfer.

Parameters

- **identifier** – TelegramPerson or TelegramMUCOccupant Identifier of the Person or Room to send the stream to.

- **fsource** – str, dict or binary data File URL or binary content from a local file. Optionally a dict with binary content plus metadata can be given. See *stream_type* for more details.

- **name** (str) – str, optional Name of the file. Not sure if this works always.

- **size** – str, optional Size of the file obtained with os.path.getsize. This is only used for debug logging purposes.

- **stream_type** – str, optional Type of the stream. Choices: 'document', 'photo', 'audio', 'video', 'sticker', 'location'.

  If 'video', a dict is optional as {'content': fsource, 'duration': str}. If 'voice', a dict is optional as {'content': fsource, 'duration': str}. If 'audio', a dict is optional as {'content': fsource, 'duration': str, 'performer': str, 'title': str}.

  For 'location' a dict is mandatory as {'latitude': str, 'longitude': str}. For 'venue': TODO # see: https://core.telegram.org/bots/api#sendvenue

Return stream str or Stream If *fsource* is str will return str, else return Stream.

serve_once()
> Connect the back-end to the server and serve a connection once (meaning until disconnected for any reason).

> Back-ends MAY choose not to implement this method, IF they implement a custom serve_forever().

> This function SHOULD raise an exception or return a value that evaluates to False in order to signal something went wrong. A return value that evaluates to True will signal the bot that serving is done and a shut-down is requested.

class errbot.backends.telegram_messenger.**TelegramBotFilter**
> Bases: object

This is a filter for the logging library that filters the "No new updates found." log message generated by telegram.bot.

This is an INFO-level log message that gets logged for every getUpdates() call where there are no new messages, so is way too verbose.

static filter(*record*)

class errbot.backends.telegram_messenger.**TelegramIdentifier**(*id*)
> Bases: *errbot.backends.base.Identifier*

__init__(*id*)
> Initialize self. See help(type(self)) for accurate signature.

aclattr

id

class errbot.backends.telegram_messenger.**TelegramMUCOccupant**(*id*, *room*, *first_name=None*, *last_name=None*, *username=None*)
> Bases: *errbot.backends.telegram_messenger.TelegramPerson*, *errbot.backends.base.RoomOccupant*

This class represents a person inside a MUC.

__init__(*id*, *room*, *first_name=None*, *last_name=None*, *username=None*)
    Initialize self. See help(type(self)) for accurate signature.

room
    Some backends have the full name of a user.

>   **Returns** the fullname of this user if available.

username

class errbot.backends.telegram_messenger.**TelegramPerson**(*id*, *first_name=None*, *last_name=None*, *username=None*)
    Bases: *errbot.backends.telegram_messenger.TelegramIdentifier*, *errbot.backends.base.Person*

__init__(*id*, *first_name=None*, *last_name=None*, *username=None*)
    Initialize self. See help(type(self)) for accurate signature.

client

>   **Returns** a backend specific unique identifier representing the device or client the person is using to talk.

first_name

fullname
    Some backends have the full name of a user.

>   **Returns** the fullname of this user if available.

id

last_name

nick

person

username

class errbot.backends.telegram_messenger.**TelegramRoom**(*id*, *title=None*)
    Bases: *errbot.backends.telegram_messenger.TelegramIdentifier*, *errbot.backends.base.Room*

__init__(*id*, *title=None*)
    Initialize self. See help(type(self)) for accurate signature.

create()
    Create the room.

    Calling this on an already existing room is a no-op.

destroy()
    Destroy the room.

    Calling this on a non-existing room is a no-op.

exists
    Boolean indicating whether this room already exists or not.

>   **Getter** Returns *True* if the room exists, *False* otherwise.

id

invite(**args*)
    Invite one or more people into the room.

> > ***argsargs** One or more identifiers to invite into the room.

join (*username: str = None*, *password: str = None*)
> Join the room.
>
> If the room does not exist yet, this will automatically call *create()* on it first.

joined
> Boolean indicating whether this room has already been joined.
>
> > **Getter** Returns *True* if the room has been joined, *False* otherwise.

leave (*reason: str = None*)
> Leave the room.
>
> > **Parameters** **reason** (str) – An optional string explaining the reason for leaving the room.

occupants
> The room's occupants.
>
> > **Getter** Returns a list of occupant identities.
> >
> > **Raises** MUCNotJoinedError if the room has not yet been joined.

title
> Return the groupchat title (only applies to groupchats)

topic
> The room topic.
>
> > **Getter** Returns the topic (a string) if one is set, *None* if no topic has been set at all.

> > ---
> > **Note:** Back-ends may return an empty string rather than *None* when no topic has been set as a network may not differentiate between no topic and an empty topic.
> > ---

> > **Raises** MUCNotJoinedError if the room has not yet been joined.

errbot.backends.test module

errbot.backends.text module

class errbot.backends.text.**TextBackend** (*config*)
> Bases: *errbot.core.ErrBot*

> **__init__** (*config*)
> > Those arguments will be directly those put in BOT_IDENTITY

> **add_reaction** (*msg: errbot.backends.base.Message*, *reaction: str*) → None

> **build_identifier** (*text_representation*)

> **build_reply** (*msg*, *text=None*, *private=False*, *threaded=False*)
> > Should be implemented by the backend

> **change_presence** (*status: str = 'online'*, *message: str = ''*) → None
> > Signal a presence change for the bot. Should be overridden by backends with a super().send_message() call.

> **mode**

prefix_groupchat_reply(*message*, *identifier*)
> Patches message with the conventional prefix to ping the specific contact For example: @gbin, you forgot the milk !

query_room(*room*)
> Query a room for information.
>
> > **Parameters** **room** – The room to query for.
> >
> > **Returns** An instance of `Room`.

readline_support()

remove_reaction(*msg: errbot.backends.base.Message*, *reaction: str*) → None

rooms
> Return a list of rooms the bot is currently in.
>
> > **Returns** A list of *[Room](#)* instances.

send_message(*msg*)
> This needs to be overridden by the backends with a super() call.
>
> > **Parameters** **msg** – the message to send.
> >
> > **Returns** None

serve_forever()
> Connect the back-end to the server and serve forever.
>
> Back-ends MAY choose to re-implement this method, in which case they are responsible for implementing reconnection logic themselves.
>
> Back-ends SHOULD trigger `connect_callback()` and `disconnect_callback()` themselves after connection/disconnection.

class errbot.backends.text.**TextOccupant**(*person*, *room*)
> Bases: *[errbot.backends.text.TextPerson](#)*, *[errbot.backends.base.RoomOccupant](#)*
>
> **__init__**(*person*, *room*)
> > Initialize self. See help(type(self)) for accurate signature.
>
> **room**
> > Some backends have the full name of a user.
> >
> > > **Returns** the fullname of this user if available.

class errbot.backends.text.**TextPerson**(*person*, *client=None*, *nick=None*, *fullname=None*)
> Bases: *[errbot.backends.base.Person](#)*
>
> Simple Person implementation which represents users as simple text strings.
>
> **__init__**(*person*, *client=None*, *nick=None*, *fullname=None*)
> > Initialize self. See help(type(self)) for accurate signature.
>
> **aclattr**
> > > **Returns** returns the unique identifier that will be used for ACL matches.
>
> **client**
> > > **Returns** a backend specific unique identifier representing the device or client the person is using to talk.
>
> **fullname**
> > Some backends have the full name of a user.

> **Returns** the fullname of this user if available.

nick

> **Returns** a backend specific nick returning the nickname of this person if available.

person

> **Returns** a backend specific unique identifier representing the person you are talking to.

class errbot.backends.text.**TextRoom**(*name*, *bot*)

Bases: *errbot.backends.base.Room*

__init__(*name*, *bot*)

Initialize self. See help(type(self)) for accurate signature.

create()

Create the room.

Calling this on an already existing room is a no-op.

destroy()

Destroy the room.

Calling this on a non-existing room is a no-op.

exists

Boolean indicating whether this room already exists or not.

> **Getter** Returns *True* if the room exists, *False* otherwise.

invite(**args*)

Invite one or more people into the room.

> ***argsargs** One or more identifiers to invite into the room.

join(*username=None*, *password=None*)

Join the room.

If the room does not exist yet, this will automatically call *create()* on it first.

joined

Boolean indicating whether this room has already been joined.

> **Getter** Returns *True* if the room has been joined, *False* otherwise.

leave(*reason=None*)

Leave the room.

> **Parameters** **reason** – An optional string explaining the reason for leaving the room.

occupants

The room's occupants.

> **Getter** Returns a list of occupant identities.

> **Raises** MUCNotJoinedError if the room has not yet been joined.

topic

The room topic.

> **Getter** Returns the topic (a string) if one is set, *None* if no topic has been set at all.

> ---
>
> **Note:** Back-ends may return an empty string rather than *None* when no topic has been set as a network may not differentiate between no topic and an empty topic.
>
> ---

> **Raises** MUCNotJoinedError if the room has not yet been joined.

errbot.backends.text.**borderless_ansi**()

> This makes a converter from markdown to ansi (console) format. It can be called like this: from errbot.rendering import ansi md_converter = ansi() # you need to cache the converter
>
> ansi_txt = md_converter.convert(md_txt)

errbot.backends.xmpp module

class errbot.backends.xmpp.**XMPPBackend**(*config*)

> Bases: *errbot.core.ErrBot*
>
> **__init__**(*config*)
>
> > Those arguments will be directly those put in BOT_IDENTITY
>
> **build_identifier**
>
> **build_reply**(*msg*, *text=None*, *private=False*, *threaded=False*)
>
> > Should be implemented by the backend
>
> **change_presence**(*status: str = 'online'*, *message: str = ''*) → None
>
> > Signal a presence change for the bot. Should be overridden by backends with a super().send_message() call.
>
> **chat_topic**(*event*)
>
> **connected**(*data*)
>
> > Callback for connection events
>
> **contact_offline**(*event*)
>
> **contact_online**(*event*)
>
> **create_connection**()
>
> **disconnected**(*data*)
>
> > Callback for disconnection events
>
> **incoming_message**(*xmppmsg*)
>
> > Callback for message events
>
> **mode**
>
> **prefix_groupchat_reply**(*message*, *identifier*)
>
> > Patches message with the conventional prefix to ping the specific contact For example: @gbin, you forgot the milk !
>
> **query_room**(*room*)
>
> > Query a room for information.
> >
> > > **Parameters** **room** – The JID/identifier of the room to query for.
> > >
> > > **Returns** An instance of XMPPMUCRoom.
>
> **room_factory**
>
> > alias of *XMPPRoom*
>
> **roomoccupant_factory**
>
> > alias of *XMPPRoomOccupant*
>
> **rooms**()
>
> > Return a list of rooms the bot is currently in.

> > **Returns** A list of `XMPPMUCRoom` instances.

send_message(*msg*)
This needs to be overridden by the backends with a super() call.

> **Parameters** `msg` – the message to send.

> **Returns** None

serve_forever()
Connect the back-end to the server and serve forever.

Back-ends MAY choose to re-implement this method, in which case they are responsible for implementing reconnection logic themselves.

Back-ends SHOULD trigger `connect_callback()` and `disconnect_callback()` themselves after connection/disconnection.

user_changed_status(*event*)

user_joined_chat(*event*)

user_left_chat(*event*)

class errbot.backends.xmpp.**XMPPConnection**(*jid*, *password*, *feature=None*, *keepalive=None*, *ca_cert=None*, *server=None*, *use_ipv6=None*, *bot=None*)

Bases: `object`

__init__(*jid*, *password*, *feature=None*, *keepalive=None*, *ca_cert=None*, *server=None*, *use_ipv6=None*, *bot=None*)
Initialize self. See help(type(self)) for accurate signature.

add_event_handler(*name*, *cb*)

connect()

del_event_handler(*name*, *cb*)

disconnect()

serve_forever()

session_start(*_*)

class errbot.backends.xmpp.**XMPPIdentifier**(*node*, *domain*, *resource*)
Bases: *errbot.backends.base.Identifier*

This class is the parent and the basic contract of all the ways the backends are identifying a person on their system.

__init__(*node*, *domain*, *resource*)
Initialize self. See help(type(self)) for accurate signature.

client

domain

fullname

nick

node

person

resource

class errbot.backends.xmpp.**XMPPPerson**(*node*, *domain*, *resource*)
> Bases: *errbot.backends.xmpp.XMPPIdentifier*, *errbot.backends.base.Person*

> **aclattr**

class errbot.backends.xmpp.**XMPPRoom**(*room_jid*, *bot*)
> Bases: *errbot.backends.xmpp.XMPPIdentifier*, *errbot.backends.base.Room*

> **__init__**(*room_jid*, *bot*)
> > Initialize self. See help(type(self)) for accurate signature.

> **configure**()
> > Configure the room.
> >
> > Currently this simply sets the default room configuration as received by the server. May be extended in the future to set a custom room configuration instead.

> **create**()
> > Not supported on this back-end (SleekXMPP doesn't support it). Will join the room to ensure it exists, instead.

> **destroy**()
> > Destroy the room.
> >
> > Calling this on a non-existing room is a no-op.

> **exists**
> > Boolean indicating whether this room already exists or not.
> >
> > > **Getter** Returns *True* if the room exists, *False* otherwise.

> **invite**(**args*)
> > Invite one or more people into the room.
> >
> > > ***argsargs** One or more JID's to invite into the room.

> **join**(*username=None*, *password=None*)
> > Join the room.
> >
> > If the room does not exist yet, this will automatically call *create()* on it first.

> **joined**
> > Boolean indicating whether this room has already been joined.
> >
> > > **Getter** Returns *True* if the room has been joined, *False* otherwise.

> **leave**(*reason=None*)
> > Leave the room.
> >
> > > **Parameters** **reason** – An optional string explaining the reason for leaving the room

> **occupants**
> > The room's occupants.
> >
> > > **Getter** Returns a list of MUCOccupant instances.
> > >
> > > **Raises** MUCNotJoinedError if the room has not yet been joined.

> **topic**
> > The room topic.
> >
> > > **Getter** Returns the topic (a string) if one is set, *None* if no topic has been set at all.
> > >
> > > **Raises** RoomNotJoinedError if the room has not yet been joined.

class errbot.backends.xmpp.**XMPPRoomOccupant** (*node*, *domain*, *resource*, *room*)
    Bases: *errbot.backends.xmpp.XMPPPerson*, *errbot.backends.base.RoomOccupant*

    __init__ (*node*, *domain*, *resource*, *room*)
        Initialize self. See help(type(self)) for accurate signature.

    nick

    person

        Returns a backend specific unique identifier representing the person you are talking to.

    real_jid
        The JID of the room occupant, they used to login. Will only work if the errbot is moderator in the MUC or it is not anonymous.

    room
        Some backends have the full name of a user.

        Returns the fullname of this user if available.

errbot.backends.xmpp.**split_identifier** (*txtrep*)

Module contents

errbot.core_plugins package

Submodules

errbot.core_plugins.acls module

class errbot.core_plugins.acls.**ACLS** (*bot*, *name=None*)
    Bases: *errbot.botplugin.BotPlugin*

    This plugin implements access controls for commands, allowing them to be restricted via various rules.

    access_denied (*msg*, *reason*, *dry_run*)

    acls (*msg*, *cmd*, *args*, *dry_run*)
        Check command against ACL rules as defined in the bot configuration.

        Parameters

            • **msg** – The original chat message.

            • **cmd** – The command name itself.

            • **args** – Arguments passed to the command.

            • **dry_run** – True when this is a dry-run.

errbot.core_plugins.acls.**ciglob** (*text*, *patterns*)
    Case-insensitive version of glob.

    Match text against the list of patterns according to unix glob rules. Return True if a match is found, False otherwise.

errbot.core_plugins.acls.**get_acl_usr** (*msg*)
    Return the ACL attribute of the sender of the given message

`errbot.core_plugins.acls.`**`glob`**(*text*, *patterns*)

  Match text against the list of patterns according to unix glob rules. Return True if a match is found, False otherwise.

errbot.core_plugins.backup module

class `errbot.core_plugins.backup.`**`Backup`**(*bot*, *name=None*)

  Bases: *errbot.botplugin.BotPlugin*

  Backup related commands.

  `backup`(*msg*, *args*)

    Backup everything. Makes a backup script called backup.py in the data bot directory. You can restore the backup from the command line with errbot –restore

errbot.core_plugins.chatRoom module

class `errbot.core_plugins.chatRoom.`**`ChatRoom`**(*bot*, *name=None*)

  Bases: *errbot.botplugin.BotPlugin*

  `callback_connect`()

    Triggered when the bot has successfully connected to the chat network.

    Override this method to get notified when the bot is connected.

  `callback_message`(*msg*)

    Triggered on every message not coming from the bot itself.

    Override this method to get notified on *ANY* message.

       Parameters **message** – representing the message that was received.

  `connected = False`

  `deactivate`()

    Triggered on plugin deactivation.

    Override this method if you want to do something at tear-down phase (don't forget to *super().deactivate()*).

  `room_create`(*message*, *args*)

    Create a chatroom.

    Usage: !room create <room>

    Examples (XMPP): !room create example-room@chat.server.tld

    Examples (IRC): !room create #example-room

  `room_destroy`(*message*, *args*)

    Destroy a chatroom.

    Usage: !room destroy <room>

    Examples (XMPP): !room destroy example-room@chat.server.tld

    Examples (IRC): !room destroy #example-room

  `room_invite`(*message*, *args*)

    Invite one or more people into a chatroom.

    Usage: !room invite <room> <identifier1> [<identifier2>, ..]

Examples (XMPP): !room invite room@conference.server.tld bob@server.tld

Examples (IRC): !room invite #example-room bob

room_join(*message*, *args*)

Join (creating it first if needed) a chatroom.

Usage: !room join <room> [<password>]

Examples (XMPP): !room join example-room@chat.server.tld !room join example-room@chat.server.tld super-secret-password

Examples (IRC): !room join #example-room !room join #example-room super-secret-password !room join #example-room "password with spaces"

room_leave(*message*, *args*)

Leave a chatroom.

Usage: !room leave <room>

Examples (XMPP): !room leave example-room@chat.server.tld

Examples (IRC): !room leave #example-room

room_list(*message*, *args*)

List chatrooms the bot has joined.

Usage: !room list

Examples: !room list

room_occupants(*message*, *args*)

List the occupants in a given chatroom.

Usage: !room occupants <room 1> [<room 2> ..]

Examples (XMPP): !room occupants room@conference.server.tld

Examples (IRC): !room occupants #example-room #another-example-room

room_topic(*message*, *args*)

Get or set the topic for a room.

Usage: !room topic <room> [<new topic>]

Examples (XMPP): !room topic example-room@chat.server.tld !room topic example-room@chat.server.tld "Err rocks!"

Examples (IRC): !room topic #example-room !room topic #example-room "Err rocks!"

errbot.core_plugins.cnf_filter module

class errbot.core_plugins.cnf_filter.**CommandNotFoundFilter**(*bot*, *name=None*)

Bases: *errbot.botplugin.BotPlugin*

cnf_filter(*msg*, *cmd*, *args*, *dry_run*, *emptycmd=False*)

Check if command exists. If not, signal plugins. This plugin will be called twice: once as a command filter and then again as a "command not found" filter. See the emptycmd parameter.

Parameters

- **msg** – Original chat message.

- **cmd** – Parsed command.

- **args** – Command arguments.

- **dry_run** – True when this is a dry-run.

- **emptycmd** (`bool`) – False when this command has been parsed and is valid.

True if the command was not found.

errbot.core_plugins.flows module

class errbot.core_plugins.flows.**Flows**(*bot*, *name=None*)

Bases: *errbot.botplugin.BotPlugin*

Management commands related to flows / conversations.

check_user(*msg*, *flow*)

Checks to make sure that either the user started the flow, or is a bot admin

flows_kill(*_*, *user*, *flow_name*)

usage: flows_kill [-h] user flow_name

Admin command to kill a specific flow.

positional arguments: user flow_name

optional arguments:

      -h, --help   show this help message and exit

flows_list(*msg*, *args*)

Displays the list of setup flows.

flows_show(*_*, *args*)

Shows the structure of a flow.

flows_start(*msg*, *args*)

Manually start a flow within the context of the calling user. You can prefeed the flow data with a json payload. Example:

    !flows start poll_setup {"title":"yeah!","options":["foo","bar","baz"]}

flows_status(*msg*, *args*)

Displays the list of started flows.

flows_stop(*msg*, *args*)

Stop flows you are in. optionally, stop a specific flow you are in.

recurse_node(*response: _io.StringIO*, *stack*, *f: errbot.flow.FlowNode*, *flow: errbot.flow.Flow = None*)

errbot.core_plugins.health module

class errbot.core_plugins.health.**Health**(*bot*, *name=None*)

Bases: *errbot.botplugin.BotPlugin*

restart(*msg*, *args*)

Restart the bot.

shutdown(*msg*, *confirmed*, *kill*)

usage: shutdown [-h] [--kill] [--confirm]

Shutdown the bot. Useful when the things are going crazy and you don't have access to the machine.

> **optional arguments:**
>
> > **-h, --help**   show this help message and exit
> >
> > **--kill**   kill the bot instantly, don't shut down gracefully
> >
> > **--confirm**   confirm you want to shut down

status(*msg*, *args*)
    If I am alive I should be able to respond to this one

status_gc(*_*, *args*)
    shows the garbage collection details

status_load(*_*, *args*)
    shows the load status

status_plugins(*_*, *args*)
    shows the plugin status

uptime(*_*, *args*)
    Return the uptime of the bot

errbot.core_plugins.help module

class errbot.core_plugins.help.**Help**(*bot*, *name=None*)
    Bases: *errbot.botplugin.BotPlugin*

    MSG_HELP_TAIL = 'Type help <command name> to get more info about that specific command

    MSG_HELP_UNDEFINED_COMMAND = 'That command is not defined.'

    about(*msg*, *args*)
        Return information about this Errbot instance and version

    apropos(*msg*, *args*)
        Returns a help string listing available options.

        Automatically assigned to the "help" command.

    help(*msg*, *args*)
        Returns a help string listing available options. Automatically assigned to the "help" command.

    is_git_directory(*path='.'*)

errbot.core_plugins.plugins module

class errbot.core_plugins.plugins.**Plugins**(*bot*, *name=None*)
    Bases: *errbot.botplugin.BotPlugin*

    formatted_plugin_list(*active_only=True*)
        Return a formatted, plain-text list of loaded plugins.

        When active_only=True, this will only return plugins which are actually active. Otherwise, it will also
        include inactive (blacklisted) plugins.

    plugin_activate(*_*, *args*)
        activate a plugin. [calls .activate() on the plugin]

plugin_blacklist(_, *args*)
> Blacklist a plugin so that it will not be loaded automatically during bot startup. If the plugin is currently activated, it will deactiveate it first.

plugin_config(_, *args*)
> configure or get the configuration / configuration template for a specific plugin ie. !plugin config Example-Bot could return a template if it is not configured: {'LOGIN': 'example@example.com', 'PASSWORD': 'password', 'DIRECTORY': '/toto'} Copy paste, adapt so can configure the plugin : !plugin config ExampleBot {'LOGIN': 'my@email.com', 'PASSWORD': 'myrealpassword', 'DIRECTORY': '/tmp'} It will then reload the plugin with this config. You can at any moment retrieve the current values: !plugin config ExampleBot should return : {'LOGIN': 'my@email.com', 'PASSWORD': 'myrealpassword', 'DIRECTORY': '/tmp'}

plugin_deactivate(_, *args*)
> deactivate a plugin. [calls .deactivate on the plugin]

plugin_info(_, *args*)
> Gives you a more technical information about a specific plugin.

plugin_reload(_, *args*)
> reload a plugin: reload the code of the plugin leaving the activation status intact.

plugin_unblacklist(_, *args*)
> Remove a plugin from the blacklist

repos(_, *args*)
> list the current active plugin repositories

repos_install(_, *args*)
> install a plugin repository from the given source or a known public repo (see !repos to find those). for example from a known repo : !install err-codebot for example a git url : git@github.com:gbin/plugin.git or an url towards a tar.gz archive : http://www.gootz.net/plugin-latest.tar.gz

repos_search(_, *args*)
> Searches the repo index. for example: !repos search jenkins

repos_uninstall(_, *repo_name*)
> uninstall a plugin repository by name.

repos_update(_, *args*)
> update the bot and/or plugins use : !repos update all to update everything or : !repos update repo_name repo_name … to update selectively some repos

errbot.core_plugins.textcmds module

class errbot.core_plugins.textcmds.**TextModeCmds**(*bot*, *name=None*)
> Bases: *errbot.botplugin.BotPlugin*

> Internal to TextBackend.

activate()
> Triggered on plugin activation.

> Override this method if you want to do something at initialization phase (don't forget to *super().activate()*).

asadmin(*msg*, _)
> This puts you in a 1-1 chat with the bot.

asuser(*msg*, *args*)
> This puts you in a room with the bot. You can specify a name otherwise it will default to 'luser'.

deactivate()
>> Triggered on plugin deactivation.

>> Override this method if you want to do something at tear-down phase (don't forget to *super().deactivate()*).

inperson(*msg*, *_*)
>> This puts you in a 1-1 chat with the bot.

inroom(*msg*, *args*)
>> This puts you in a room with the bot.

ml(*msg*, *_*)
>> Switch back and forth between normal mode and multiline mode. Use this if you want to test commands spanning multiple lines. Note: in multiline, press enter twice to end and send the message.

errbot.core_plugins.utils module

class errbot.core_plugins.utils.**Utils**(*bot*, *name=None*)
>> Bases: *errbot.botplugin.BotPlugin*

>> **echo**(*_*, *args*)
>>> A simple echo command. Useful for encoding tests etc . . .

>> **history**(*msg*, *args*)
>>> display the command history

>> **log_tail**(*msg*, *args*)
>>> Display a tail of the log of n lines or 40 by default use : !log tail 10

>> **render_test**(*_*, *args*)
>>> Tests / showcases the markdown rendering on your current backend

>> **whoami**(*msg*, *args*)
>>> A simple command echoing the details of your identifier. Useful to debug identity problems.

errbot.core_plugins.utils.**tail**(*f*, *window=20*)

errbot.core_plugins.vcheck module

class errbot.core_plugins.vcheck.**VersionChecker**(*bot*, *name=None*)
>> Bases: *errbot.botplugin.BotPlugin*

>> **activate**()
>>> Triggered on plugin activation.

>>> Override this method if you want to do something at initialization phase (don't forget to *super().activate()*).

>> **activated = False**

>> **callback_connect**()
>>> Triggered when the bot has successfully connected to the chat network.

>>> Override this method to get notified when the bot is connected.

>> **connected = False**

>> **deactivate**()
>>> Triggered on plugin deactivation.

>>> Override this method if you want to do something at tear-down phase (don't forget to *super().deactivate()*).

version_check()

errbot.core_plugins.webserver module

class errbot.core_plugins.webserver.**Webserver**(**args*, ***kwargs*)
    Bases: *errbot.botplugin.BotPlugin*

    __init__(**args*, ***kwargs*)
        Initialize self. See help(type(self)) for accurate signature.

    activate()
        Triggered on plugin activation.

        Override this method if you want to do something at initialization phase (don't forget to *super().activate()*).

    check_configuration(*configuration*)
        By default, this method will do only a BASIC check. You need to override it if you want to do more complex checks. It will be called before the configure callback. Note if the config_template is None, it will never be called.

        It means recusively:

        1. in case of a dictionary, it will check if all the entries and from the same type are there and not more.

        2. in case of an array or tuple, it will assume array members of the same type of first element of the template (no mix typed is supported)

        In case of validation error it should raise a errbot.ValidationException

            Parameters configuration – the configuration to be checked.

    deactivate()
        Triggered on plugin deactivation.

        Override this method if you want to do something at tear-down phase (don't forget to *super().deactivate()*).

    echo(*incoming_request*)
        A simple test webhook

    generate_certificate(*_*, *args*)
        Generate a self-signed SSL certificate for the Webserver

    get_configuration_template()
        If your plugin needs a configuration, override this method and return a configuration template.

        For example a dictionary like: return {'LOGIN' : 'example@example.com', 'PASSWORD' : 'password'}

        Note: if this method returns None, the plugin won't be configured

    run_server()

    webhook_test(*_*, *args*)

        Test your webhooks from within err.

        The syntax is : !webhook test [relative_url] [post content]

        It triggers the notification and generate also a little test report.

    webstatus(*msg*, *args*)
        Gives a quick status of what is mapped in the internal webserver

errbot.core_plugins.webserver.**make_ssl_certificate**(*key_path*, *cert_path*)
> Generate a self-signed certificate

> The generated key will be written out to key_path, with the corresponding certificate itself being written to cert_path. :param cert_path: path where to write the certificate. :param key_path: path where to write the key.

errbot.core_plugins.wsview module

class errbot.core_plugins.wsview.**WebView**(*func*, *form_param*, *raw*)
> Bases: flask.views.View

> **__init__**(*func*, *form_param*, *raw*)
> > Initialize self. See help(type(self)) for accurate signature.

> **dispatch_request**(**args*, ***kwargs*)
> > Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

errbot.core_plugins.wsview.**reset_app**()
> Zap everything here, useful for unit tests

errbot.core_plugins.wsview.**route**(*obj*)
> Check for functions to route in obj and route them.

errbot.core_plugins.wsview.**strip_path**()

errbot.core_plugins.wsview.**try_decode_json**(*req*)

Module contents

errbot.rendering package

Submodules

errbot.rendering.ansiext module

class errbot.rendering.ansiext.**AnsiExtension**(**args*, ***kwargs*)
> Bases: markdown.extensions.Extension

> (kinda hackish) This is just a private extension to postprocess the html text to ansi text

> **extendMarkdown**(*md*, *md_globals*)
> > Add the various proccesors and patterns to the Markdown Instance.

> > This method must be overriden by every extension.

> > Keyword arguments:

> > - md: The Markdown instance.

> > - md_globals: Global variables in the markdown module namespace.

class errbot.rendering.ansiext.**AnsiPostprocessor**(*markdown_instance=None*)
> Bases: markdown.postprocessors.Postprocessor

> Markdown generates html entities, this reputs them back to their unicode equivalent

run(*text*)

    Subclasses of Postprocessor should implement a *run* method, which takes the html document as a single text string and returns a (possibly modified) string.

class errbot.rendering.ansiext.**AnsiPreprocessor**(*md*)

    Bases: markdown.extensions.fenced_code.FencedBlockPreprocessor

    run(*lines*)

        Match and store Fenced Code Blocks in the HtmlStash.

class errbot.rendering.ansiext.**BorderlessTable**(*chr_table*)

    Bases: object

    __init__(*chr_table*)

        Initialize self. See help(type(self)) for accurate signature.

    add_col()

    add_header()

    begin_headers()

    end_headers()

    next_row()

    write(*text*)

class errbot.rendering.ansiext.**CharacterTable**(*fg_black*, *fg_red*, *fg_green*, *fg_yellow*, *fg_blue*, *fg_magenta*, *fg_cyan*, *fg_white*, *fg_default*, *bg_black*, *bg_red*, *bg_green*, *bg_yellow*, *bg_blue*, *bg_magenta*, *bg_cyan*, *bg_white*, *bg_default*, *fx_reset*, *fx_bold*, *fx_italic*, *fx_underline*, *fx_not_italic*, *fx_not_underline*, *fx_normal*, *fixed_width*, *end_fixed_width*, *inline_code*, *end_inline_code*)

    Bases: tuple

    bg_black

        Alias for field number 9

    bg_blue

        Alias for field number 13

    bg_cyan

        Alias for field number 15

    bg_default

        Alias for field number 17

    bg_green

        Alias for field number 11

    bg_magenta

        Alias for field number 14

    bg_red

        Alias for field number 10

    bg_white

        Alias for field number 16

bg_yellow
: Alias for field number 12

end_fixed_width
: Alias for field number 26

end_inline_code
: Alias for field number 28

fg_black
: Alias for field number 0

fg_blue
: Alias for field number 4

fg_cyan
: Alias for field number 6

fg_default
: Alias for field number 8

fg_green
: Alias for field number 2

fg_magenta
: Alias for field number 5

fg_red
: Alias for field number 1

fg_white
: Alias for field number 7

fg_yellow
: Alias for field number 3

fixed_width
: Alias for field number 25

fx_bold
: Alias for field number 19

fx_italic
: Alias for field number 20

fx_normal
: Alias for field number 24

fx_not_italic
: Alias for field number 22

fx_not_underline
: Alias for field number 23

fx_reset
: Alias for field number 18

fx_underline
: Alias for field number 21

inline_code
: Alias for field number 27

class errbot.rendering.ansiext.**NSC**(*s*)

    Bases: object

    __init__(*s*)

        Initialize self. See help(type(self)) for accurate signature.

class errbot.rendering.ansiext.**Table**(*chr_table*)

    Bases: object

    __init__(*chr_table*)

        Initialize self. See help(type(self)) for accurate signature.

    add_col()

    add_header()

    begin_headers()

    end_headers()

    next_row()

    write(*text*)

errbot.rendering.ansiext.**enable_format**(*name*, *chr_table*, *borders=True*)

errbot.rendering.ansiext.**recurse**(*write*, *chr_table*, *element*, *table=None*, *borders=True*)

errbot.rendering.ansiext.**translate**(*element*, *chr_table=CharacterTable(fg_black=<ansi.colour.base.Graphic object>, fg_red=<ansi.colour.base.Graphic object>, fg_green=<ansi.colour.base.Graphic object>, fg_yellow=<ansi.colour.base.Graphic object>, fg_blue=<ansi.colour.base.Graphic object>, fg_magenta=<ansi.colour.base.Graphic object>, fg_cyan=<ansi.colour.base.Graphic object>, fg_white=<ansi.colour.base.Graphic object>, fg_default=<ansi.colour.base.Graphic object>, bg_black=<ansi.colour.base.Graphic object>, bg_red=<ansi.colour.base.Graphic object>, bg_green=<ansi.colour.base.Graphic object>, bg_yellow=<ansi.colour.base.Graphic object>, bg_blue=<ansi.colour.base.Graphic object>, bg_magenta=<ansi.colour.base.Graphic object>, bg_cyan=<ansi.colour.base.Graphic object>, bg_white=<ansi.colour.base.Graphic object>, bg_default=<ansi.colour.base.Graphic object>, fx_reset=<ansi.colour.base.Graphic object>, fx_bold=<ansi.colour.base.Graphic object>, fx_italic=<ansi.colour.base.Graphic object>, fx_underline=<ansi.colour.base.Graphic object>, fx_not_italic=<ansi.colour.base.Graphic object>, fx_not_underline=<ansi.colour.base.Graphic object>, fx_normal=<ansi.colour.base.Graphic object>, fixed_width='', end_fixed_width='', inline_code='', end_inline_code='')*, *borders=True*)

errbot.rendering.xhtmlim module

errbot.rendering.xhtmlim.**unescape**(*s*)

Module contents

class `errbot.rendering.`**Mde2mdConverter**
>   Bases: `object`
>
>   **convert**(*mde*)

`errbot.rendering.`**ansi**()
>   This makes a converter from markdown to ansi (console) format. It can be called like this: from errbot.rendering import ansi md_converter = ansi() # you need to cache the converter
>
>   ansi_txt = md_converter.convert(md_txt)

`errbot.rendering.`**imtext**()
>   This makes a converter from markdown to imtext (unicode) format. imtest is the format like gtalk, slack or skype with simple _ or * markup.
>
>   It can be called like this: from errbot.rendering import imtext md_converter = imtext() # you need to cache the converter
>
>   im_text = md_converter.convert(md_txt)

`errbot.rendering.`**md**()
>   This makes a converter from markdown-extra to markdown, stripping the attributes from extra.

`errbot.rendering.`**md_escape**(*txt*)
>   Call this if you want to be sure your text won't be interpreted as markdown :param txt: bare text to escape.

`errbot.rendering.`**text**()
>   This makes a converter from markdown to text (unicode) format. It can be called like this: from errbot.rendering import text md_converter = text() # you need to cache the converter
>
>   pure_text = md_converter.convert(md_txt)

`errbot.rendering.`**xhtml**()
>   This makes a converter from markdown to xhtml format. It can be called like this: from errbot.rendering import xhtml md_converter = xhtml() # you need to cache the converter
>
>   html = md_converter.convert(md_txt)

errbot.storage package

Submodules

errbot.storage.base module

class `errbot.storage.base.`**StorageBase**
>   Bases: `object`
>
>   Contract to implemement a storage.
>
>   **close**() → None
>   >   Sync and close the storage. The caller of close will protect against close on non open and double close.
>
>   **get**(*key: str*) → Any
>   >   Get the value stored for key. Raises KeyError if the key doesn't exist. The caller of get will protect against get on non open.
>   >
>   >   >   **Parameters key** (`str`) – the key
>   >   >
>   >   >   **Returns** the value

---

keys() → Iterable[str]

> **Returns** an iterator on all the entries

len() → int

> **Returns** the number of keys set.

remove(*key: str*) → None
> Remove key. Raises KeyError if the key doesn't exist. The caller of get will protect against get on non open.
>
> > **Parameters key** (str) – the key

set(*key: str*, *value: Any*) → None
> Atomically set the key to the given value. The caller of set will protect against set on non open.
>
> > **Parameters**
> >
> > - **key** (str) – string as key
> >
> > - **value** – pickalable python object

class errbot.storage.base.**StoragePluginBase**(*bot_config*)
> Bases: object

Base to implement a storage plugin. This is a factory for the namespaces.

__init__(*bot_config*)
> Initialize self. See help(type(self)) for accurate signature.

open(*namespace: str*) → errbot.storage.base.StorageBase
> Open the storage with the given namespace (core, or plugin name) and config. The caller of open will protect against double opens.
>
> > **Return type** *StorageBase*
> >
> > **Parameters namespace** (str) – a namespace to isolate the plugin storages.
> >
> > **Returns**

errbot.storage.memory module

class errbot.storage.memory.**MemoryStorage**(*namespace*)
> Bases: *errbot.storage.base.StorageBase*

__init__(*namespace*)
> Initialize self. See help(type(self)) for accurate signature.

close() → None
> Sync and close the storage. The caller of close will protect against close on non open and double close.

get(*key: str*) → Any
> Get the value stored for key. Raises KeyError if the key doesn't exist. The caller of get will protect against get on non open.
>
> > **Parameters key** (str) – the key
> >
> > **Returns** the value

keys()

> **Returns** an iterator on all the entries

len()

> **Returns** the number of keys set.

remove (*key: str*)
> Remove key. Raises KeyError if the key doesn't exist. The caller of get will protect against get on non open.
>
> > **Parameters** **key** (`str`) – the key

set (*key: str*, *value: Any*) → None
> Atomically set the key to the given value. The caller of set will protect against set on non open.
>
> > **Parameters**
> >
> > - **key** (`str`) – string as key
> >
> > - **value** – pickalable python object

class errbot.storage.memory.**MemoryStoragePlugin** (*bot_config*)
> Bases: *errbot.storage.base.StoragePluginBase*

open (*namespace: str*) → errbot.storage.base.StorageBase
> Open the storage with the given namespace (core, or plugin name) and config. The caller of open will protect against double opens.
>
> > **Return type** *StorageBase*
>
> > **Parameters** **namespace** (`str`) – a namespace to isolate the plugin storages.
>
> > **Returns**

errbot.storage.shelf module

class errbot.storage.shelf.**ShelfStorage** (*path*)
> Bases: *errbot.storage.base.StorageBase*

__init__ (*path*)
> Initialize self. See help(type(self)) for accurate signature.

close () → None
> Sync and close the storage. The caller of close will protect against close on non open and double close.

get (*key: str*) → Any
> Get the value stored for key. Raises KeyError if the key doesn't exist. The caller of get will protect against get on non open.
>
> > **Parameters** **key** (`str`) – the key
>
> > **Returns** the value

keys ()
> > **Returns** an iterator on all the entries

len ()
> > **Returns** the number of keys set.

remove (*key: str*)
> Remove key. Raises KeyError if the key doesn't exist. The caller of get will protect against get on non open.
>
> > **Parameters** **key** (`str`) – the key

set (*key: str*, *value: Any*) → None
> Atomically set the key to the given value. The caller of set will protect against set on non open.

> > **Parameters**

> > > - **key** (`str`) – string as key

> > > - **value** – pickalable python object

class errbot.storage.shelf.**ShelfStoragePlugin**(*bot_config*)
> Bases: *errbot.storage.base.StoragePluginBase*

> **__init__**(*bot_config*)
> > Initialize self. See help(type(self)) for accurate signature.

> **open**(*namespace: str*) → errbot.storage.base.StorageBase
> > Open the storage with the given namespace (core, or plugin name) and config. The caller of open will protect against double opens.

> > **Return type** *StorageBase*

> > **Parameters namespace** (`str`) – a namespace to isolate the plugin storages.

> > **Returns**

Module contents

exception errbot.storage.**StoreAlreadyOpenError**
> Bases: *errbot.storage.StoreException*

exception errbot.storage.**StoreException**
> Bases: Exception

class errbot.storage.**StoreMixin**
> Bases: collections.abc.MutableMapping

> This class handle the basic needs of bot plugins and core like loading, unloading and creating a storage

> **__init__**()
> > Initialize self. See help(type(self)) for accurate signature.

> **close_storage**()

> **keys**() → a set-like object providing a view on D's keys

> **mutable**(*key*, *default=None*)

> **open_storage**(*storage_plugin*, *namespace*)

exception errbot.storage.**StoreNotOpenError**
> Bases: *errbot.storage.StoreException*

8.1.2 Submodules

errbot.backend_plugin_manager module

class errbot.backend_plugin_manager.**BackendPluginManager**(*bot_config, base_module: str, plugin_name: str, base_class: Type[CT_co], base_search_dir, extra_search_dirs=()*)

    Bases: `object`

    This is a one shot plugin manager for Backends and Storage plugins.

    __init__(*bot_config, base_module: str, plugin_name: str, base_class: Type[CT_co], base_search_dir, extra_search_dirs=()*)
        Initialize self. See help(type(self)) for accurate signature.

    load_plugin() → Any

exception errbot.backend_plugin_manager.**PluginNotFoundException**
    Bases: `Exception`

errbot.backend_plugin_manager.**enumerate_backend_plugins**(*all_plugins_paths*)

errbot.bootstrap module

errbot.bootstrap.**bootstrap**(*bot_class*, *logger*, *config*, *restore=None*)
    Main starting point of Errbot.

        Parameters

            • **bot_class** – The backend class inheriting from Errbot you want to start.

            • **logger** – The logger you want to use.

            • **config** – The config.py module.

            • **restore** – Start Errbot in restore mode (from a backup).

errbot.bootstrap.**bot_config_defaults**(*config*)

errbot.bootstrap.**get_storage_plugin**(*config*)
    Find and load the storage plugin :param config: the bot configuration. :return: the storage plugin

errbot.bootstrap.**restore_bot_from_backup**(*backup_filename*, *, *bot*, *log*)
    Restores the given bot by executing the 'backup' script.

    The backup file is a python script which manually execute a series of commands on the bot to restore it to its previous state.

        Parameters

            • **backup_filename** – the full path to the backup script.

            • **bot** – the bot instance to restore

            • **log** – logger to use during the restoration process

errbot.bootstrap.**setup_bot**(*backend_name: str*, *logger*, *config*, *restore=None*) → errbot.core.ErrBot

errbot.botplugin module

class errbot.botplugin.**ArgParserBase**
Bases: object

The *ArgSplitterBase* class defines the API which is used for argument splitting (used by the *split_args_with* parameter on botcmd()).

parse_args(*args: str*)
This method takes a string of un-split arguments and parses it, returning a list that is the result of splitting.

If splitting fails for any reason it should return an exception of some kind.

> **Parameters args** (str) – string to parse

class errbot.botplugin.**BotPlugin**(*bot*, *name=None*)
Bases: *errbot.botplugin.BotPluginBase*

activate() → None
Triggered on plugin activation.

Override this method if you want to do something at initialization phase (don't forget to *super().activate()*).

build_identifier(*txtrep: str*) → errbot.backends.base.Identifier
Transform a textual representation of a user identifier to the correct Identifier object you can set in Message.to and Message.frm.

> **Return type** *Identifier*

> **Parameters txtrep** (str) – the textual representation of the identifier (it is backend dependent).

> **Returns** a user identifier.

callback_botmessage(*message: errbot.backends.base.Message*)
Triggered on every message coming from the bot itself.

Override this method to get notified on all messages coming from the bot itself (including those from other plugins).

> **Parameters message** (*Message*) – An instance of *Message* representing the message that was received.

callback_connect() → None
Triggered when the bot has successfully connected to the chat network.

Override this method to get notified when the bot is connected.

callback_mention(*message: errbot.backends.base.Message, mentioned_people: Sequence[errbot.backends.base.Identifier]*) → None
Triggered if there are mentioned people in message.

Override this method to get notified when someone was mentioned in message. [Note: This might not be implemented by all backends.]

> **Parameters**
>
> • **message** (*Message*) – representing the message that was received.
>
> • **mentioned_people** – all mentioned people in this message.

callback_message(*message: errbot.backends.base.Message*) → None
Triggered on every message not coming from the bot itself.

Override this method to get notified on *ANY* message.

> > > **Parameters** **message** (*Message*) – representing the message that was received.

callback_presence (*presence: errbot.backends.base.Presence*) → None
 Triggered on every presence change.

> > **Parameters** **presence** (*Presence*) – An instance of *Presence* representing the new presence state that was received.

callback_room_joined (*room: errbot.backends.base.Room*)
 Triggered when the bot has joined a MUC.

> > **Parameters** **room** (*Room*) – An instance of MUCRoom representing the room that was joined.

callback_room_left (*room: errbot.backends.base.Room*)
 Triggered when the bot has left a MUC.

> > **Parameters** **room** (*Room*) – An instance of MUCRoom representing the room that was left.

callback_room_topic (*room: errbot.backends.base.Room*)
 Triggered when the topic in a MUC changes.

> > **Parameters** **room** (*Room*) – An instance of MUCRoom representing the room for which the topic changed.

callback_stream (*stream: errbot.backends.base.Stream*) → None
 Triggered asynchronously (in a different thread context) on every incoming stream request or file transfert requests. You can block this call until you are done with the stream. To signal that you accept / reject the file, simply call stream.accept() or stream.reject() and return.

> > **Parameters** **stream** (*Stream*) – the incoming stream request.

change_presence (*status: str = 'online'*, *message: str = ''*) → None

> Changes the presence/status of the bot.

> **Parameters**

> > - **status** (str) – One of the constant defined in base.py : ONLINE, OFFLINE, DND,. . .
> > - **message** (str) – Additional message

> **Returns** None

check_configuration (*configuration: Mapping[KT, VT_co]*) → None
 By default, this method will do only a BASIC check. You need to override it if you want to do more complex checks. It will be called before the configure callback. Note if the config_template is None, it will never be called.

> It means recusively:

>   1. in case of a dictionary, it will check if all the entries and from the same type are there and not more.

>   2. in case of an array or tuple, it will assume array members of the same type of first element of the template (no mix typed is supported)

> In case of validation error it should raise a errbot.ValidationException

> **Parameters** **configuration** – the configuration to be checked.

configure (*configuration: Mapping[KT, VT_co]*) → None
 By default, it will just store the current configuration in the self.config field of your plugin. If this plugin has no configuration yet, the framework will call this function anyway with None.

 This method will be called before activation so don't expect to be activated at that point.

> **Parameters** `configuration` – injected configuration for the plugin.

`deactivate()` → None

> Triggered on plugin deactivation.
>
> Override this method if you want to do something at tear-down phase (don't forget to *super().deactivate()*).

`get_configuration_template()` → Mapping[KT, VT_co]

> If your plugin needs a configuration, override this method and return a configuration template.
>
> For example a dictionary like: return {'LOGIN' : 'example@example.com', 'PASSWORD' : 'password'}
>
> Note: if this method returns None, the plugin won't be configured

`query_room`(*room: str*) → errbot.backends.base.Room

> Query a room for information.
>
> > **Return type** *Room*
> >
> > **Parameters** `room` (`str`) – The JID/identifier of the room to query for.
> >
> > **Returns** An instance of `MUCRoom`.
> >
> > **Raises** *RoomDoesNotExistError* if the room doesn't exist.

`rooms()` → Sequence[errbot.backends.base.Room]

> The list of rooms the bot is currently in.

`send`(*identifier: errbot.backends.base.Identifier*, *text: str*, *in_reply_to: errbot.backends.base.Message = None*, *groupchat_nick_reply: bool = False*) → None

> Send a message to a room or a user.
>
> **Parameters**
>
> - `groupchat_nick_reply` (`bool`) – if True the message will mention the user in the chatroom.
>
> - `in_reply_to` (*Message*) – the original message this message is a reply to (optional). In some backends it will start a thread.
>
> - `text` (`str`) – markdown formatted text to send to the user.
>
> - `identifier` (*Identifier*) – An Identifier representing the user or room to message. Identifiers may be created with *[build_identifier()](build_identifier)*.

`send_card`(*body: str = ''*, *to: errbot.backends.base.Identifier = None*, *in_reply_to: errbot.backends.base.Message = None*, *summary: str = None*, *title: str = ''*, *link: str = None*, *image: str = None*, *thumbnail: str = None*, *color: str = 'green'*, *fields: Tuple[Tuple[str, str], ...] = ()*) → None

> Sends a card.
>
> A Card is a special type of preformatted message. If it matches with a backend similar concept like on Slack or Hipchat it will be rendered natively, otherwise it will be sent as a regular formatted message.
>
> **Parameters**
>
> - `body` (`str`) – main text of the card in markdown.
>
> - `to` (*Identifier*) – the card is sent to this identifier (Room, RoomOccupant, Person. . .).
>
> - `in_reply_to` (*Message*) – the original message this message is a reply to (optional).
>
> - `summary` (`str`) – (optional) One liner summary of the card, possibly collapsed to it.
>
> - `title` (`str`) – (optional) Title possibly linking.
>
> - `link` (`str`) – (optional) url the title link is pointing to.

- **image** (str) – (optional) link to the main image of the card.

- **thumbnail** (str) – (optional) link to an icon / thumbnail.

- **color** (str) – (optional) background color or color indicator.

- **fields** – (optional) a tuple of (key, value) pairs.

send_stream_request(*user: errbot.backends.base.Identifier, fsource: io.IOBase, name: str = None, size: int = None, stream_type: str = None*)
  Sends asynchronously a stream/file to a user.

  Parameters

  - **user** ([*Identifier*](#)) – is the identifier of the person you want to send it to.

  - **fsource** (IOBase) – is a file object you want to send.

  - **name** (str) – is an optional filename for it.

  - **size** (int) – is optional and is the espected size for it.

  - **stream_type** (str) – is optional for the mime_type of the content.

  It will return a Stream object on which you can monitor the progress of it.

send_templated(*identifier: errbot.backends.base.Identifier, template_name: str, template_parameters: Mapping[KT, VT_co], in_reply_to: errbot.backends.base.Message = None, groupchat_nick_reply: bool = False*) → None
  Sends asynchronously a message to a room or a user.

  Same as send but passing a template name and parameters instead of directly the markdown text. :type groupchat_nick_reply: bool :type in_reply_to: [*Message*](#) :type template_name: str :type identifier: [*Identifier*](#) :param template_parameters: arguments for the template. :param template_name: name of the template to use. :param groupchat_nick_reply: if True it will mention the user in the chatroom. :param in_reply_to: optionally, the original message this message is the answer to. :param identifier: identifier of the user or room to which you want to send a message to.

start_poller(*interval: float, method: Callable[[...], None], times: int = None, args: Tuple = None, kwargs: Mapping[KT, VT_co] = None*)
  Start to poll a method at specific interval in seconds.

  Note: it will call the method with the initial interval delay for the first time

  Also, you can program for example : self.program_poller(self, 30, fetch_stuff) where you have def fetch_stuff(self) in your plugin

  Parameters

  - **interval** (float) – interval in seconds

  - **method** – targetted method

  - **times** (int) – number of times polling should happen (defaults to``None`` which causes the polling to happen indefinitely)

  - **args** – args for the targetted method

  - **kwargs** – kwargs for the targetting method

stop_poller(*method: Callable[[...], None], args: Tuple = None, kwargs: Mapping[KT, VT_co] = None*)
  stop poller(s).

  If the method equals None -> it stops all the pollers you need to regive the same parameters as the original start_poller to match a specific poller to stop

---

> Parameters
>> • **kwargs** – The initial kwargs you gave to start_poller.
>>
>> • **args** – The initial args you gave to start_poller.
>>
>> • **method** – The initial method you passed to start_poller.

warn_admins(*warning: str*) → None

> Send a warning to the administrators of the bot.
>
>> Parameters **warning** (str) – The markdown-formatted text of the message to send.

class errbot.botplugin.**BotPluginBase**(*bot*, *name=None*)

> Bases: *errbot.storage.StoreMixin*
>
> This class handle the basic needs of bot plugins like loading, unloading and creating a storage It is the main contract between the plugins and the bot
>
> **__init__**(*bot*, *name=None*)
>
>> Initialize self. See help(type(self)) for accurate signature.
>
> **activate**() → None
>
>> Override if you want to do something at initialization phase (don't forget to super(Gnagna, self).activate())
>
> **bot_config**
>
>> Get the bot configuration from config.py. For example you can access: self.bot_config.BOT_DATA_DIR
>
> **bot_identifier**
>
>> Get bot identifier on current active backend.
>>
>> :return Identifier
>
> **create_dynamic_plugin**(*name: str, commands: Tuple[errbot.botplugin.Command], doc: str = ''*)
>
>> Creates a plugin dynamically and exposes its commands right away.
>>
>>> Parameters
>>>> • **name** (str) – name of the plugin.
>>>>
>>>> • **commands** – a tuple of command definition.
>>>>
>>>> • **doc** (str) – the main documentation of the plugin.
>
> **deactivate**() → None
>
>> Override if you want to do something at tear down phase (don't forget to super(Gnagna, self).deactivate())
>
> **destroy_dynamic_plugin**(*name: str*)
>
>> Reverse operation of create_dynamic_plugin.
>>
>> This allows you to dynamically refresh the list of commands for example. :type name: str :param name: the name of the dynamic plugin given to create_dynamic_plugin.
>
> **get_plugin**(*name*) → errbot.botplugin.BotPlugin
>
>> Gets a plugin your plugin depends on. The name of the dependency needs to be listed in [Code] section key DependsOn of your plug file. This method can only be used after your plugin activation (or having called super().activate() from activate itself). It will return a plugin object.
>>
>>> Parameters **name** – the name
>>>
>>> Returns the BotPlugin object requested.
>
> **init_storage**() → None
>
> **mode**
>
>> Get the current active backend.

> > **Returns** the mode like 'tox', 'xmpp' etc. . .

> **name**
> > Get the name of this plugin as described in its .plug file.

> > **Returns** The plugin name.

> **poller**(*interval: float, method: Callable[[...], None], times: int = None, args: Tuple = None, kwargs: Mapping[KT, VT_co] = None*)

> **program_next_poll**(*interval: float, method: Callable[[...], None], times: int = None, args: Tuple = None, kwargs: Mapping[KT, VT_co] = None*)

> **start_poller**(*interval: float, method: Callable[[...], None], times: int = None, args: Tuple = None, kwargs: Mapping[KT, VT_co] = None*)
> > Starts a poller that will be called at a regular interval

> > **Parameters**

> > > * **interval** (`float`) – interval in seconds

> > > * **method** – targetted method

> > > * **times** (`int`) – number of times polling should happen (defaults to``None`` which causes the polling to happen indefinitely)

> > > * **args** – args for the targetted method

> > > * **kwargs** – kwargs for the targetting method

> **stop_poller**(*method: Callable[[...], None], args: Tuple = None, kwargs: Mapping[KT, VT_co] = None*)

class errbot.botplugin.**Command**(*function, cmd_type=None, cmd_args=None, cmd_kwargs=None, name=None, doc=None*)
> Bases: `object`

> This is a dynamic definition of an errbot command.

> **__init__**(*function, cmd_type=None, cmd_args=None, cmd_kwargs=None, name=None, doc=None*)
> > Create a Command definition.

> > **Parameters**

> > > * **function** – a function or a lambda with the correct signature for the type of command to inject for example *def mycmd(plugin, msg, args)* for a botcmd. Note: the first parameter will be the plugin itself (equivalent to self).

> > > * **cmd_type** – defaults to *botcmd* but can be any decorator function used for errbot commands.

> > > * **cmd_args** – the parameters of the decorator.

> > > * **cmd_kwargs** – the kwargs parameter of the decorator.

> > > * **name** – defaults to the name of the function you are passing if it is a first class function or needs to be set if you use a lambda.

> > > * **doc** – defaults to the doc of the given function if it is a first class function. It can be set for a lambda or overridden for a function with this.

exception errbot.botplugin.**CommandError**(*reason: str, template: str = None*)
> Bases: `Exception`

> Use this class to report an error condition from your commands, the command did not proceed for a known "business" reason.

__init__ (*reason: str*, *template: str = None*)

> Parameters

>> • **reason** (`str`) – the reason for the error in the command.

>> • **template** (`str`) – apply this specific template to report the error.

class errbot.botplugin.**SeparatorArgParser**(*separator: str = None*, *maxsplit: int = -1*)
    Bases: *errbot.botplugin.ArgParserBase*

This argument splitter splits args on a given separator, like `str.split()` does.

__init__ (*separator: str = None*, *maxsplit: int = -1*)

> Parameters

>> • **separator** (`str`) – The separator on which arguments should be split. If sep is None, any whitespace string is a separator and empty strings are removed from the result.

>> • **maxsplit** (`int`) – If given, do at most this many splits.

parse_args(*args: str*)
    This method takes a string of un-split arguments and parses it, returning a list that is the result of splitting.

    If splitting fails for any reason it should return an exception of some kind.

> Parameters **args** (`str`) – string to parse

class errbot.botplugin.**ShlexArgParser**
    Bases: *errbot.botplugin.ArgParserBase*

This argument splitter splits args using posix shell quoting rules, like `shlex.split()` does.

parse_args(*args*)
    This method takes a string of un-split arguments and parses it, returning a list that is the result of splitting.

    If splitting fails for any reason it should return an exception of some kind.

> Parameters **args** – string to parse

exception errbot.botplugin.**ValidationException**
    Bases: `Exception`

errbot.botplugin.**recurse_check_structure**(*sample*, *to_check*)

errbot.cli module

errbot.cli.**debug**(*sig*, *frame*)
    Interrupt running process, and provide a python prompt for interactive debugging.

errbot.cli.**get_config**(*config_path*)

errbot.cli.**main**()

errbot.config-template module

errbot.core module

class errbot.core.**ErrBot**(*bot_config*)
    Bases: *errbot.backends.base.Backend*, *errbot.storage.StoreMixin*

ErrBot is the layer taking care of commands management and dispatching.

MSG_ERROR_OCCURRED = 'Computer says nooo. See logs for details'

MSG_UNKNOWN_COMMAND = 'Unknown command: "%(command)s". '

__init__(*bot_config*)
　　Those arguments will be directly those put in BOT_IDENTITY

all_commands
　　Return both commands and re_commands together.

attach_plugin_manager(*plugin_manager*)

attach_repo_manager(*repo_manager*)

attach_storage_plugin(*storage_plugin*)

callback_mention(*msg*, *people*)

callback_message(*msg*)
　　Processes for commands and dispatches the message to all the plugins.

callback_presence(*pres*)
　　Implemented by errBot.

callback_room_joined(*room*)
　　Triggered when the bot has joined a MUC.

　　　　Parameters room – An instance of `MUCRoom` representing the room that was joined.

callback_room_left(*room*)
　　Triggered when the bot has left a MUC.

　　　　Parameters room – An instance of `MUCRoom` representing the room that was left.

callback_room_topic(*room*)
　　Triggered when the topic in a MUC changes.

　　　　Parameters room – An instance of `MUCRoom` representing the room for which the topic changed.

callback_stream(*stream*)

connect_callback()

disconnect_callback()

get_command_classes()

get_doc(*command*)
　　Get command documentation

static get_plugin_class_from_method(*meth*)

initialize_backend_storage()
　　Initialize storage for the backend to use.

inject_command_filters_from(*instance_to_inject*)

inject_commands_from(*instance_to_inject*)

inject_flows_from(*instance_to_inject*)

prefix_groupchat_reply(*message:　errbot.backends.base.Message*, *identifier:　errbot.backends.base.Identifier*)
　　Patches message with the conventional prefix to ping the specific contact For example: @gbin, you forgot the milk !

process_message(*msg*)

> Check if the given message is a command for the bot and act on it. It return True for triggering the callback_messages on the .callback_messages on the plugins.
>
> > **Parameters msg** – the incoming message.

static process_template(*template_name*, *template_parameters*)

remove_command_filters_from(*instance_to_inject*)

remove_commands_from(*instance_to_inject*)

remove_flows_from(*instance_to_inject*)

send(*identifier*, *text*, *in_reply_to=None*, *groupchat_nick_reply=False*)

> Sends a simple message to the specified user.
>
> > **Parameters**
> >
> > - **identifier** – an identifier from build_identifier or from an incoming message
> >
> > - **in_reply_to** – the original message the bot is answering from
> >
> > - **text** – the markdown text you want to send
> >
> > - **groupchat_nick_reply** (`bool`) – authorized the prefixing with the nick form the user

send_card(*card*)

> Sends a card, this can be overriden by the backends *without* a super() call.
>
> > **Parameters card** – the card to send.
> >
> > **Returns** None

send_message(*msg*)

> This needs to be overridden by the backends with a super() call.
>
> > **Parameters msg** – the message to send.
> >
> > **Returns** None

send_simple_reply(*msg*, *text*, *private=False*, *threaded=False*)

> Send a simple response to a given incoming message
>
> > **Parameters**
> >
> > - **private** (`bool`) – if True will force a response in private.
> >
> > - **threaded** (`bool`) – if True and if the backend supports it, sends the response in a threaded message.
> >
> > - **text** – the markdown text of the message.
> >
> > - **msg** – the message you are replying to.

send_templated(*identifier*, *template_name*, *template_parameters*, *in_reply_to=None*, *groupchat_nick_reply=False*)

> Sends a simple message to the specified user using a template.
>
> > **Parameters**
> >
> > - **template_parameters** – the parameters for the template.
> >
> > - **template_name** – the template name you want to use.
> >
> > - **identifier** – an identifier from build_identifier or from an incoming message, a room etc.

- **in_reply_to** – the original message the bot is answering from

- **groupchat_nick_reply** (bool) – authorized the prefixing with the nick form the user

shutdown()

signal_connect_to_all_plugins()

split_and_send_message(*msg*)

startup_time = datetime.datetime(2019, 7, 7, 14, 19, 22, 134115)

unknown_command(*_*, *cmd*, *args*)
> Override the default unknown command behavior

warn_admins(*warning: str*) → None
> Send a warning to the administrators of the bot.
>
> > **Parameters warning** (str) – The markdown-formatted text of the message to send.

errbot.flow module

class errbot.flow.**BotFlow**(*bot*, *name=None*)
> Bases: object
>
> Defines a Flow plugin ie. a plugin that will define new flows from its methods with the @botflow decorator.
>
> **__init__**(*bot*, *name=None*)
> > Initialize self. See help(type(self)) for accurate signature.
>
> **activate**() → None
> > Override if you want to do something at initialization phase (don't forget to super(Gnagna, self).activate())
>
> **deactivate**() → None
> > Override if you want to do something at tear down phase (don't forget to super(Gnagna, self).deactivate())
>
> **get_command**(*command_name: str*)
> > Helper to get a specific command.
>
> **name**
> > Get the name of this flow as described in its .plug file.
> >
> > > **Returns** The flow name.

errbot.flow.**FLOW_END = <errbot.flow._FlowEnd object>**
> Flow marker indicating that the flow ends.

class errbot.flow.**Flow**(*root: errbot.flow.FlowRoot, requestor: errbot.backends.base.Identifier, initial_context: Mapping[str, Any]*)
> Bases: object
>
> This is a live Flow. It keeps context of the conversation (requestor and context). Context is just a python dictionary representing the state of the conversation.
>
> **__init__**(*root: errbot.flow.FlowRoot, requestor: errbot.backends.base.Identifier, initial_context: Mapping[str, Any]*)
> > **Parameters**
> >
> > - **root** (*FlowRoot*) – the root of this flow.
> >
> > - **requestor** (*Identifier*) – the user requesting this flow.

- **initial_context** – any data we already have that could help executing this flow automatically.

advance(*next_step: errbot.flow.FlowNode*, *enforce_predicate=True*)

Move on along the flow. :type enforce_predicate: `bool` :type next_step: *FlowNode* :param next_step: Which node you want to move the flow forward to. :param enforce_predicate: Do you want to check if the predicate is verified for this step or not.

Usually, if it is a manual step, the predicate is irrelevant because the user will give the missing information as parameters to the command.

check_identifier(*identifier: errbot.backends.base.Identifier*)

current_step

The current step this Flow is waiting on.

name

Helper property to get the name of the flow.

next_autosteps() → List[errbot.flow.FlowNode]

Get the next steps that can be automatically executed according to the set predicates.

next_steps() → List[errbot.flow.FlowNode]

Get all the possible next steps after this one (predicates statisfied or not).

root

The original flowroot of this flow.

class errbot.flow.**FlowExecutor**(*bot*)

Bases: `object`

This is a instance that can monitor and execute flow instances.

__init__(*bot*)

Initialize self. See help(type(self)) for accurate signature.

add_flow(*flow: errbot.flow.FlowRoot*)

Register a flow with this executor.

check_inflight_already_running(*user: errbot.backends.base.Identifier*) → bool

Check if user is already running a flow.

Return type `bool`

Parameters **user** (*Identifier*) – the user

check_inflight_flow_triggered(*cmd: str*, *user: errbot.backends.base.Identifier*) → Tuple[Optional[errbot.flow.Flow], Optional[errbot.flow.FlowNode]]

Check if a command from a specific user was expected in one of the running flow. :type user: *Identifier* :type cmd: `str` :param cmd: the command that has just been executed. :param user: the identifier of the person who started this flow :returns: The name of the flow it triggered or None if none were matching.

execute(*flow: errbot.flow.Flow*)

This is where the flow execution happens from one of the thread of the pool.

start_flow(*name: str, requestor: errbot.backends.base.Identifier, initial_context: Mapping[str, Any]*) → errbot.flow.Flow

Starts the execution of a Flow.

stop_flow (*name: str*, *requestor: errbot.backends.base.Identifier*) → Optional[errbot.flow.Flow]
> Stops a specific flow. It is a no op if the flow doesn't exist. Returns the stopped flow if found.

trigger (*cmd: str*, *requestor: errbot.backends.base.Identifier*, *extra_context=None*) → Optional[errbot.flow.Flow]
> Trigger workflows that may have command cmd as a auto_trigger or an in flight flow waiting for command. This assume cmd has been correctly executed. :type requestor: `Identifier` :type cmd: `str` :param requestor: the identifier of the person who started this flow :param cmd: the command that has just been executed. :param extra_context: extra context from the current conversation :returns: The flow it triggered or None if none were matching.

class errbot.flow.**FlowNode** (*command: str = None*, *hints: bool = True*)
> Bases: `object`

> This is a step in a Flow/conversation. It is linked to a specific botcmd and also a "predicate".

> The predicate is a function that tells the flow executor if the flow can enter the step without the user intervention (automatically). The predicates defaults to False.

> The predicate is a function that takes one parameter, the context of the conversation.

> **__init__** (*command: str = None*, *hints: bool = True*)
> > Creates a FlowNone, takes the command to which the Node is linked to. :type hints: `bool` :type command: `str` :param command: the command this Node is linked to. Can only be None if this Node is a Root. :param hints: hints the users for the next steps in chat.

> **connect** (*node_or_command: Union[FlowNode, str]*, *predicate: Callable[[Mapping[str, Any]], bool] = <function FlowNode.<lambda>>*, *hints: bool = True*)
> > Construct the flow graph by connecting this node to another node or a command. The predicate is a function that tells the flow executor if the flow can enter the step without the user intervention (automatically). :type hints: `bool` :param node_or_command: the node or a string for a command you want to connect this Node to

> > (this node or command will be the follow up of this one)

> > **Parameters**
> > - **predicate** – function with one parameter, the context, to determine of the flow executor can continue automatically this flow with no user intervention.
> > - **hints** – hints the user on the next step possible.

> > **Returns** the newly created node if you passed a command or the node you gave it to be easily chainable.

> **predicate_for_node** (*node: errbot.flow.FlowNode*)
> > gets the predicate function for the specified child node. :param node: the child node :return: the predicate that allows the automatic execution of that node.

class errbot.flow.**FlowRoot** (*name: str*, *description: str*)
> Bases: `errbot.flow.FlowNode`

> This represent the entry point of a flow description.

> **__init__** (*name: str*, *description: str*)
> > **Parameters**
> > - **name** (`str`) – The name of the conversation/flow.
> > - **description** (`str`) – A human description of what this flow does.
> > - **hints** – Hints for the next steps when triggered.

connect (*node_or_command: Union[FlowNode, str], predicate: Callable[[Mapping[str, Any]], bool]* *= <function FlowRoot.<lambda>>, auto_trigger: bool = False, room_flow: bool = False*)

> **See** FlowNode except fot auto_trigger
>
> **Parameters**
>
> - **predicate** –
>
>   see FlowNode
>
> - **node_or_command** –
>
>   see FlowNode
>
> - **auto_trigger** (`bool`) – Flag this root as autotriggering: it will start a flow if this command is executed in the chat.
>
> - **room_flow** (`bool`) – Bind the flow to the room instead of a single person

exception errbot.flow.**InvalidState**

> Bases: `Exception`
>
> Raised when the Flow Executor is asked to do something contrary to the contraints it has been given.

errbot.logs module

errbot.logs.**format_logs** (*formatter=None*, *theme_color=None*)

> You may either use the formatter parameter to provide your own custom formatter, or the theme_color parameter to use the built in color scheme formatter.

errbot.logs.**get_log_colors** (*theme_color=None*)

> Return a tuple containing the log format string and a log color dict

errbot.logs.**ispydevd** ()

errbot.plugin_info module

class errbot.plugin_info.**PluginInfo** (*name: str,   module: str,   doc: str,   core: bool, python_version: Tuple[int, int, int], errbot_minversion: Tuple[int, int, int], errbot_maxversion: Tuple[int, int, int], dependencies: List[str], location: pathlib.Path = None*)

> Bases: `object`
>
> **__init__** (*name: str, module: str, doc: str, core: bool, python_version: Tuple[int, int, int], errbot_minversion: Tuple[int, int, int], errbot_maxversion: Tuple[int, int, int], dependencies: List[str], location: pathlib.Path = None*) → None
>
> **static load** (*plugfile_path: pathlib.Path*) → errbot.plugin_info.PluginInfo
>
> **static load_file** (*plugfile, location: pathlib.Path*) → errbot.plugin_info.PluginInfo
>
> **load_plugin_classes** (*base_module_name: str, baseclass: Type[CT_co]*)
>
> **location = None**
>
> **static parse** (*config: configparser.ConfigParser*) → errbot.plugin_info.PluginInfo
>
> > **Throws ConfigParserError with a meaningful message if the ConfigParser doesn't contain the minimal** information required.

errbot.plugin_manager module

Logic related to plugin loading and lifecycle

class errbot.plugin_manager.**BotPluginManager**(*storage_plugin:* *er-*
rbot.storage.base.StoragePluginBase,
extra_plugin_dir: *Optional[str],*
autoinstall_deps: *bool,*
core_plugins: *Tuple[str, ...], plu-*
gin_instance_callback: *Callable[[str,*
Type[errbot.botplugin.BotPlugin]],
errbot.botplugin.BotPlugin], *plug-*
ins_callback_order: *Tuple[Optional[str],*
...])

Bases: *errbot.storage.StoreMixin*

__init__(*storage_plugin: errbot.storage.base.StoragePluginBase, extra_plugin_dir: Optional[str],*
autoinstall_deps: *bool, core_plugins:* *Tuple[str, ...], plugin_instance_callback:*
Callable[[str, Type[errbot.botplugin.BotPlugin]], errbot.botplugin.BotPlugin], plug-
ins_callback_order: Tuple[Optional[str], ...])

Creates a Plugin manager :type autoinstall_deps: bool :type storage_plugin: *StoragePluginBase*
:param storage_plugin: the plugin used to store to config for this manager :param extra_plugin_dir:
an extra directory to search for plugins :param autoinstall_deps: if True, will install also the plugin
deps from requirements.txt :param core_plugins: the list of core plugin that will be started :param plu-
gin_instance_callback: the callback to instantiate a plugin (to inject the dependency on the bot) :param
plugins_callback_order: the order on which the plugins will be callbacked

activate_flow(*name: str*)

activate_non_started_plugins()

Activates all plugins that are not activated, respecting its dependencies.

> **Returns** Empty string if no problem occured or a string explaining what went wrong.

activate_plugin(*name: str*)

Activate a plugin with its dependencies.

blacklist_plugin(*name*)

deactivate_all_plugins()

deactivate_flow(*name: str*)

deactivate_plugin(*name: str*)

get_all_active_plugin_names()

get_all_active_plugins() → List[errbot.botplugin.BotPlugin]

This returns the list of plugins in the callback ordered defined from the config.

get_all_plugin_names()

get_blacklisted_plugin()

get_plugin_configuration(*name*)

get_plugin_obj_by_name(*name: str*) → errbot.botplugin.BotPlugin

is_plugin_blacklisted(*name*)

reload_plugin_by_name(*name*)

Completely reload the given plugin, including reloading of the module's code :throws PluginActiva-
tionException: needs to be taken care of by the callers.

remove_plugin(*plugin: errbot.botplugin.BotPlugin*)

> Deactivate and remove a plugin completely. :type plugin: [*BotPlugin*](#) :param plugin: the plugin to remove :return:

remove_plugins_from_path(*root*)

> Remove all the plugins that are in the filetree pointed by root.

set_plugin_configuration(*name*, *obj*)

shutdown()

unblacklist_plugin(*name*)

update_plugin_places(*path_list*) → Dict[pathlib.Path, str]

> This updates where this manager is trying to find plugins and try to load newly found ones. :param path_list: the path list where to search for plugins. :return: the feedback for any specific path in case of error.

exception errbot.plugin_manager.**IncompatiblePluginException**

> Bases: [*errbot.plugin_manager.PluginActivationException*](#)

exception errbot.plugin_manager.**PluginActivationException**

> Bases: Exception

exception errbot.plugin_manager.**PluginConfigurationException**

> Bases: [*errbot.plugin_manager.PluginActivationException*](#)

errbot.plugin_manager.**check_errbot_version**(*plugin_info: errbot.plugin_info.PluginInfo*)

> Checks if a plugin version between min_version and max_version is ok for this errbot. Raises Incompatible-PluginException if not.

errbot.plugin_manager.**check_python_plug_section**(*plugin_info:* *errbot.plugin_info.PluginInfo*) → bool

> Checks if we have the correct version to run this plugin. Returns true if the plugin is loadable

errbot.plugin_manager.**install_packages**(*req_path: pathlib.Path*)

> Installs all the packages from the given requirements.txt

> Return an exc_info if it fails otherwise None.

errbot.plugin_manager.**populate_doc**(*plugin_object:* *errbot.botplugin.BotPlugin*, *plugin_info:* *errbot.plugin_info.PluginInfo*) → None

errbot.plugin_wizard module

errbot.plugin_wizard.**ask**(*question*, *valid_responses=None*, *validation_regex=None*)

> Ask the user for some input. If valid_responses is supplied, the user must respond with something present in this list.

errbot.plugin_wizard.**new_plugin_wizard**(*directory=None*)

> Start the wizard to create a new plugin in the current working directory.

errbot.plugin_wizard.**render_plugin**(*values*)

> Render the Jinja template for the plugin with the given values.

errbot.repo_manager module

class errbot.repo_manager.**BotRepoManager**(*storage_plugin:* *er-* *rbot.storage.base.StoragePluginBase, plugin_dir:* *str, plugin_indexes: Tuple[str, ...]*)

Bases: *errbot.storage.StoreMixin*

Manages the repo list, git clones/updates or the repos.

__init__(*storage_plugin: errbot.storage.base.StoragePluginBase, plugin_dir: str, plugin_indexes: Tuple[str, ...]*) → None

Make a repo manager. :type plugin_dir: str :type storage_plugin: *StoragePluginBase* :param storage_plugin: where the manager store its state. :param plugin_dir: where on disk it will git clone the repos. :param plugin_indexes: a list of URL / path to get the json repo index.

add_plugin_repo(*name: str*, *url: str*) → None

check_for_index_update() → None

get_all_repos_paths() → List[str]

get_installed_plugin_repos() → Dict[str, str]

get_repo_from_index(*repo_name: str*) → List[errbot.repo_manager.RepoEntry]

Retrieve the list of plugins for the repo_name from the index.

> **Parameters** **repo_name** (str) – the name of the repo
>
> **Returns** a list of RepoEntry

index_update() → None

install_repo(*repo: str*) → str

Install the repository from repo

> **Return type** str
>
> **Parameters** **repo** (str) –
>
>> **The url, git url or path on disk of a repository. It can point to either a git repo or** a .tar.gz of a plugin
>
> **Returns** The path on disk where the repo has been installed on.
>
> **Raises** *RepoException* if an error occured.

search_repos(*query: str*) → Generator[errbot.repo_manager.RepoEntry, None, None]

A simple search feature, keywords are AND and case insensitive on all the fields.

> **Parameters** **query** (str) – a string query
>
> **Returns** an iterator of RepoEntry

set_plugin_repos(*repos: Dict[str, str]*) → None

Used externally.

shutdown() → None

uninstall_repo(*name: str*) → None

update_all_repos() → Generator[Tuple[str, int, str], None, None]

update_repos(*repos*) → Generator[Tuple[str, int, str], None, None]

This git pulls the specified repos on disk. Yields tuples like (name, success, reason)

class errbot.repo_manager.**RepoEntry**(*entry_name*, *name*, *python*, *repo*, *path*, *avatar_url*, *documentation*)

    Bases: tuple

    avatar_url
        Alias for field number 5

    documentation
        Alias for field number 6

    entry_name
        Alias for field number 0

    name
        Alias for field number 1

    path
        Alias for field number 4

    python
        Alias for field number 2

    repo
        Alias for field number 3

exception errbot.repo_manager.**RepoException**

    Bases: Exception

errbot.repo_manager.**check_dependencies**(*req_path: pathlib.Path*) → Tuple[Optional[str], Sequence[str]]

    This methods returns a pair of (message, packages missing). Or None, [] if everything is OK.

errbot.repo_manager.**human_name_for_git_url**(*url*)

errbot.repo_manager.**makeEntry**(*repo_name: str*, *plugin_name: str*, *json_value*)

errbot.repo_manager.**tokenizeJsonEntry**(*json_dict*)

    Returns all the words in a repo entry.

errbot.repo_manager.**which**(*program*)

errbot.streaming module

class errbot.streaming.**Tee**(*incoming_stream*, *clients*)

    Bases: object

    Tee implements a multi reader / single writer

    __init__(*incoming_stream*, *clients*)
        clients is a list of objects implementing callback_stream

    run()
        streams to all the clients synchronously

    start()
        starts the transfer asynchronously

errbot.streaming.**repeatfunc**(*func*, *times=None*, **args*)

    Repeat calls to func with specified arguments.

    Example: repeatfunc(random.random)

        Parameters

- **args** – params to the function to call.
- **times** – number of times to repeat.
- **func** – the function to repeatedly call.

errbot.templating module

errbot.templating.**add_plugin_templates_path**(*plugin_info: errbot.plugin_info.PluginInfo*)

errbot.templating.**make_templates_path**(*root: pathlib.Path*) → pathlib.Path

errbot.templating.**remove_plugin_templates_path**(*plugin_info: errbot.plugin_info.PluginInfo*)

errbot.templating.**tenv**()

errbot.utils module

errbot.utils.**collect_roots**(*base_paths*, *file_sig='*.plug'*)
　　Collects all the paths from base_paths recursively that contains files of type *file_sig*.

　　　　Parameters

- **base_paths** – a list of base paths to walk from elements can be a string or a list/tuple of strings
- **file_sig** (str) – the file pattern to look for

　　　　Returns a set of paths

class errbot.utils.**deprecated**(*new=None*)
　　Bases: object

　　deprecated decorator. emits a warning on a call on an old method and call the new method anyway

　　__init__(*new=None*)
　　　　Initialize self. See help(type(self)) for accurate signature.

errbot.utils.**find_roots**(*path*, *file_sig='*.plug'*)
　　Collects all the paths from path recursively that contains files of type *file_sig*.

　　　　Parameters

- **path** – a base path to walk from
- **file_sig** (str) – the file pattern to look for

　　　　Returns a set of paths

errbot.utils.**format_timedelta**(*timedelta*)

errbot.utils.**git_clone**(*url: str*, *path: str*) → None
　　Clones a repository from git url to path

errbot.utils.**git_pull**(*repo_path: str*) → None
　　Does a git pull on a repository

errbot.utils.**git_tag_list**(*repo_path: str*) → List[str]
　　Lists git tags on a cloned repo

errbot.utils.**global_restart**()
　　Restart the current process.

errbot.utils.**rate_limited**(*min_interval*)
    decorator to rate limit a function.

> > **Parameters** **min_interval** – minimum interval allowed between 2 consecutive calls.
>
> > **Returns** the decorated function

errbot.utils.**split_string_after**(*str_*, *n*)
    Yield chunks of length *n* from the given string

> > **Parameters**
> >
> > > • **n** – length of the chunks.
> > >
> > > • **str** – the given string.

errbot.utils.**version2tuple**(*version*)

errbot.version module

8.1.3 Module contents

class errbot.**BotPlugin**(*bot*, *name=None*)
    Bases: *errbot.botplugin.BotPluginBase*

    activate() → None
        Triggered on plugin activation.

        Override this method if you want to do something at initialization phase (don't forget to *super().activate()*).

    build_identifier(*txtrep: str*) → errbot.backends.base.Identifier
        Transform a textual representation of a user identifier to the correct Identifier object you can set in Message.to and Message.frm.

> > **Return type** *Identifier*

> > **Parameters** **txtrep** (str) – the textual representation of the identifier (it is backend dependent).

> > **Returns** a user identifier.

    callback_botmessage(*message: errbot.backends.base.Message*)
        Triggered on every message coming from the bot itself.

        Override this method to get notified on all messages coming from the bot itself (including those from other plugins).

> > **Parameters** **message** (*Message*) – An instance of *Message* representing the message that was received.

    callback_connect() → None
        Triggered when the bot has successfully connected to the chat network.

        Override this method to get notified when the bot is connected.

    callback_mention(*message: errbot.backends.base.Message, mentioned_people: Sequence[errbot.backends.base.Identifier]*) → None
        Triggered if there are mentioned people in message.

        Override this method to get notified when someone was mentioned in message. [Note: This might not be implemented by all backends.]

> > **Parameters**

- **message** (`Message`) – representing the message that was received.

- **mentioned_people** – all mentioned people in this message.

callback_message (*message: errbot.backends.base.Message*) → None
    Triggered on every message not coming from the bot itself.

    Override this method to get notified on *ANY* message.

        Parameters message (`Message`) – representing the message that was received.

callback_presence (*presence: errbot.backends.base.Presence*) → None
    Triggered on every presence change.

        Parameters presence (`Presence`) – An instance of `Presence` representing the new presence state that was received.

callback_room_joined (*room: errbot.backends.base.Room*)
    Triggered when the bot has joined a MUC.

        Parameters room (`Room`) – An instance of MUCRoom representing the room that was joined.

callback_room_left (*room: errbot.backends.base.Room*)
    Triggered when the bot has left a MUC.

        Parameters room (`Room`) – An instance of MUCRoom representing the room that was left.

callback_room_topic (*room: errbot.backends.base.Room*)
    Triggered when the topic in a MUC changes.

        Parameters room (`Room`) – An instance of MUCRoom representing the room for which the topic changed.

callback_stream (*stream: errbot.backends.base.Stream*) → None
    Triggered asynchronously (in a different thread context) on every incoming stream request or file transfert requests. You can block this call until you are done with the stream. To signal that you accept / reject the file, simply call stream.accept() or stream.reject() and return.

        Parameters stream (`Stream`) – the incoming stream request.

change_presence (*status: str = 'online'*, *message: str = ''*) → None

    Changes the presence/status of the bot.

        Parameters

        - **status** (`str`) – One of the constant defined in base.py : ONLINE, OFFLINE, DND,. . .

        - **message** (`str`) – Additional message

        Returns None

check_configuration (*configuration: Mapping[KT, VT_co]*) → None
    By default, this method will do only a BASIC check. You need to override it if you want to do more complex checks. It will be called before the configure callback. Note if the config_template is None, it will never be called.

    It means recusively:

    1. in case of a dictionary, it will check if all the entries and from the same type are there and not more.

    2. in case of an array or tuple, it will assume array members of the same type of first element of the template (no mix typed is supported)

In case of validation error it should raise a errbot.ValidationException

> **Parameters** **configuration** – the configuration to be checked.

configure(*configuration: Mapping[KT, VT_co]*) → None
> By default, it will just store the current configuration in the self.config field of your plugin. If this plugin has no configuration yet, the framework will call this function anyway with None.
>
> This method will be called before activation so don't expect to be activated at that point.
>
> > **Parameters** **configuration** – injected configuration for the plugin.

deactivate() → None
> Triggered on plugin deactivation.
>
> Override this method if you want to do something at tear-down phase (don't forget to *super().deactivate()*).

get_configuration_template() → Mapping[KT, VT_co]
> If your plugin needs a configuration, override this method and return a configuration template.
>
> For example a dictionary like: return {'LOGIN' : 'example@example.com', 'PASSWORD' : 'password'}
>
> Note: if this method returns None, the plugin won't be configured

query_room(*room: str*) → errbot.backends.base.Room
> Query a room for information.
>
> > **Return type** *Room*
> >
> > **Parameters** **room** (str) – The JID/identifier of the room to query for.
> >
> > **Returns** An instance of MUCRoom.
> >
> > **Raises** *RoomDoesNotExistError* if the room doesn't exist.

rooms() → Sequence[errbot.backends.base.Room]
> The list of rooms the bot is currently in.

send(*identifier: errbot.backends.base.Identifier*, *text: str*, *in_reply_to: errbot.backends.base.Message = None*, *groupchat_nick_reply: bool = False*) → None
Send a message to a room or a user.

> **Parameters**
>
> - **groupchat_nick_reply** (bool) – if True the message will mention the user in the chatroom.
>
> - **in_reply_to** (*Message*) – the original message this message is a reply to (optional). In some backends it will start a thread.
>
> - **text** (str) – markdown formatted text to send to the user.
>
> - **identifier** (*Identifier*) – An Identifier representing the user or room to message. Identifiers may be created with *build_identifier()*.

send_card(*body: str = ''*, *to: errbot.backends.base.Identifier = None*, *in_reply_to: errbot.backends.base.Message = None*, *summary: str = None*, *title: str = ''*, *link: str = None*, *image: str = None*, *thumbnail: str = None*, *color: str = 'green'*, *fields: Tuple[Tuple[str, str], ...] = ()*) → None
Sends a card.

> A Card is a special type of preformatted message. If it matches with a backend similar concept like on Slack or Hipchat it will be rendered natively, otherwise it will be sent as a regular formatted message.
>
> **Parameters**
>
> - **body** (str) – main text of the card in markdown.

- **to** (*Identifier*) – the card is sent to this identifier (Room, RoomOccupant, Person...).

- **in_reply_to** (*Message*) – the original message this message is a reply to (optional).

- **summary** (str) – (optional) One liner summary of the card, possibly collapsed to it.

- **title** (str) – (optional) Title possibly linking.

- **link** (str) – (optional) url the title link is pointing to.

- **image** (str) – (optional) link to the main image of the card.

- **thumbnail** (str) – (optional) link to an icon / thumbnail.

- **color** (str) – (optional) background color or color indicator.

- **fields** – (optional) a tuple of (key, value) pairs.

send_stream_request(*user: errbot.backends.base.Identifier, fsource: io.IOBase, name: str = None, size: int = None, stream_type: str = None*)
    Sends asynchronously a stream/file to a user.

> **Parameters**
>
> - **user** (*Identifier*) – is the identifier of the person you want to send it to.
>
> - **fsource** (IOBase) – is a file object you want to send.
>
> - **name** (str) – is an optional filename for it.
>
> - **size** (int) – is optional and is the espected size for it.
>
> - **stream_type** (str) – is optional for the mime_type of the content.

> It will return a Stream object on which you can monitor the progress of it.

send_templated(*identifier: errbot.backends.base.Identifier,   template_name: str, template_parameters: Mapping[KT,   VT_co],   in_reply_to: errbot.backends.base.Message = None, groupchat_nick_reply: bool = False*)
    → None
    Sends asynchronously a message to a room or a user.

> Same as send but passing a template name and parameters instead of directly the markdown text. :type groupchat_nick_reply: bool :type in_reply_to: *Message* :type template_name: str :type identifier: *Identifier* :param template_parameters: arguments for the template. :param template_name: name of the template to use. :param groupchat_nick_reply: if True it will mention the user in the chatroom. :param in_reply_to: optionally, the original message this message is the answer to. :param identifier: identifier of the user or room to which you want to send a message to.

start_poller(*interval: float, method: Callable[[...], None], times: int = None, args: Tuple = None, kwargs: Mapping[KT, VT_co] = None*)
    Start to poll a method at specific interval in seconds.

> Note: it will call the method with the initial interval delay for the first time

> Also, you can program for example : self.program_poller(self, 30, fetch_stuff) where you have def fetch_stuff(self) in your plugin

> **Parameters**
>
> - **interval** (float) – interval in seconds
>
> - **method** – targetted method

> - **times** (int) – number of times polling should happen (defaults to``None`` which causes the polling to happen indefinitely)
>
> - **args** – args for the targetted method
>
> - **kwargs** – kwargs for the targetting method

stop_poller(*method: Callable[[...], None], args: Tuple = None, kwargs: Mapping[KT, VT_co] = None*)
    stop poller(s).

    If the method equals None -> it stops all the pollers you need to regive the same parameters as the original start_poller to match a specific poller to stop

    > **Parameters**
    >
    > > - **kwargs** – The initial kwargs you gave to start_poller.
    > >
    > > - **args** – The initial args you gave to start_poller.
    > >
    > > - **method** – The initial method you passed to start_poller.

warn_admins(*warning: str*) → None
    Send a warning to the administrators of the bot.

    > **Parameters warning** (str) – The markdown-formatted text of the message to send.

exception errbot.**CommandError**(*reason: str*, *template: str = None*)
    Bases: Exception

    Use this class to report an error condition from your commands, the command did not proceed for a known "business" reason.

    __init__(*reason: str*, *template: str = None*)

    > **Parameters**
    >
    > > - **reason** (str) – the reason for the error in the command.
    > >
    > > - **template** (str) – apply this specific template to report the error.

class errbot.**Command**(*function, cmd_type=None, cmd_args=None, cmd_kwargs=None, name=None, doc=None*)
    Bases: object

    This is a dynamic definition of an errbot command.

    __init__(*function, cmd_type=None, cmd_args=None, cmd_kwargs=None, name=None, doc=None*)
        Create a Command definition.

    > **Parameters**
    >
    > > - **function** – a function or a lambda with the correct signature for the type of command to inject for example *def mycmd(plugin, msg, args)* for a botcmd. Note: the first parameter will be the plugin itself (equivalent to self).
    > >
    > > - **cmd_type** – defaults to *botcmd* but can be any decorator function used for errbot commands.
    > >
    > > - **cmd_args** – the parameters of the decorator.
    > >
    > > - **cmd_kwargs** – the kwargs parameter of the decorator.
    > >
    > > - **name** – defaults to the name of the function you are passing if it is a first class function or needs to be set if you use a lambda.

> > - **doc** – defaults to the doc of the given function if it is a first class function. It can be
> >   set for a lambda or overridden for a function with this.

errbot.**webhook**(**args*, *methods: Tuple[str] = ('POST', 'GET')*, *form_param: str = None*, *raw: bool =
        False*) → Callable[[errbot.botplugin.BotPlugin, Any], str]
> Decorator for webhooks

> > **Parameters**

> > > - **uri_rule** – The URL to use for this webhook, as per Bottle request routing syntax. For
> > >   more information, see:

> > > > - http://bottlepy.org/docs/dev/tutorial.html#request-routing

> > > > - http://bottlepy.org/docs/dev/routing.html

> > > - **methods** – A tuple of allowed HTTP methods. By default, only GET and POST are
> > >   allowed.

> > > - **form_param** (str) – The key who's contents will be passed to your method's *payload*
> > >   parameter. This is used for example when using the *application/x-www-form-urlencoded*
> > >   mimetype.

> > > - **raw** (bool) – When set to true, this overrides the request decoding (including
> > >   form_param) and passes the raw http request to your method's *payload* parameter. The
> > >   value of payload will be a Bottle BaseRequest.

> This decorator should be applied to methods of *BotPlugin* classes to turn them into webhooks which can be
> reached on Err's built-in webserver. The bundled *Webserver* plugin needs to be configured before these URL's
> become reachable.

> Methods with this decorator are expected to have a signature like the following:

```
@webhook
def a_webhook(self, payload):
    pass
```

errbot.**webroute**(*obj*)
> Check for functions to route in obj and route them.

errbot.**cmdfilter**(**args*, ***kwargs*)
> Decorator for command filters.

> This decorator should be applied to methods of *BotPlugin* classes to turn them into command filters.

> These filters are executed just before the execution of a command and provide the means to add features such as
> custom security, logging, auditing, etc.

> These methods are expected to have a signature and tuple response like the following:

```
@cmdfilter
def some_filter(self, msg, cmd, args, dry_run):
    """
    :param msg: The original chat message.
    :param cmd: The command name itself.
    :param args: Arguments passed to the command.
    :param dry_run: True when this is a dry-run.
       Dry-runs are performed by certain commands (such as !help)
       to check whether a user is allowed to perform that command
       if they were to issue it. If dry_run is True then the plugin
       shouldn't actually do anything beyond returning whether the
       command is authorized or not.
```

(continues on next page)

```
    """
    # If wishing to block the incoming command:
    return None, None, None
    # Otherwise pass data through to the (potential) next filter:
    return msg, cmd, args
```

Note that a cmdfilter plugin *could* modify *cmd* or *args* above and send that through in order to make it appear as if the user issued a different command.

errbot.**botcmd**(**args*, *hidden: bool = None*, *name: str = None*, *split_args_with: str = ''*, *admin_only: bool = False*, *historize: bool = True*, *template: str = None*, *flow_only: bool = False*, *syntax: str = None*) → Callable[[errbot.botplugin.BotPlugin, errbot.backends.base.Message, Any], Any]
Decorator for bot command functions

> **Parameters**
>
> > - **hidden** (`bool`) – Prevents the command from being shown by the built-in help command when *True*.
> >
> > - **name** (`str`) – The name to give to the command. Defaults to name of the function itself.
> >
> > - **split_args_with** (`str`) – Automatically split arguments on the given separator. Behaviour of this argument is identical to `str.split()`
> >
> > - **admin_only** (`bool`) – Only allow the command to be executed by admins when *True*.
> >
> > - **historize** (`bool`) – Store the command in the history list (*!history*). This is enabled by default.
> >
> > - **template** (`str`) – The markdown template to use.
> >
> > - **syntax** (`str`) – The argument syntax you expect for example: '[name] <mandatory>'.
> >
> > - **flow_only** (`bool`) – Flag this command to be available only when it is part of a flow. If True and hidden is None, it will switch hidden to True.

This decorator should be applied to methods of *BotPlugin* classes to turn them into commands that can be given to the bot. These methods are expected to have a signature like the following:

```
@botcmd
def some_command(self, msg, args):
    pass
```

The given *msg* will be the full message object that was received, which includes data like sender, receiver, the plain-text and html body (if applicable), etc. *args* will be a string or list (depending on your value of *split_args_with*) of parameters that were given to the command by the user.

errbot.**re_botcmd**(**args*, *hidden: bool = None*, *name: str = None*, *admin_only: bool = False*, *historize: bool = True*, *template: str = None*, *pattern: str = None*, *flags: int = 0*, *matchall: bool = False*, *prefixed: bool = True*, *flow_only: bool = False*, *re_cmd_name_help: str = None*) → Callable[[errbot.botplugin.BotPlugin, errbot.backends.base.Message, Any], Any]
Decorator for regex-based bot command functions

> **Parameters**
>
> > - **pattern** (`str`) – The regular expression a message should match against in order to trigger the command.

- **flags** (int) – The *flags* parameter which should be passed to `re.compile()`. This allows the expression's behaviour to be modified, such as making it case-insensitive for example.

- **matchall** (bool) – By default, only the first match of the regular expression is returned (as a *re.MatchObject*). When *matchall* is *True*, all non-overlapping matches are returned (as a list of *re.MatchObject* items).

- **prefixed** (bool) – Requires user input to start with a bot prefix in order for the pattern to be applied when *True* (the default).

- **hidden** (bool) – Prevents the command from being shown by the built-in help command when *True*.

- **name** (str) – The name to give to the command. Defaults to name of the function itself.

- **admin_only** (bool) – Only allow the command to be executed by admins when *True*.

- **historize** (bool) – Store the command in the history list (*!history*). This is enabled by default.

- **template** (str) – The template to use when using markdown output

- **flow_only** (bool) – Flag this command to be available only when it is part of a flow. If True and hidden is None, it will switch hidden to True.

This decorator should be applied to methods of [`BotPlugin`](#) classes to turn them into commands that can be given to the bot. These methods are expected to have a signature like the following:

```
@re_botcmd(pattern=r'^some command$')
def some_command(self, msg, match):
    pass
```

The given *msg* will be the full message object that was received, which includes data like sender, receiver, the plain-text and html body (if applicable), etc. *match* will be a `re.MatchObject` containing the result of applying the regular expression on the user's input.

errbot.**arg_botcmd**(*args*, *hidden: bool = None*, *name: str = None*, *admin_only: bool = False*, *historize: bool = True*, *template: str = None*, *flow_only: bool = False*, *unpack_args: bool = True*, ***kwargs*) → Callable[[errbot.botplugin.BotPlugin, errbot.backends.base.Message, Any], Any]
Decorator for argparse-based bot command functions

https://docs.python.org/3/library/argparse.html

This decorator creates an argparse.ArgumentParser and uses it to parse the commands arguments.

This decorator can be used multiple times to specify multiple arguments.

Any valid argparse.add_argument() parameters can be passed into the decorator. Each time this decorator is used it adds a new argparse argument to the command.

Parameters

- **hidden** (bool) – Prevents the command from being shown by the built-in help command when *True*.

- **name** (str) – The name to give to the command. Defaults to name of the function itself.

- **admin_only** (bool) – Only allow the command to be executed by admins when *True*.

- **historize** (bool) – Store the command in the history list (*!history*). This is enabled by default.

- **template** (str) – The template to use when using markdown output

- **flow_only** (bool) – Flag this command to be available only when it is part of a flow. If True and hidden is None, it will switch hidden to True.

- **unpack_args** (bool) – Should the argparser arguments be "unpacked" and passed on the the bot command individually? If this is True (the default) you must define all arguments in the function separately. If this is False you must define a single argument *args* (or whichever name you prefer) to receive the result of *ArgumentParser.parse_args()*.

This decorator should be applied to methods of [`BotPlugin`](#) classes to turn them into commands that can be given to the bot. The methods will be called with the original msg and the argparse parsed arguments. These methods are expected to have a signature like the following (assuming *unpack_args=True*):

```python
@arg_botcmd('value', type=str)
@arg_botcmd('--repeat-count', dest='repeat', type=int, default=2)
def repeat_the_value(self, msg, value=None, repeat=None):
    return value * repeat
```

The given *msg* will be the full message object that was received, which includes data like sender, receiver, the plain-text and html body (if applicable), etc. *value* will hold the value passed in place of the *value* argument and *repeat* will hold the value passed in place of the *–repeat-count* argument.

If you don't like this automatic *"unpacking"* of the arguments, you can use *unpack_args=False* like this:

```python
@arg_botcmd('value', type=str)
@arg_botcmd('--repeat-count', dest='repeat', type=int, default=2, unpack_
→args=False)
def repeat_the_value(self, msg, args):
    return arg.value * args.repeat
```

Note: The *unpack_args=False* only needs to be specified once, on the bottom *@args_botcmd* statement.

---

errbot.**botflow**(*args*, ***kwargs*)
　　Decorator for flow of commands.

　　TODO(gbin): example / docs

errbot.**botmatch**(*args*, ***kwargs*)
　　Decorator for regex-based message match.

　　　　Parameters

- ***args** – The regular expression a message should match against in order to trigger the command.

- **flags** – The *flags* parameter which should be passed to `re.compile()`. This allows the expression's behaviour to be modified, such as making it case-insensitive for example.

- **matchall** – By default, only the first match of the regular expression is returned (as a *re.MatchObject*). When *matchall* is *True*, all non-overlapping matches are returned (as a list of *re.MatchObject* items).

- **hidden** – Prevents the command from being shown by the built-in help command when *True*.

- **name** – The name to give to the command. Defaults to name of the function itself.

- **admin_only** – Only allow the command to be executed by admins when *True*.

- **historize** – Store the command in the history list (*!history*). This is enabled by default.

- **template** – The template to use when using Markdown output.

- **flow_only** – Flag this command to be available only when it is part of a flow. If True and hidden is None, it will switch hidden to True.

For example:

```
@botmatch(r'^(?:Yes|No)$')
def yes_or_no(self, msg, match):
    pass
```

class errbot.**BotFlow**(*bot*, *name=None*)

Bases: `object`

Defines a Flow plugin ie. a plugin that will define new flows from its methods with the @botflow decorator.

__init__(*bot*, *name=None*)

Initialize self. See help(type(self)) for accurate signature.

activate() → None

Override if you want to do something at initialization phase (don't forget to super(Gnagna, self).activate())

deactivate() → None

Override if you want to do something at tear down phase (don't forget to super(Gnagna, self).deactivate())

get_command(*command_name: str*)

Helper to get a specific command.

name

Get the name of this flow as described in its .plug file.

> **Returns** The flow name.

class errbot.**FlowRoot**(*name: str*, *description: str*)

Bases: `errbot.flow.FlowNode`

This represent the entry point of a flow description.

__init__(*name: str*, *description: str*)

> **Parameters**

> - **name** (`str`) – The name of the conversation/flow.

> - **description** (`str`) – A human description of what this flow does.

> - **hints** – Hints for the next steps when triggered.

connect(*node_or_command: Union[FlowNode, str]*, *predicate: Callable[[Mapping[str, Any]], bool]* = *<function FlowRoot.<lambda>>*, *auto_trigger: bool = False*, *room_flow: bool = False*)

> **See** FlowNode except fot auto_trigger

> **Parameters**

> - **predicate** –

>   see FlowNode

> - **node_or_command** –

>   see FlowNode

> - **auto_trigger** (`bool`) – Flag this root as autotriggering: it will start a flow if this command is executed in the chat.

> - **room_flow** (`bool`) – Bind the flow to the room instead of a single person

class errbot.**Flow**(*root:* *errbot.flow.FlowRoot,* *requestor:* *errbot.backends.base.Identifier,* *initial_context: Mapping[str, Any]*)

    Bases: `object`

    This is a live Flow. It keeps context of the conversation (requestor and context). Context is just a python dictionary representing the state of the conversation.

    __init__(*root:* *errbot.flow.FlowRoot,* *requestor:* *errbot.backends.base.Identifier,* *initial_context: Mapping[str, Any]*)

        Parameters

- **root** (*FlowRoot*) – the root of this flow.
- **requestor** (*Identifier*) – the user requesting this flow.
- **initial_context** – any data we already have that could help executing this flow automatically.

    advance(*next_step: errbot.flow.FlowNode*, *enforce_predicate=True*)

        Move on along the flow. :type enforce_predicate: bool :type next_step: *FlowNode* :param next_step: Which node you want to move the flow forward to. :param enforce_predicate: Do you want to check if the predicate is verified for this step or not.

        Usually, if it is a manual step, the predicate is irrelevant because the user will give the missing information as parameters to the command.

    check_identifier(*identifier: errbot.backends.base.Identifier*)

    current_step

        The current step this Flow is waiting on.

    name

        Helper property to get the name of the flow.

    next_autosteps() → List[errbot.flow.FlowNode]

        Get the next steps that can be automatically executed according to the set predicates.

    next_steps() → List[errbot.flow.FlowNode]

        Get all the possible next steps after this one (predicates statisfied or not).

    root

        The original flowroot of this flow.

Release history

9.1 v6.1.1 (2019-06-22)

fixes:

- Installation using wheel distribution on python 3.6 or older

9.2 v6.1.0 (2019-06-16)

features:

- Use python git instead of system git binary (#1296)

fixes:

- *errbot -l* cli error (#1315)

- Slack backend by pinning slackclient to supported version (#1343)

- Make –storage-merge merge configs (#1311)

- Exporting values in backup command (#1328)

- Rename Spark to Webex Teams (#1323)

- Various documentation fixes (#1310, #1327, #1331)

9.3 v6.0.0 (2019-03-23)

features:

- TestBot: Implement inject_mocks method (#1235)

- TestBot: Add multi-line command test support (#1238)

- Added optional room arg to inroom
- Adds ability to go back to a previous room
- Pass telegram message id to the callback

fixes:

- Remove extra spaces in uptime output
- Fix/backend import error messages (#1248)
- Add docker support for installing package dependencies (#1245)
- variable name typo (#1244)
- Fix invalid variable name (#1241)
- sanitize comma quotation marks too (#1236)
- Fix missing string formatting in "Command not found" output (#1259)
- Fix webhook test to not call fixture directly
- fix: arg_botcmd decorator now can be used as plain method
- setup: removing dnspython
- pin markdown <3.0 because safe is deprecated

9.4 v6.0.0-alpha (2018-06-10)

major refactoring:

- Removed Yapsy dependency
- Replaced back Bottle and Rocket by Flask
- new Pep8 compliance
- added Python 3.7 support
- removed Python 3.5 support
- removed old compatibility cruft
- ported formats and % str ops to f-strings
- Started to add field types to improve type visibility across the codebase
- removed cross dependencies between PluginManager & RepoManager

fixes:

- Use sys.executable explicitly instead of just 'pip' (thx Bruno Oliveira)
- Pycodestyle fixes (thx Nitanshu)
- Help: don't add bot prefix to non-prefixed re cmds (#1199) (thx Robin Gloster)
- split_string_after: fix empty string handling (thx Robin Gloster)
- Escaping bug in dynamic plugins
- botmatch is now visible from the errbot module (fp to Guillaume Binet)
- flows: hint boolean was not forwarded

- Fix possible event without bot_id (#1073) (thx Roi Dayan)

- decorators were working only if kwargs were empty

- Message.clone was ignoring partial and flows

features:

- partial boolean to flag partial mesages (thx Meet Mangukiya)

- Slack: room joined callback (thx Jeremy Kenyon)

- XMPP: real_jid to get the jid the users logged in (thx Robin Gloster)

- The callback order set in the config is not globally respected

- Added a default parameter to the storage context manager

9.5 v5.2.0 (2018-04-04)

fixes:

- backup fix : SyntaxError: literal_eval on file with statements (thx Bruno Oliveira)

- plugin_manager: skip plugins not in CORE_PLUGIN entirely (thx Dylan Page)

- repository search fix (thx Sijis)

- Text: mentions in the Text backend (thx Sijis)

- Text: double @ in replies (thx Sijis)

- Slack: Support breaking messages body attachment

- Slack: Add channelname to Slackroom (thx Davis Garana Pena)

features:

- Enable split arguments on room_join so you can use '' (thx Robert Honig)

- Add support for specifying a custom log formatter (Thx Oz Linden)

- Add Sentry transport support (thx Dylan Page)

- File transfert support (send_stream_request) on the Hipchat backend (thx Brad Payne)

- Show user where they are in a flow (thx Elijah Roberts)

- Help commands are sorted alphabetically (thx Fabian Chong)

- Proxy support for Slack (thx deferato)

9.6 v5.1.3 (2017-10-15)

fixes:

- Default –init config is now compatible with Text backend requirements.

- Windows: Config directories as raw string (Thx defAnfaenger)

- Windows: Repo Manager first time update (Thx Jake Shadle)

- Slack: fix Slack identities to be hashable

- Hipchat: fix HicpChat Server XMPP namespace (Thx Antti Palsola)

- Hipchat: more aggressive cashing of user list to avoid API quota exceeds (thx Roman)

9.7 v5.1.2 (2017-08-26)

fixes:

- Text: BOT_IDENTITY to stay optional in config.py
- Hipchat: send_card fix for room name lookup (thx Jason Kincl)
- Hipchat: ACL in rooms

9.8 v5.1.1 (2017-08-12)

fixes:

- allows spaces in BOT_PREFIX.
- Text: ACLs were not working (@user vs user inconsistency).

9.9 v5.1.0 (2017-07-24)

fixes:

- allow webhook receivers on / (tx Robin Gloster)
- force utf-8 to release changes (thx Robert Krambovitis)
- don't generate an errbot section if no version is specified in plugin gen (thx Meet Mangukiya)
- callback on all unknown commands filters
- user friendly message when a room is not found
- webhook with no uri but kwargs now work as intended
- Slack: support for Enterprise Grid (thx Jasper)
- Hipchat: fix room str repr. (thx Roman)
- XMPP: fix for MUC users with @ in their names (thx Joon Guillen)
- certificate generation was failing under some conditions

features:

- Support for threaded messages (Slack initially but API is done for other backends to use)
- Text: now the text backend can emulate an inroom/inperson or asuser/asadmin behavior
- Text: autocomplete of command is now supported
- Text: multiline messages are now supported
- start_poller can now be restricted to a number of execution (thx Marek Suppa)
- recurse_check_structure back to public API (thx Alex Sheluchin)
- better flow status (thx lijah Roberts)
- !about returns a git tag instead of just 9.9.9 as version for a git checkout. (thx Sven)

- admin notifications can be set up to a set of users (thx Sijis Aviles)

- logs can be colorized with drak, light or nocolor as preference.

9.10 v5.0.1 (2017-05-08)

hotfixes for v5.0.0.

fixes: - fix crash for SUPPRESS_CMD_NOT_FOUND=True (thx Romuald Texier-Marcadé!)

breaking / API cleanups: - Missed patch for 5.0.0: now the name of a plugin is defined by its name in .plug and not its class name.

9.11 v5.0.0 (2017-04-23)

features:

- Add support for cascaded subcommands (cmd_sub1_sub2_sub3) (thx Jeremiah Lowin)

- You can now use symbolic links for your plugins

- Telegram: send_stream_request support added (thx Alexandre Manhaes Savio)

- Callback to unhandled messages (thx tamarin)

- flows: New option to disable the next step hint (thx Aviv Laufer)

- IRC: Added Notice support (bot can listen to them)

- Slack: Original slack event message is attached to Message (Thx Bryan Shelton)

- Slack: Added reaction support and Message.extras['url'] (Thx Tomer Chachamu)

- Text backend: readline support (thx Robert Coup)

- Test backend: stream requests support (thx Thomas Lee)

fixes:

- When a templated cmd crashes, it was crashing in the handling of the error.

- Slack: no more crash if a message only contains attachments

- Slack: fix for some corner case links (Thx Tomer Chachamu)

- Slack: fixed LRU for better performance on large teams

- Slack: fix for undefined key 'username' when the bot doesn't have one (thx Octavio Antonelli)

other:

- Tests: use conftest module to specify testbot fixture location (thx Pavel Savchenko)

- Python 3.6.x added to travis.

- Ported the yield tests to pytest 4.0

- Removed a deprecated dependency for the threadpool, now uses the standard one (thx Muri Nicanor)

breaking / API cleanups:

- removed deprecated presence attributes (nick and occupant)

- removed deprecated type from messages.

- utils.ValidationException has moved to errbot.ValidationException and is fully part of the API.
- {utils, errbot}.get_class_that_defined_method is now _bot.get_plugin_class_from_method
- utils.utf8 has been removed, it was a leftover for python 2 compat.
- utils.compat_str has been removed, it was a vestige for python 2 too.

9.12 v4.3.7 (2017-02-08)

fixes:

- slack: compatibility with slackclient > 1.0.5.
- render test fix (thx Sandeep Shantharam)

9.13 v4.3.6 (2017-01-28)

fixes:

- regression with Markdown 2.6.8.

9.14 v4.3.5 (2016-12-21)

fixes:

- slack: compatibility with slackclient > 1.0.2
- slack: block on reads on RTM (better response time) (Thx Tomer Chachamu)
- slack: fix link names ('')
- slack: ignore channel_topic messages (thx Mikhail Sobolev)
- slack: Match ACLs for bots on integration ID
- slack: Process messages from webhook users
- slack: don't crash when unable to look up alternate prefix
- slack: trm_read refactoring (thx Chris Niemira)
- telegram: fix telegram ID test against ACLs
- telegram: ID as strings intead of ints (thx Pmoranga)
- fixed path to the config template in the startup error message (Thx Ondrej Skopek)

9.15 v4.3.4 (2016-10-05)

features:

- Slack: Stream (files) uploads are now supported
- Hipchat: Supports for self-signed server certificates.

fixes:

- Card emulation support for links (Thx Robin Gloster)

- IRC: Character limits fix (Thx lqaz)

- Dependency check fix.

9.16 v4.3.3 (2016-09-09)

fixes:

- err references leftovers

- requirements.txt is now standard (you can use git+https:// for example)

9.17 v4.3.2 (2016-09-04)

hotfix:

- removed the hard dependency on pytest for the Text backend

9.18 v4.3.1 (2016-09-03)

features:

- now the threadpool is of size 10 by default and added a configuration.

fixes:

- fixed imporlib/use pip as process (#835) (thx Raphael Wouters)

- if pip is not found, don't crash errbot

- build_identifier to send message to IRC channels (thx mr Shu)

9.19 v4.3.0 (2016-08-10)

9.19.1 v4.3 features

- *DependsOn:* entry in .plug and *self.get_plugin(. . .)* allowing you to make a plugin dependent from another.

- New entry in config.py: PLUGINS_CALLBACK_ORDER allows you to force a callback order on your installed plugins.

- Flows can be shared by a room if you build the flow with *FlowRoot(room_flow=True)* (thx Tobias Wilken)

- New construct for persistence: *with self.mutable(key) as value:* that allows you to change by side effect value without bothering to save value back.

9.19.2 v4.3 Miscellaneous changes

- This version work only on Python 3.4+ (see 4.2 announcement)

- Presence.nick is deprecated, simply use presence.identifier.nick instead.

- Slack: Bot identity is automatically added to BOT_ALT_PREFIXES

- The version checker now reports your Python version to be sure to not upgrade Python 2 users to 4.3

- Moved testing to Tox. We used to use a custom script, this improves a lot the local testing setup etc. (Thx Pedro Rodrigues)

9.19.3 v4.3 fixes

- IRC: fixed IRC_ACL_PATTERN

- Slack: Mention callback improvements (Thx Ash Caire)

- Encoding error report was inconsistent with the value checked (Thx Steve Jarvis)

- core: better support for all the types of virtualenvs (Thx Raphael Wouters)

9.20 v4.2.2 (2016-06-24)

fixes:

- send_templated fix

- CHATROOM_RELAY fix

- Blacklisting feedback message corrected

9.21 v4.2.1 (2016-06-10)

Hotfix

- packaging failure under python2

- better README

9.22 v4.2.0 (2016-06-10)

9.22.1 v4.2 Announcement

- Bye bye Python 2 ! This 4.2 branch will be the last to support Python 2. We will maintain bug fixes on it for at least the end of 2016 so you can transition nicely, but please start now !

  Python 3 has been released 8 years ago, now all the major distributions finally have it available, the ecosystem has moved on too. This was not the case at all when we started to port Errbot to Python 3.

  This will clean up *a lot* of code with ugly *if PY2*, unicode hacks, 3to2 reverse hacks all over the place and packaging tricks. But most of all it will finally unite the Errbot ecosystem under one language and open up new possibilities as we refrained from using py3 only features.

- A clarification on Errbot's license has been accepted. The contributors never intended to have the GPL licence be enforced for external plugins. Even if it was not clear it would apply, our new licence exception makes sure it isn't. Big big thanks for the amazing turnout on this one !

9.22.2 v4.2 New features

- Errbot initial installation. The initial installation has been drastically simplified:

```
$ pip install errbot
$ mkdir errbot; cd errbot
$ errbot --init
$ errbot -T
>>>      <- You are game !!
```

Not only that but it also install a development directory in there so it now takes only seconds to have an Errbot development environment.

- Part of this change, we also made most of the config.py entries with sane defaults, a lot of those settings were not even relevant for most users.
- cards are now supported on the graphic backend with a nice rendering (errbot -G)
- Hipchat: mentions are now supported.

9.22.3 v4.2 Miscellaneous changes

- Documentation improvements
- Reorganization and rename of the startup files. Those were historically the first ones to be created and their meaning drifted over the years. We had err.py, main.py and errBot.py, it was really not clear what were their functions and why one has been violating the python module naming convention for so long :) They are now bootstrap.py (everything about configuring errbot), cli.py (everything about the errbot command line) and finally core.py (everything about the commands, and dispatching etc...).
- setup.py cleanup. The hacks in there were incorrect.

9.22.4 v4.2 fixes

- core: excpetion formatting was failing on some plugin load failures.
- core: When replacing the prefix *!* from the doctrings only real commands get replaced (thx Raphael Boidol)
- core: empty lines on plugins requirements.txt does crash errbot anymore
- core: Better error message in case of malformed .plug file
- Text: fix on build_identifier (thx Pawet Adamcak)
- Slack: several fixes for identifiers parsing, the backend is fully compliant with Errbot's contract now (thx Raphael Boidol and Samuel Loretan)
- Hipchat: fix on room occupants (thx Roman Forkosh)
- Hipchat: fix for organizations with more than 100 rooms. (thx Naman Bharadwaj)
- Hipchat: fixed a crash on build_identifier

9.23 v4.1.3 (2016-05-10)

hotfixes:

- Slack: regression on build_identifier
- Hipchat: regression on build_identifier (query for room is not supported)

9.24 v4.1.2 (2016-05-10)

fixes:

- cards for hipchat and slack were not merged.

9.25 v4.1.1 (2016-05-09)

fixes:

- Python 2.7 conversion error on err.py.

9.26 v4.1.0 (2016-05-09)

9.26.1 v4.1 features

- Conversation flows: Errbot can now keep track of conversations with its users and automate part of the interactions in a state machine manageable from chat. see the flows documentation for more information.
- Cards API: Various backends have a "canned" type of formatted response. We now support that for a better native integration with Slack and Hipchat.
- Dynamic Plugins API: Errbot has now an official API to build plugins at runtime (on the fly). see the dynamic plugins doc
- Storage command line interface: It is now possible to provision any persistent setting from the command line. It is helpful if you want to automate end to end the deployment of your chatbot. see provisioning doc

9.26.2 v4.1 Miscellaneous changes

- Now if no [python] section is set in the .plug file, we assume Python 3 instead of Python 2.
- Slack: identifier.person now gives its username instead of slack id
- IRC: Topic change callback fixed. Thx Ezequiel Brizuela.
- Text/Test: Makes the identifier behave more like a real backend.
- Text: new TEXT_DEMO_MODE that removes the logs once the chat is started: it is made for presentations / demos.
- XMPP: build_identifier can now resolve a Room (it will eventually be available on other backends)
- Graphic Test backend: renders way better the chat, TEXT_DEMO_MODE makes it full screen for your presentations.

- ACLs: We now allow a simple string as an entry with only one element.

- Unit Tests are now all pure py.test instead of a mix of (py.test, nose and unittest)

9.26.3 v4.1 fixed

- Better resillience on concurrent modifications of the commands structures.

- Allow multiline table cells. Thx Ilya Figotin.

- Plugin template was incorrectly showing how to check config. Thx Christian Weiske.

- Slack: DIVERT_TO_PRIVATE fix.

- Plugin Activate was not reporting correctly some errors.

- tar.gz packaged plugins are working again.

9.27 v4.0.3 (2016-03-17)

fixes:

- XMPP backend compatibility with python 2.7

- Telegram startup error

- daemonize regression

- UTF-8 detection

9.28 v4.0.2 (2016-03-15)

hotfixes:

- configparser needs to be pinned to a 3.5.0b2 beta

- Hipchat regression on Identifiers

- Slack: avoid URI expansion.

9.29 v4.0.1 (2016-03-14)

hotfixes:

- v4 doesn't migrate plugin repos entries from v3.

- py2 compatibility.

9.30 v4.0.0 (2016-03-13)

This is the next major release of errbot with significant changes under the hood.

9.30.1 v4.0 New features

- Storage is now implemented as a plugin as well, similar to command plugins and backends. This means you can now select different storage implementations or even write your own.

The following storage backends are currently available:

- The traditional Python *shelf* storage.

- In-memory storage for tests or ephemeral storage.

- SQL storage which supports relational databases such as MySQL, Postgres, Redshift etc.

- Firebase storage for the Google Firebase DB.

- Redis storage (thanks Sijis Aviles!) which uses the Redis in-memory data structure store.

- Unix-style glob support in *BOT_ADMINS* and *ACCESS_CONTROLS* (see the updated *config-template.py* for documentation).

- The ability to apply ACLs to all commands exposed by a plugin (see the updated *config-template.py* for documentation).

- The mention_callcack() on IRC (mr. Shu).

- A new (externally maintained) Skype backend.

- The ability to disable core plugins (such as *!help*, *!status*, etc) from loading (see *CORE_PLUGINS* in the updated *config-template.py*).

- Added a *–new-plugin* flag to *errbot* which can create an emply plugin skeleton for you.

- IPv6 configuration support on IRC (Mike Burke)

- More flexible access controls on IRC based on nickmasks (in part thanks to Marcus Carlsson). IRC users, see the new *IRC_ACL_PATTERN* in *config-template.py*.

- A new *callback_mention()* for plugins (not available on all backends).

- Admins are now notified about plugin startup errors which happen during bot startup

- The repos listed by the *!repos* command are now fetched from a public index and can be queried with *!repos query [keyword]*. Additionally, it is now possible to add your own index(es) to this list as well in case you wish to maintain a private index (special thanks to Sijis Aviles for the initial proof-of-concept implementation).

9.30.2 v4.0 fixed

- IRC backend no longer crashes on invalid UTF-8 characters but instead replaces them (mr. Shu).

- Fixed joining password-protected rooms (Mikko Lehto)

- Compatibility to API changes introduced in slackclient-1.0.0 (used by the Slack backend).

- Corrected room joining on IRC (Ezequiel Hector Brizuela).

- Fixed *"team_join event handler raised an exception"* on Slack.

- Fixed *DIVERT_TO_PRIVATE* on HipChat.

- Fixed *DIVERT_TO_PRIVATE* on Slack.

- Fixed *GROUPCHAT_NICK_PREFIXED* not prefixing the user on regular commands.

- Fixed *HIDE_RESTRICTED_ACCESS* from accidentally sending messages when issuing *!help*.

- Fixed *DIVERT_TO_PRIVATE* on IRC.

- Fixed markdown rendering breaking with *GROUPCHAT_NICK_PREFIXED* enabled.

- Fixed *AttributeError* with *AUTOINSTALL_DEPS* enabled.

- IRC backend now cleanly disconnects from IRC servers instead of just cutting the connection.

- Text mode now displays the prompt beneath the log output

- Plugins which fail to install no longer remain behind, obstructing a new installation attempt

9.30.3 v4.0 Breaking changes

- The underlying implementation of Identifiers has been drastically refactored to be more clear and correct. This makes it a lot easier to construct Identifiers and send messages to specific people or rooms.

- The file format for *–backup* and *–restore* has changed between 3.x and 4.0 On the v3.2 branch, backup can now backup using the new v4 format with *!backupv4* to make it possible to use with *–restore* on errbot 4.0.

A number of features which had previously been deprecated have now been removed. These include:

- *configure_room* and *invite_in_room* in *XMPPBackend* (use the equivalent functions on the *XMPPRoom* object instead)

- The *–xmpp*, *–hipchat*, *–slack* and *–irc* command-line options from *errbot* (set a proper *BACKEND* in *config.py* instead).

9.30.4 v 4.0 Miscellaneous changes

- Version information is now specified in plugin *.plug* files instead of in the Python class of the plugin.

- Updated *!help* output, more similar to Hubot's help output (James O'Beirne and Sijis Aviles).

- XHTML-IM output can now be enabled on XMPP again.

- New *–version* flag on *errbot* (mr. Shu).

- Made *!log tail* admin only (Nicolas Sebrecht).

- Made the version checker asynchronous, improving startup times.

- Optionally allow bot configuration from groupchat

- *Message.type* is now deprecated in favor of *Message.is_direct* and *Message.is_group*.

- Some bundled dependencies have been refactored out into external dependencies.

- Many improvements have been made to the documentation, both in docstrings internally as well as the user guide on the website at http://errbot.io.

9.30.5 Further info on identifier changes

- Person, RoomOccupant and Room are now all equal and can be used as-is to send a message to a person, a person in a Room or a Room itself.

The relationship is as follow:
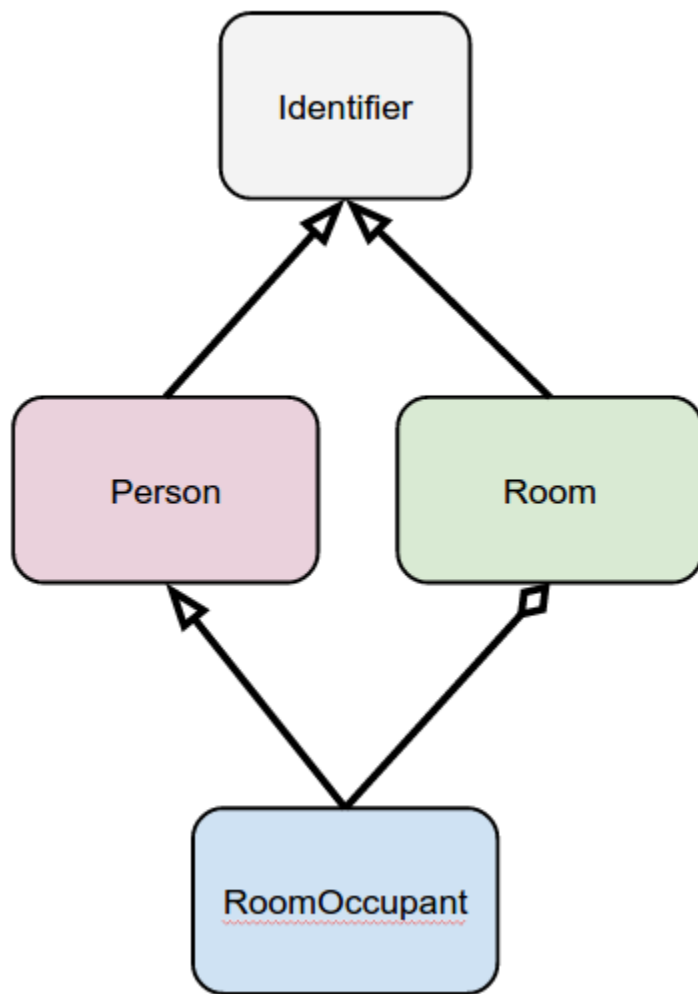
For example: A Message sent from a room will have a RoomOccupant as frm and a Room as to.

This means that you can now do things like:

- *self.send(msg.frm, "Message")*
- *self.send(self.query_room("#general"), "Hello everyone")*

License

Errbot is free software, available under the GPL-3 license. Please refer to the `full license text` for more details.

e

Index