EH Forwarder Bot Documentation

Release 2.1.1

Eana Hufwe, and the EH Forwarder Bot contributors

Feb 13, 2022

Contents

EH Forwarder Bot

An extensible message tunneling chat bot framework.

Codename **EH Forwarder Bot** (EFB) is an extensible message tunneling chat bot framework which delivers messages to and from multiple platforms and remotely control your accounts.

CHAPTER 1

Getting started

A few simple steps to get started with EFB.

1.1 Install EH Forwarder Bot

EH Forwarder Bot can be installed in the following ways:

1.1.1 Install from PyPI

`pip` will by default install the latest stable version from PyPI, but development versions are available at PyPI as well.

```
pip3 install ehforwarderbot
```

1.1.2 Install from GitHub

This will install the latest commit from GitHub. It might not be stable, so proceed with caution.

```
pip3 install git+https://github.com/ehForwarderBot/ehforwarderbot.git
```

1.1.3 Alternative installation methods

You can find a list of alternative installation methods contributed by the community in the project wiki.

For scripts, containers (e.g. Docker), etc. that may include one or more external modules, please visit the modules repository.

Note: These alternative installation methods are maintained by the community, please consult their respective author or maintainer for help related to those methods.

1.2 A stable internet connection

Since the majority of our channels are using polling for message retrieval, a stable internet connection is necessary for channels to run smoothly. An unstable connection may lead to slow response, or loss of messages.

1.3 Create local directories

EFB uses a *nix user configuration style, which is described in details in *Directories*. In short, if you are using the default configuration, you need to create `~/.ehforwarderbot`, and give read and write permission to the user running EFB.

1.4 Choose, install and enable modules

Currently, all modules that was submitted to us are recorded in the modules repository. You can choose the channels that fits your need the best.

Instructions about installing each channel is available at their respective documentations.

1.4.1 Set up with the configuration wizard

When you have successfully installed the modules of your choices, you can the use the configuration wizard which guides you to enable channels and middlewares, and continue to setup those modules if they also have provided a similar wizard.

You can start the wizard by running the following command in a compatible console or terminal emulator:

```
efb-wizard
```

If you want to start the wizard of a module for a profile individually, run:

```
efb-wizard -p <profile name> -m <module ID>
```

1.4.2 Set up manually

Alternatively, you can enable those modules manually it by listing its Channel ID in the *configuration file*. The default path is `~/.ehforwarderbot/profiles/default/config.yaml`. Please refer to *Directories* if you have configured otherwise.

Please note that although you can have more than one slaves channels running at the same time, you can only have exactly one master channels running in one profile. Meanwhile, middlewares are completely optional.

For example, to enable the following modules:

- **Master channel**
  - Demo Master (`foo.demo_master`)
- **Slave channels**
  - Demo Slave (`foo.demo_slave`)
  - Dummy Slave (`bar.dummy`)
- **Middlewares**
  - Null Middleware (`foo.null`)

`config.yaml` should have the following lines:

```
master_channel: foo.demo_master
slave_channels:
- foo.demo_slave
- bar.dummy
middlewares:
- foo.null
```

If you have enabled modules manually, you might also need configure each module manually too. Please consult the documentation of each module for instructions.

1.5 Launch EFB

```
ehforwarderbot
```

This will launch EFB directly in the current environment. The default *Profiles* is named `default`, to launch EFB in a different profile, append `--profile <profile-name>` to the command.

For more command line options, use `--help` option.

1.5.1 Use EFB in another language

EFB supports translated user interface and prompts. You can set your system language or locale environmental variables (`LANGUAGE`, `LC_ALL`, `LC_MESSAGES` or `LANG`) to one of our supported languages to switch language.

You can help to translate this project into your languages on our Crowdin page.

Note: If your are installing from source code, you will not get translations of the user interface without manual compile of message catalogs (`.mo`) prior to installation.

1.5.2 Launch EFB as a daemon process

Since version 2, EH Forwarder Bot has removed the daemon helper as it is unstable to use. We recommend you to use mature solutions for daemon management, such as systemd, upstart, or pm2.

CHAPTER 2

Configuration File

EFB has an overall configuration file to manage all enabled modules. It is located under the *directory* of current profile, and named `config.yaml`.

2.1 Syntax

The configuration file is in the YAML syntax. If you are not familiar with its syntax, please check its documentations and tutorials for details.

- The ID of the master channel enabled is under the key `master_channel`
- The ID of slave channels enabled is listed under the key `slave_channels`. It has to be a list even if just one channel is to be enabled.
- The ID of middlewares enabled are listed under the key `middlewares`. It has to be a list even if just one middleware is to be enabled. However, if you don't want to enable any middleware, just omit the section completely.

2.2 Instance ID

To have multiple accounts running simultaneously, you can appoint an instance ID to a module. Instance ID can be defined by the user, and if defined, it must has nothing other than letters, numbers and underscores, i.e. in regular expressions `[a-zA-Z0-9_]+`. When instance ID is not defined, the channel will run in the "default" instance with no instance ID.

To indicate the instance ID of an instance, append `#` following by the instance ID to the module ID. For example, slave channel `bar.dummy` running with instance ID `alice` should be written as `bar.dummy#alice`. If the channel requires configurations, it should be done in the directory with the same name (e.g. `EFB_DATA_PATH/profiles/`*`PROFILE`*`/bar.dummy#alice`), so as to isolate instances.

Please avoid having two modules with the same set of module ID and instance ID as it may leads to unexpected results.

For example, to enable the following modules:

- **Master channel**

  - Demo Master (foo.demo_master)
- **Slave channels**
  - Demo Slave (foo.demo_slave)
  - Dummy Slave (bar.dummy)
  - Dummy Slave (bar.dummy) at alt instance
- **Middlewares**
  - Message Archiver (foo.msg_archiver)
  - Null Middleware (foo.null)

config.yaml should have the following lines:

```
master_channel: foo.demo_master
slave_channels:
- foo.demo_slave
- bar.dummy
- bar.dummy#alt
middlewares:
- foo.msg_archiver
- foo.null
```

2.3 Granulated logging control

If you have special needs on processing and controlling the log produced by the framework and running modules, you can use specify the log configuration with Python's configuration dictionary schema under section logging.

An example of logging control settings:

```
logging:
    version: 1
    disable_existing_loggers: false
    formatters:
        standard:
            format: '%(asctime)s [%(levelname)s] %(name)s: %(message)s'
    handlers:
        default:
            level: INFO
            formatter: standard
            class: logging.StreamHandler
            stream: ext://sys.stdout
    loggers:
          '':
                handlers: [default,]
                level: INFO
                propagate: true
          AliceIRCChannel:
                handlers: [default, ]
                level: WARN
                propagate: false
```

CHAPTER 3

Launch the framework

EH Forwarder Bot offered 2 ways to launch the framework:

- `ehforwarderbot`
- `python3 -m ehforwarderbot`

Both commands are exactly the same thing, accept the same flags, run the same code. The latter is only a backup in case the former does not work.

3.1 Options

- `-h`, `--help`: Show help message
- `-p PROFILE`, `--profile PROFILE`: Switch *profile*

  From version 2, EFB supports running different instances under the same user, identified by their profiles. The default profile is named `default`.
- `-V`, `--version`: Print version information

  This shows version number of Python you are using, the EFB framework, and all channels and middlewares enabled.
- `-v`, `--verbose`: Print verbose log

  This option enables verbose log of EFB and all enabled modules. This, together with `--version`, is particularly useful in debugging and issue reporting.
- `--trace-threads`: Trace hanging threads

  This option is useful to identify source of the issue when you encounter situations where you had to force quit EFB. When this option is enabled, once the first stop signal (`SIGINT` or `SIGTERM`) is sent, threads that are *asleep* will be identified and reported every 10 seconds, until a second stop signal is seen.

  In order to use this option, you need to install extra Python dependencies using the following command.

```
pip3 install 'ehforwarderbot[trace]'
```

3.2 Quitting EFB

If you started EFB in a shell, you can simply press `Control-c` to trigger the quit process. Otherwise, ask your service manager to issue a `SIGTERM` for a graceful exit. The exit process may take a few second to complete.

Important: It is important for you to issue a graceful termination signal (e.g. `SIGTERM`), and **NOT** to use `SIGKILL`. Otherwise you may face the risk of losing data and breaking programs.

If you have encountered any issue quitting EFB, press `Control-c` for 5 times consecutively to trigger a force quit. In case you have frequently encountered situations where you had to force quit EFB, there might be a bug with EFB or any modules enabled. You may want to use the `--trace-threads` option described above to identify the source of issue, and report this to relevant developers.

CHAPTER 4

Directories

Since EH Forwarder Bot 2.0, most modules should be installed with the Python Package Manager `pip`, while configurations and data are stored in the "EFB data directory".

By default, the data directory is user specific, located in the user's home directory, `~/.ehforwarderbot`. This can be overridden with the environment variable `EFB_DATA_PATH`. This path defined here should be an **absolute path**.

4.1 Directory structure

Using the default configuration as an example, this section will introduce about the structure of EFB data directory.

```
./ehforwarderbot                or $EFB_DATA_PATH
|- profiles
|  |- default                   The default profile.
|  |  |- config.yaml            Main configuration file.
|  |  |- dummy_ch_master        Directory for data of the channel
|  |  |  |- config.yaml         Config file of the channel. (example)
|  |  |  |- ...
|  |  |- random_ch_slave
|  |  |  |- ...
|  |- profile2                  Alternative profile
|  |  |- config.yaml
|  |  |- ...
|  |- ...
|- modules                      Place for source code of your own channels/middlewares
|  |- random_ch_mod_slave       Channels here have a higher priority while importing
|  |  |- __init__.py
|  |  |- ...
```

CHAPTER 5

Profiles

Starting from EFB version 2, profiles are introduced to allow users in need to run multiple EFB instances simultaneously without affecting each other.

Each profile has its own set of configuration files a set of channels that share the same code, but has different data files, so that they can run on their own.

The default profile name is called `default`. To switch to a different profile, specify the profile name in `--profile` flag while starting EFB.

5.1 Start a new profile

To create a new profile, you need to create a directory in the *EFB_DATA_PATH*`/profiles`, and create a new configuration file as described in chapter *Getting started*.

When everything is configured properly, you are good to go.

CHAPTER 6

Support

6.1 Bug reports and feature requests

See *contribution guideline* for details.

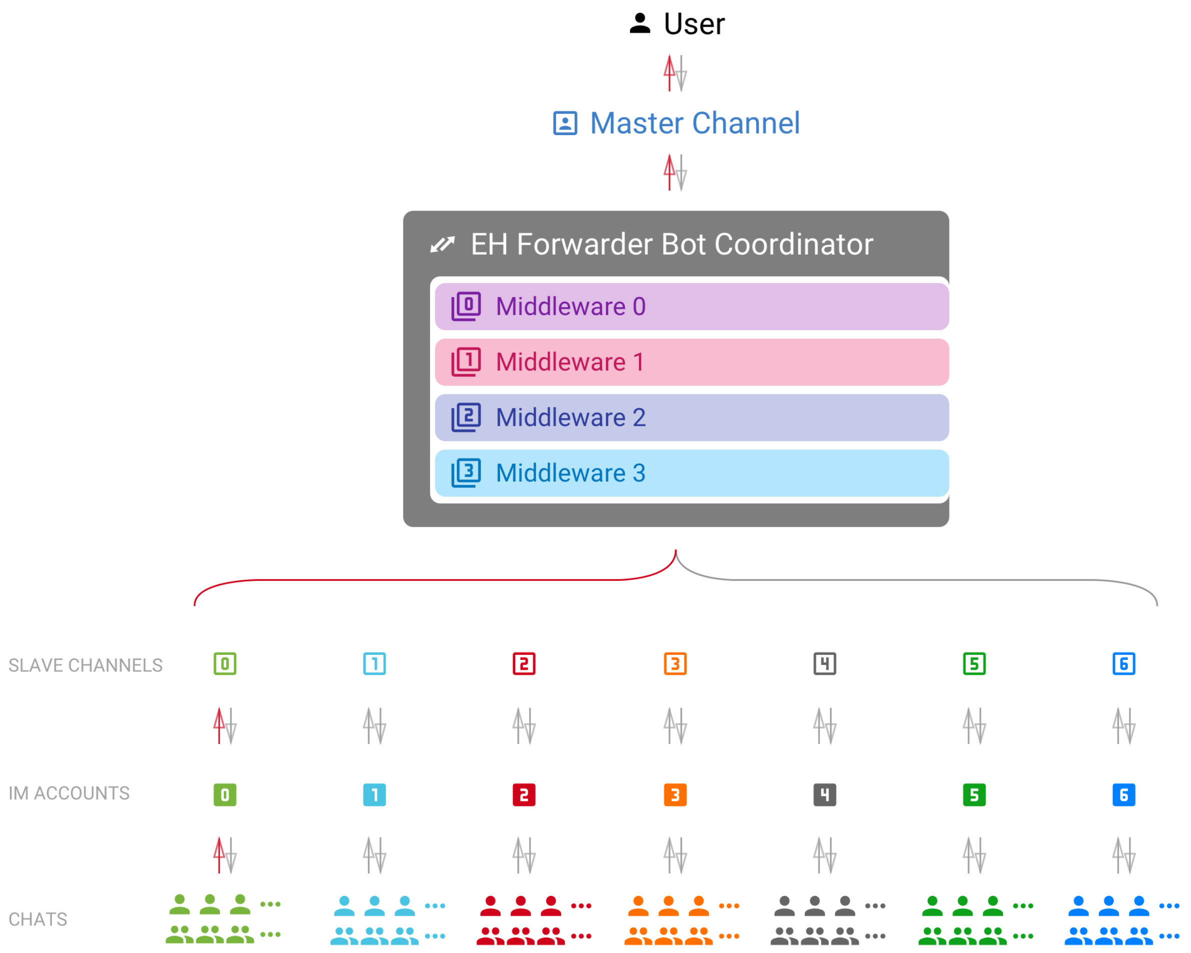
6.2 Questions about development and usage

If you have any question about developing a module for EFB, or about usages, you can always visit our GitHub Discussions forum or join our Telegram Group for help.

CHAPTER 7

Walk-through — How EFB works

EH Forwarder Bot is an extensible framework that allows user to control and manage accounts from different chat platforms in a unified interface. It consists of 4 parts: a Master Channel, some Slave Channels, some Middlewares and a Coordinator.



master channel The channel that directly interact with *the User*. It is guaranteed to have one and only one master channel in an EFB session.

slave channel The channel that delivers messages to and from their relative platform. There is at lease one slave channel in an EFB session.

coordinator Component of the framework that maintains the instances of channels, and delivers messages between channels.

middleware Module that processes messages and statuses delivered between channels, and make modifications where needed.

7.1 Concepts to know

module A common term that refers to both channels and middlewares.

the User

the User Themself This term[1] can refer to the user of the current instance of EH Forwarder Bot, operating the master channel, and the account of an IM platform logged in by a slave channel.

chat A place where conversations happen, it can be either a *private chat*, a *group chat*, or a *system chat*.

private chat A conversation with a single person on the IM platform. Messages from a private conversation shall only has an author of *the User Themself*, the other person, or a "system member".

For platforms that support bot or something similar, they would also be considered as a "user", unless messages in such chat can be sent from any user other than the bot.

For chats that *the User* receive messages, but cannot send message to, it should also be considered as a private chat, only to raise an exception when messages was trying to send to the chat.

group chat A chat that involves more than two members. A group chat MUST provide a list of members that is involved in the conversation.

system chat A chat that is a part of the system. Usually used for chats that are either a part of the IM platform, the *slave channel*, or a *middleware*. *Slave channel*s can use this chat type to send system message and notifications to the master channel.

chat member A participant of a chat. It can be *the User Themself*, another person or bot in the chat, or a virtual one created by the IM platform, the *slave channel*, or a *middleware*.

message Messages are delivered strictly between the master channel and a slave channel. It usually carries an information of a certain type.

Each message should at least have a unique ID that is distinct within the slave channel related to it. Any edited message should be able to be identified with the same unique ID.

status Information that is not formatted into a message. Usually includes updates of chats and members of chats, and removal of messages.

7.2 Slave Channels

The job of slave channels is relatively simple.

1. Deliver messages to and from the master channel.
2. Maintains a list of all available chats, and group members.
3. Monitors changes of chats and notify the master channel.

Features that does not fit into the standard EFB Slave Channel model can be offered as *Additional features*.

[1] "Themself" here is used as a derived form of a gender-neutral singular third-person pronoun.

7.3 Master Channels

Master channels is relatively more complicated and also more flexible. As it directly faces the User, its user interface should be user-friendly, or at least friendly to the targeted users.

The job of the master channel includes:

1. Receive, process and display messages from slave channels.
2. Display a full list of chats from all slave channels.
3. Offer an interface for the User to use “extra functions” from slave channels.
4. Process updates from slave channels.
5. Provide a user-friendly interface as far as possible.

7.4 Middlewares

Middlewares can monitor and make changes to or nullify messages and statuses delivered between channels. Middlewares are executed in order of registration, one after another. A middleware will always receive the messages processed by the preceding middleware if available. Once a middleware nullify a message or status, the message will not be processed and delivered any further.

CHAPTER 8

Development guide

This section includes guides on how to develop channels and middlewares for EH Forwarder Bot.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [**RFC 2119**] [**RFC 8174**] when, and only when, they appear in all capitals, as shown here.

8.1 Slave channels

Slave channel is more like a wrap over an API of IM, it encloses messages from the IM into appropriate objects and deliver it to the master channel.

Although we suggest that slave channel should match with an IM platform, but you may try to model it for anything that can deliver information as messages, and has a limited list of end-points to deliver messages to and from as chats.

In most of the cases, slave channels SHOULD be identified as one single user from the IM platform, instead of a bot. You should only use a bot for slave channels when:

- the IM platform puts no difference between a user and a bot, or
- bots on the IM platform can do exactly same things, if not more, as a user, and bots can be created easier than user account.

8.1.1 Additional features

Slave channels can offer more functions than what EFB requires, such as creating groups, search for friends, etc, via *additional features*.

Such features are accessed by the user in a CLI-like style. An "additional feature" method MUST only take one string parameter aside from `self`, and wrap it with *`extra()`* decorator. The `extra` decorator takes 2 arguments: `name` – a short name of the feature, and `desc` – a description of the feature with its usage.

`desc` SHOULD describe what the feature does and how to use it. It's more like the help text for an CLI program. Since method of invoking the feature depends on the implementation of the master channel, you SHOULD use `"{function_name}"` as its name in `desc`, and master channel will replace it with respective name depend on their implementation.

The method MUST in the end return a string, which will be shown to the user as its result, or `None` to notify the master channel there will be further interaction happen. Depending on the functionality of the feature, it may be just a simple success message, or a long chunk of results.

The callable MUST NOT raise any exception to its caller. Any exceptions occurred within should be `expect`ed and processed.

Callable name of such methods has a more strict standard than a normal Python 3 identifier name, for compatibility reason. An additional feature callable name MUST:

- be case sensitive
- include only upper and lower-case letters, digits, and underscore.
- does not start with a digit.
- be in a length between 1 and 20 inclusive
- *be as short and concise as possible, but keep understandable*

It can be expressed in a regular expression as:

```
^[A-Za-z][A-Za-z0-9_]{0,19}$
```

An example is as follows:

```
@extra(name="Echo",
       desc="Return back the same string from input.\n"
            "Usage:\n"
            "    {function_name} text")
def echo(self, arguments: str = "") -> str:
    return arguments
```

8.1.2 Message commands

Message commands are usually sent by slave channels so that users can respond to certain messages that has specific required actions.

Possible cases when message commands could be useful:

- Add as friends when a contact card is received.
- Accept or decline when a friend request is received.
- Vote to a voting message.

A message can be attached with a `list` of commands, in which each of them has:

- a human-friendly name,
- a callable name,
- a `list` of positional arguments (`*args`), and
- a `dict` of keyword arguments (`**kwargs`)

When the User clicked the button, the corresponding method of your channel will be called with provided arguments.

Note that all such methods MUST return a `str` as a respond to the action from user, and they MUST NOT raise any exception to its caller. Any exceptions occurred within MUST be `expect`ed and processed.

8.1.3 Message delivery

Slave channels SHOULD deliver all messages that the IM provides, including what the User sent outside of this channel. But it SHOULD NOT deliver message sent from the master channel again back to the master channel as a new message.

8.1.4 Implementation details

See *`SlaveChannel`*.

8.2 Master channels

Master channels are the interface that directly or indirectly interact with the user. Despite the first master channel of EFB (EFB Telegram Master) is written in a form of Telegram Bot, master channels can be written in many forms, such as:

- A web app
- A server that expose APIs to dedicated desktop and mobile clients
- A chat bot on an existing IM
- A server that compiles with a generic IM Protocol
- A CLI client
- Anything else you can think of…

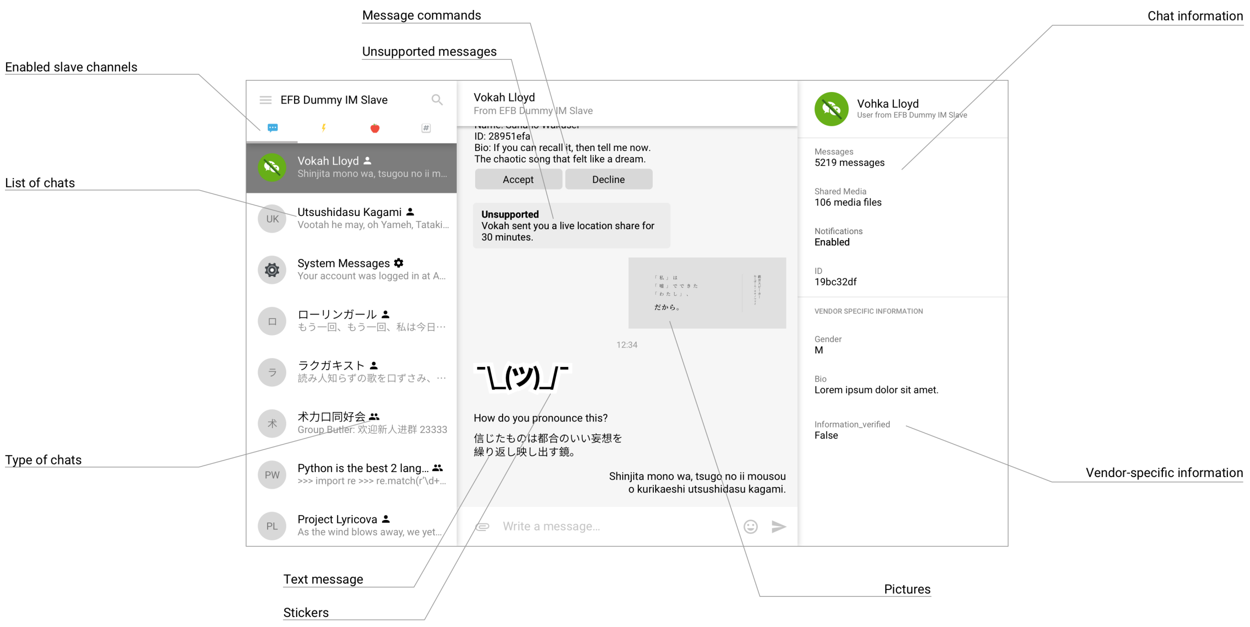
8.2.1 Design guideline

When the master channel is implemented on an existing protocol or platform, as far as possible, while considering the user experience, a master channel SHOULD:

- maintain one conversation thread per chat, indicating its name, source channel and type;
- support all, if not most, types of messages defined in the framework, process and deliver messages between the user and slave channels;
- support all, if not most, features of messages, including: targeted message reply, chat substitution in text (usually used in @ references), commands, etc. Master channel SHOULD be able to process incoming messages with such features, and send messages with such features to slave channels if applicable;
- be able to invoke and process "additional features" offered by slave channels.

Optionally, a master channel can also support / identify vendor-specified information from certain slave channels.

Depends on your implementation, a master channel may probably needs to maintain a list of chats and messages, for presentation or other purposes.



Shinjita mono wa, tsugou no ii m...
Vootah he may, oh Yameh, Tataki...
Your account was logged in at A...
もう一回、もう一回、私は今日…
読み人知らずの歌を口ずさみ、…
Group Butler: 欢迎新人进群 23333
>>> import re >>> re.match(r'\d+...
As the wind blows away, we yet...

ID: 28951efa
Bio: If you can recall it, then tell me now.
The chaotic song that felt like a dream.

Vokah sent you a live location share for 30 minutes.

¯_(ツ)_/¯

How do you pronounce this?

信じたものは都合のいい妄想を
繰り返し映し出す鏡。

Shinjita mono wa, tsugo no ii mousou
o kurikaeshi utsushidasu kagami.

5219 messages
106 media files
Enabled
19bc32df
M
Lorem ipsum dolor sit amet.
False

Figure1: An example of an ideal design of a master channel, inspired by Telegram Desktop

8.2.2 Message delivery

Note that sometimes the User may send messages outside of this EFB session, so that slave channels MAY provide a message with its author in the "self" type.

8.2.3 Implementation details

See `MasterChannel`.

8.3 Middlewares

Middlewares works in between the master channel and slave channels, they look through messages and statuses delivered between channels, passing them on, make changes or discarding them, one after another.

Like channels, middlewares will also each have an instance per EFB session, managed by the coordinator. However, they don't have centrally polling threads, which means if a middleware wants to have a polling thread or something similar running in the background, it has to stop the thread using Python's `atexit` or otherwise.

8.3.1 Message and Status Processing

Each middleware by default has 2 methods, `process_message()` which processes message objects, and `process_status()` which processes status objects. If they are not overridden, they will not touch on the object and pass it on as is.

To modify an object, just override the relative method and make changes to it. To discard an object, simply return `None`. When an object is discarded, it will not be passed further to other middlewares or channels, which means a middleware or a channel will never receive a `None` message or status.

8.3.2 Other Usages

Having rather few limitation compare to channels, middlewares are rather easy to write, which allows it to do more than just intercept messages and statuses.

Some ideas:

- Periodic broadcast to master / slave channels
- Integration with chat bots
- Automated operations on vendor-specific commands / additional features
- Share user session from slave channel with other programs
- etc…

8.3.3 Implementation details

See `Middleware`.

8.4 Lifecycle

This section talks about the lifecycle of an EFB instance, and that of a message / status.

8.4.1 Lifecycle of an EFB instance

The diagram below outlines the lifecycle of an EFB instance, and how channels and middlewares are involved in it.

8.4.2 Lifecycle of a message

The diagram below outlines the lifecycle of a message sending from a channel, going through all middlewares, sent to the destination channel, and returned back to the sending channel.

Status objects processed in the same way.

8.5 Media processing

8.5.1 Choosing media formats

Both Master and Slave channel can take their charges to convert media files they send or receive. In general: **if a media file received from remote server is not a common format, convert it before deliver it on; if a media file sent to remote server requires to be in a specific format, it should be converted before sending out**. Nevertheless, this is only a guideline on channels' responsibility regarding media processing, and everyone has their own opinion on what is a common format / encoding. Therefore we only recommend this behaviour, but do not force in our code. This is to say that, you still have to take care of the accepted type of media encoding of your corresponding method of presentation, and convert and/or fallback to different type of representation if necessary. After all, the delivery of information is more important.

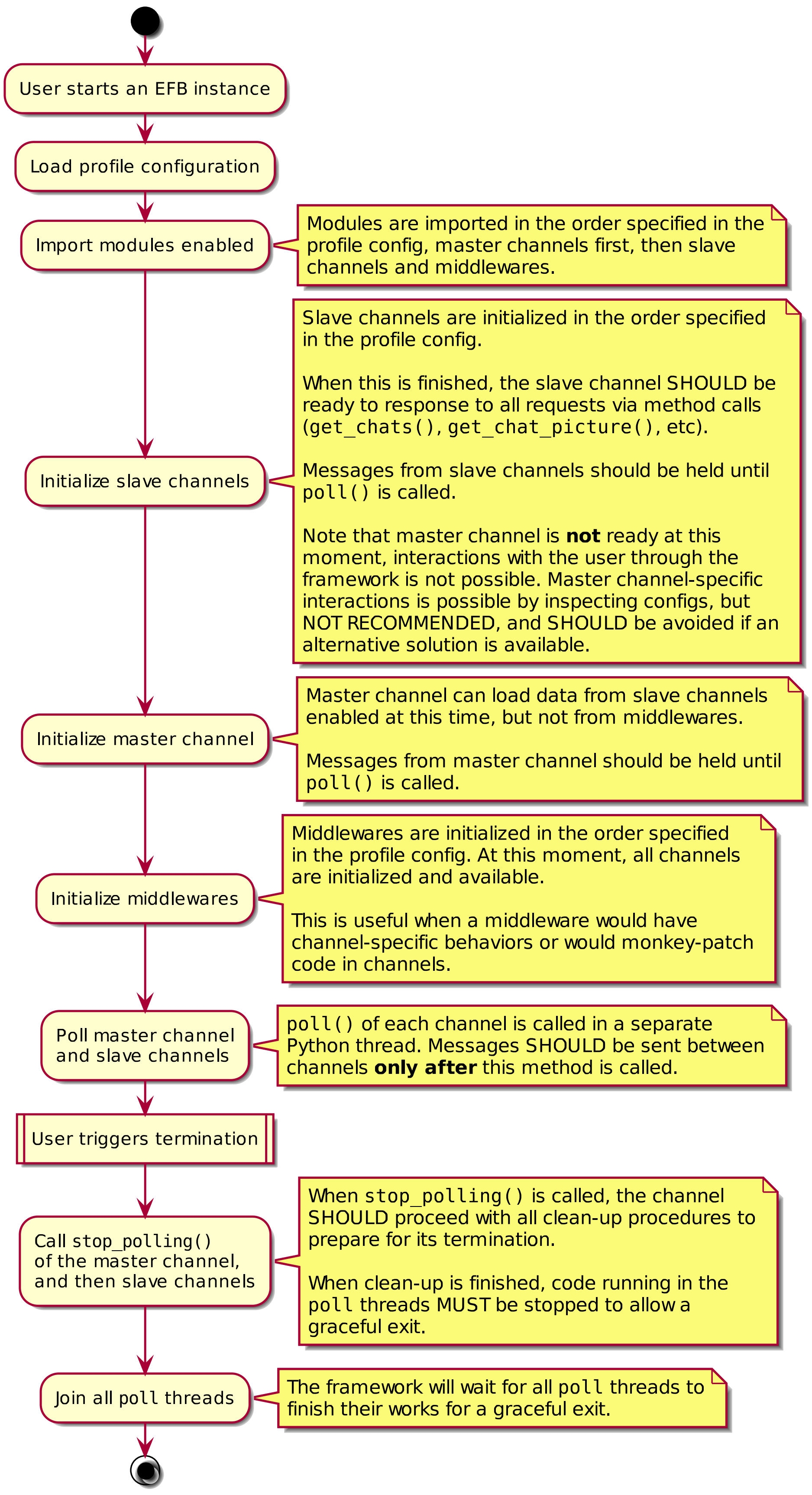

User starts an EFB instance
Load profile configuration
Import modules enabled
Modules are imported in the order specified in the profile config, master channels first, then slave channels and middlewares.
Initialize slave channels
Slave channels are initialized in the order specified in the profile config.
When this is finished, the slave channel SHOULD be ready to response to all requests via method calls (get_chats(), get_chat_picture(), etc).
Messages from slave channels should be held until poll() is called.
Note that master channel is **not** ready at this moment, interactions with the user through the framework is not possible. Master channel-specific interactions is possible by inspecting configs, but NOT RECOMMENDED, and SHOULD be avoided if an alternative solution is available.
Initialize master channel
Master channel can load data from slave channels enabled at this time, but not from middlewares.
Messages from master channel should be held until poll() is called.
Initialize middlewares
Middlewares are initialized in the order specified in the profile config. At this moment, all channels are initialized and available.
This is useful when a middleware would have channel-specific behaviors or would monkey-patch code in channels.
Poll master channel and slave channels
poll() of each channel is called in a separate Python thread. Messages SHOULD be sent between channels **only after** this method is called.
User triggers termination
Call stop_polling() of the master channel, and then slave channels
When stop_polling() is called, the channel SHOULD proceed with all clean-up procedures to prepare for its termination.
When clean-up is finished, code running in the poll threads MUST be stopped to allow a graceful exit.
Join all poll threads
The framework will wait for all poll threads to finish their works for a graceful exit.

Figure2: Lifecycle of an EFB instance

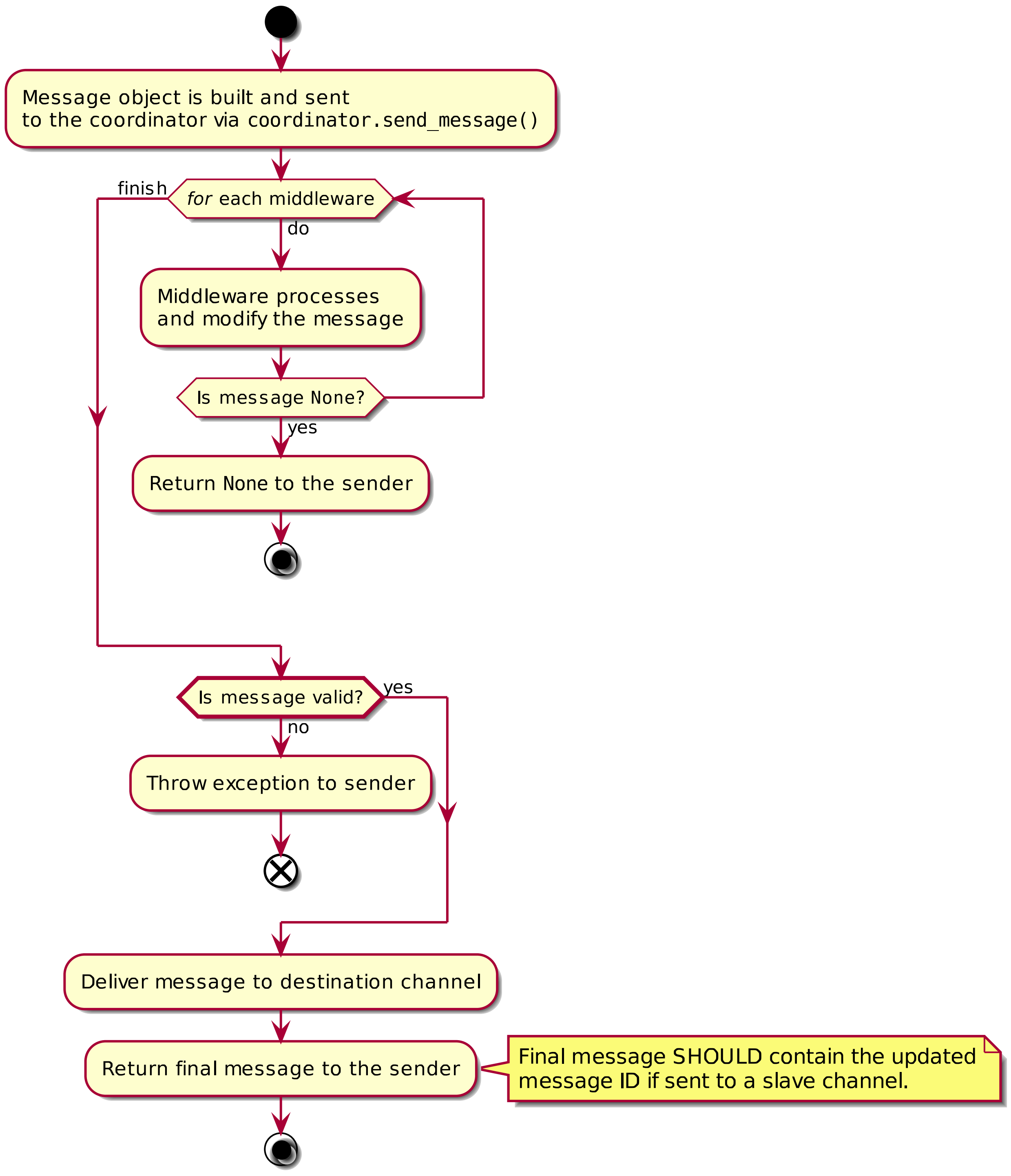

Message object is built and sent
to the coordinator via coordinator.send_message()
for each middleware
finish
do
Middleware processes
and modify the message
Is message None?
yes
Return None to the sender
Is message valid?
yes
no
Throw exception to sender
Deliver message to destination channel
Return final message to the sender
Final message SHOULD contain the updated
message ID if sent to a slave channel.

Figure3: Lifecycle of a message

8.5.2 Media encoders

Similarly, we will also not put a strict limit on this as well, but just a recommendation. As you might have already know, there are few mature pure Python media processing libraries, most of them will more or less requires internal or external binary dependencies.

We try to aim to use as few different libraries as we can, as more library to install means more space, install time, and complexity. While processing media files, we recommend to use the following libraries if possible:

- Pillow
- FFmpeg

8.5.3 Files in messages

When a file sent out from a channel, it MUST be open, and sought back to 0 (`file.seek(0)`) before sending.

Files sent MUST be able to be located somewhere in the file system, and SHOULD with a appropriate extension name, but not required. All files MUST also have its MIME type specified in the message object. If the channel is not sure about the correct MIME type, it can try to guess with `libmagic`, or fallback to `application/octet-stream`. Always try the best to provide the most suitable MIME type when sending.

For such files, we use `close` to signify the end of its lifecycle. If the file is not required by the sender's channel anymore, it can be safely discarded.

Generally, `tempfile.NamedTemporaryFile` should work for ordinary cases.

8.6 Configurations and storage

8.6.1 Configurations and Permanent Storage

As described in *Directories*, each module has been allocated with a folder per profile for configurations and other storage. The path can be obtained using `get_data_path()` with your module ID. All such storage is specific to only one profile.

For configurations, we recommend to use `<module_data_path>/config.yaml`. Similarly, we prepared `get_config_path()` to get the path for default config file. Again, you are not forced to use this name or YAML as the format of your config file.

Usually in the storage folder lives:

- Configuration files
- User credentials / Session storage
- Databases

8.6.2 Temporary Storage

While processing multimedia messages, we inevitably need to store certain files temporarily, either within the channel or across channels. Usually, temporary files can be handled with Python's `tempfile` library.

8.6.3 Wizard

If your module requires relatively complicated configuration, it would be helpful to provide users with a wizard to *check prerequisites of your module* and *guide them to setup your module for use*.

From version 2, EFB introduced a centralised wizard program to allow users to enable or disable modules in a text-based user interface (TUI). If you want to include your wizard program as a part of the wizard, you can include a new entry point in your `setup.py` with Setuptools' Entry Point feature.

The group for wizard program is `ehforwarderbot.wizard`, and the entry point function MUST accept 2 positional arguments: profile ID and instance ID.

Example

`setup.py` script

```
setup(
    # ...
    entry_points={
        "ehforwarderbot.wizard": ['alice.irc = efb_irc_slave.wizard:main']
    },
    # ...
)
```

`.egg-info/entry_points.txt`

```
[ehforwarderbot.wizard]
alice.irc = efb_irc_slave.wizard:main
```

`efb_irc_slave/wizard.py`

```
# ...

def main(profile, instance):
    print("Welcome to the setup wizard of my channel.")
    print("You are setting up this channel in profile "
          "'{0}' and instance '{1}'.".format(profile, instance))
    print("Press ENTER/RETURN to continue.")
    input()

    step1()

    # ...
```

8.7 Packaging and Publish

8.7.1 Publish your module on PyPI

Publish modules on PyPI is one of the easiest way for users to install your package. Please refer to related documentation and tutorials about PyPI and pip for publishing packages.

For EFB modules, the package is RECOMMENDED to have a name starts with `efb-`, or in the format of `efb-platform-type`, e.g. `efb-irc-slave` or `efb-wechat-mp-filter-middleware`. If there is a collision of name, you MAY adjust the package name accordingly while keeping the package name starting with `efb-`.

When you are ready, you may also want to add your module to the Modules Repository of EFB.

8.7.2 Module discovery

EH Forwarder Bot uses Setuptools' Entry Point feature to discover and manage channels and middlewares. In your `setup.py` script or `.egg-info/entry_points.txt`, specify the group and object as follows:

- Group for master channels: `ehforwarderbot.master`
- Group for slave channels: `ehforwarderbot.slave`
- Group for middlewares: `ehforwarderbot.middleware`

Convention for object names is `<author>.<platform>`, e.g. `alice.irc`. This MUST also be your module's ID.

Object reference MUST point to your module's class, which is a subclass of either *`Channel`* or *`Middleware`*.

8.7.3 Example

`setup.py` script

```
setup(
    # ...
    entry_points={
        "ehforwarderbot.slave": ['alice.irc = efb_irc_slave:IRCChannel']
    },
    # ...
)
```

`.egg-info/entry_points.txt`

```
[ehforwarderbot.slave]
alice.irc = efb_irc_slave:IRCChannel
```

8.7.4 Private modules

If you want to extend from, or make changes on existing modules for your own use, you can have your modules in the private modules *directory*.

For such modules, your channel ID MUST be the fully-qualified name of the class. For example, if your class is located at `<EFB_BASE_PATH>/modules/bob_irc_mod/__init__.py:IRCChannel`, the channel MUST have ID `bob_ric_mod.IRCChannel` for the framework to recognise it.

8.8 Miscellaneous

8.8.1 Logging

In complex modules, you should have detailed logs in DEBUG level and optionally INFO level. All your log handlers SHOULD follow that of the root logger, which is controlled by the framework. This could be helpful when for you to locate issues reported by users.

8.8.2 Vendor-specifics

If you are going to include vendor specific information in messages and/or chats, please make your effort to document them in your README or documentation, so that other developers can refer to it when adapting your module.

8.8.3 Threading

All channels are RECOMMENDED a separate thread while processing a new message, so as to prevent unexpectedly long thread blocking.

We are also considering to move completely to asynchronous programming when most channels are ready for the change.

8.8.4 Static type checking

EH Forwarder Bot is fully labeled in the Python 3 type hint notations. Since sometimes maintaining a module with high complexity could be difficult, we RECOMMEND you to type your module too and use tools like mypy to check your code statically.

CHAPTER 9

How to contribute

First of all, thanks for taking your time to contribute!

Please note that only questions on the framework will be answered here. For issue related with any channels, please contact their respective authors or post in their corresponding repositories.

Here is a simple guide on how you can file in an issue, or submit a pull request that is useful and effective.

If you need help, or want to talk to the authors, feel free to visit our GitHub Discussions forum, or chat with us at our Telegram support group.

Before you ask a question, please read and follow this guide as far as possible. Without doing so might lead to unfriendly or no response from the community, although we try to refrain from doing so.

9.1 Reporting bugs

9.1.1 Before submitting a bug report

- Please ensure if your issue is about the framework itself, not about any module. Reports about modules should go to their respective issue trackers.
- Read through the documentation to see if it has covered your question.
- Check the current issue list to see if it's been reported.

9.1.2 How Do I Submit A (Good) Bug Report?

- **Use a clear and descriptive title** for the issue to identify the problem.
- **Describe the exact steps which reproduce the problem** in as many details as possible.
- **Provide specific examples to demonstrate the steps.**
- **Describe the behavior you observed after following the steps** and point out what exactly is the problem with that behavior.
- **Explain which behavior you expected to see instead and why.**
- **If the problem wasn't triggered by a specific action**, describe what you were doing before the problem happened and share more information using the guidelines below.
- **Provide log related to the issue.** Use the verbose flag to start the logging process, and submit the entire log from the first step you performed.

Provide more context by answering these questions:

- **Did the problem start happening recently** (e.g. after updating to the latest version) or was this always a problem?
- **Can you reliably reproduce the issue?** If not, provide details about how often the problem happens and under which conditions it normally happens.

Include details about your configuration and environment:

- **What version of EFB are you using?** You can get the version by using the flag `--version`.
- **What's the name and version of the OS you're using?**

Attention: When submitting your log, please remember to hide your private information.

9.2 Suggesting enhancements

If you have any suggestions, feel free to raise it up in the issue list. Please try to provide as much details as you can, that includes:

- **Use a clear and descriptive title** for the issue to identify the suggestion.
- **Give details on how the enhancement behave**.
- **Provide specific examples to demonstrate the abstraction**.
- The enhancement to the framework must be applicable to considerably many IM platforms, not just for a single IM. Suggestions for a specific IM should be made to their relative channel.

  Adapted from Atom contribution guide by GitHub Inc.

9.3 Pull requests

When you have done some changes and want to submit it to us, fork it to your account and submit a GitHub pull request. Please write a detailed description for your pull request on:

- **What changes have you made?**
- **What problem have you solved?**
- **Which issue have you addressed** if applicable.

Always write a clear log message for your commits. One-line messages are fine for small changes, but bigger changes needs a detailed description after the one-liner.

> Adapted from OpenGovernment contribution guide by Participatory Politics Foundation

CHAPTER 10

API documentations

This section contains documentations for the current API of EH Forwarder Bot.

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “NOT RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in BCP 14 [**RFC 2119**] [**RFC 8174**] when, and only when, they appear in all capitals, as shown here.

10.1 Channel

class `ehforwarderbot.channel.`**`Channel`**(*instance_id=None*)

The abstract channel class.

`channel_name`

A human-friendly name of the channel.

Type str

`channel_emoji`

Emoji icon of the channel. Recommended to use a visually-length-one (i.e. a single grapheme cluster) emoji or other symbol that represents the channel best.

Type str

`channel_id`

Unique identifier of the channel. Convention of IDs is specified in *Packaging and Publish*. This ID will be appended with its instance ID when available.

Type *`ModuleID`* (str)

`instance_id`

The instance ID if available.

Type str

`__init__`(*instance_id=None*)

Initialize the channel. Inherited initializer MUST call the “super init” method at the beginning.

Parameters **instance_id** (`Optional[NewType()(InstanceID, str)]`) – Instance ID of the channel.

get_message_by_id(*chat*, *msg_id*)
Get message entity by its ID. Applicable to both master channels and slave channels. Return `None` when message not found.

Override this method and raise *EFBOperationNotSupported* if it is not feasible to perform this for your platform.

Parameters

- **chat** (*Chat*) – Chat in slave channel / middleware.
- **msg_id** (`NewType()(MessageID, str)`) – ID of message from the chat in slave channel / middleware.

Return type `Optional[Message]`

abstract poll()
Method to poll for messages. This method is called when the framework is initialized. This method SHOULD be blocking.

abstract send_message(*msg*)
Process a message that is sent to, or edited in this channel.

Notes

Master channel MUST take care of the returned object that contains the updated message ID. Depends on the implementation of slave channels, the message ID MAY change even after being edited. The old message ID MAY be disregarded for the new one.

Parameters **msg** (*Message*) – Message object to be processed.

Returns The same message object. Message ID of the object MAY be changed by the slave channel once sent. This can happen even when the message sent is an edited message.

Return type *Message*

Raises

- ***EFBChatNotFound*** – Raised when a chat required is not found.
- ***EFBMessageTypeNotSupported*** – Raised when the message type sent is not supported by the channel.
- ***EFBOperationNotSupported*** – Raised when an message edit request is sent, but not supported by the channel.
- ***EFBMessageNotFound*** – Raised when an existing message indicated is not found. E.g.: The message to be edited, the message referred in the `msg.target` attribute.
- ***EFBMessageError*** – Raised when other error occurred while sending or editing the message.

abstract send_status(*status*)
Process a status that is sent to this channel.

Parameters **status** (*Status*) – the status object.

Raises

- ***EFBChatNotFound*** – Raised when a chat required is not found.
- ***EFBMessageNotFound*** – Raised when an existing message indicated is not found. E.g.: The message to be removed.
- ***EFBOperationNotSupported*** – Raised when the channel does not support message removal.
- ***EFBMessageError*** – Raised when other error occurred while removing the message.

Note: Exceptions SHOULD NOT be raised from this method by master channels as it would be hard for a slave channel to process the exception.

This method is not applicable to Slave Channels.

stop_polling()

When EFB framework is asked to stop gracefully, this method is called to each channel object to stop all processes in the channel, save all status if necessary, and terminate polling.

When the channel is ready to stop, the polling function MUST stop blocking. EFB framework will quit completely when all polling threads end.

class `ehforwarderbot.channel.`**MasterChannel**(*instance_id=None*)

The abstract master channel class. All master channels MUST inherit this class.

class `ehforwarderbot.channel.`**SlaveChannel**(*instance_id=None*)

The abstract slave channel class. All slave channels MUST inherit this class.

supported_message_types

Types of messages that the slave channel accepts as incoming messages. Master channels may use this value to decide what type of messages to send to your slave channel.

Leaving this empty may cause the master channel to refuse sending anything to your slave channel.

Type Set[*MsgType*]

suggested_reactions

A list of suggested reactions to be applied to messages.

Reactions SHOULD be ordered in a meaningful way, e.g., the order used by the IM platform, or frequency of usage. Note that it is not necessary to list all suggested reactions if that is too long, or not feasible.

Set to `None` when it is known that no reaction is supported to ANY message in the channel. Set to empty list when it is not feasible to provide a list of suggested reactions, for example, the list of reactions is different for each chat or message.

Type Optional[Sequence[str]]

abstract get_chat(*chat_uid*)

Get the chat object from a slave channel.

Parameters **chat_uid** (`NewType()(ChatID, str)`) – ID of the chat.

Returns The chat found.

Return type .Chat

Raises ***EFBChatNotFound*** – Raised when a chat required is not found.

abstract get_chat_picture(*chat*)

Get the profile picture of a chat. Profile picture is also referred as profile photo, avatar, “head image” sometimes.

Parameters **chat** (*.Chat*) – Chat to get picture from.

Returns

Opened temporary file object. The file object MUST have appropriate extension name that matches to the format of picture sent, and seek to position 0.

It MAY be deleted or discarded once closed, if not needed otherwise.

Return type BinaryIO

Raises

- ***EFBChatNotFound*** – Raised when a chat required is not found.
- ***EFBOperationNotSupported*** – Raised when the chat does not offer a profile picture.

Examples

```
if chat.channel_uid != self.channel_uid:
    raise EFBChannelNotFound()
file = tempfile.NamedTemporaryFile(suffix=".png")
response = requests.post("https://api.example.com/get_profile_picture/png",
                         data={"uid": chat.uid})
if response.status_code == 404:
    raise EFBChatNotFound()
file.write(response.content)
file.seek(0)
return file
```

abstract get_chats()

Return a list of available chats in the channel.

Returns a list of available chats in the channel.

Return type Collection[*Chat*]

get_extra_functions()

Get a list of additional features

Return type Dict[NewType()(ExtraCommandName, str), Callable]

Returns A dict of methods marked as additional features. Method can be called with `get_extra_functions()["methodName"]()`.

10.1.1 Common operations

Sending messages and statuses

Sending messages and statuses to other channels is the most common operation of a channel. When enough information is gathered from external sources, the channel would then further process and pack them into the relevant objects, i.e. *Message* and *Status*.

When the object is built, the channel should sent it to the coordinator for following steps.

For now, both *Message* and *Status* has an attribute that indicates that where this object would be delivered to (`deliver_to` and *destination_channel*). This is used by the coordinator when delivering the message.

Messages MUST be sent using `coordinator.send_message()`. Statuses MUST be sent using `coordinator.send_status()`.

When the object is passed onto the coordinator, it will be further processed by the middleware and then to its destination.

For example, to send a message to the master channel

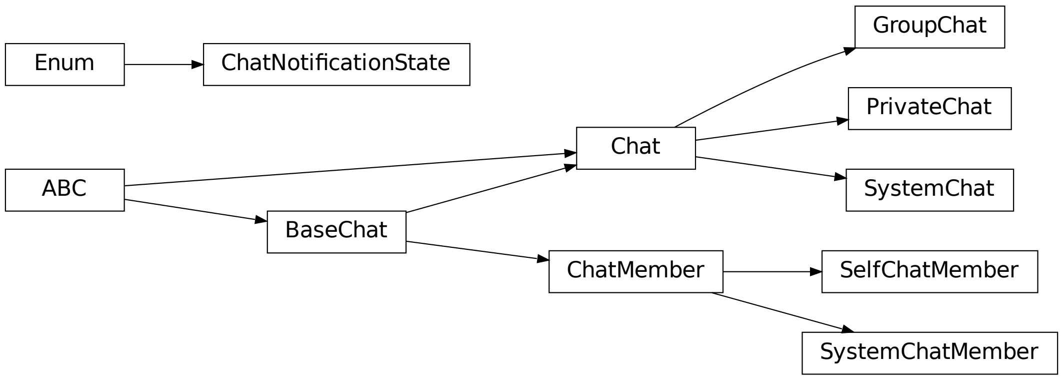
```python
def on_message(self, data: Dict[str, Any]):
    """Callback when a message is received by the slave channel from
    the IM platform.
    """
    # Prepare message content ...
    message = coordinator.send_message(Message(
        chat=chat,
        author=author,
        type=message_type,
        text=text,
        # more details ...
        uid=data['uid'],
        deliver_to=coordinator.master
    ))
    # Post-processing ...
```

10.1.2 About Channel ID

With the introduction of instance IDs, it is required to use the `self.channel_id` or equivalent instead of any hard-coded ID or constants while referring to the channel (e.g. while retrieving the path to the configuration files, creating chat and message objects, etc).

10.2 Chat and Chat Members

Inheritance diagram



Summary

PrivateChat(*[, channel, middleware, ...])	A private chat, where usually only the User Themself and the other participant are in the chat.
SystemChat(*[, channel, middleware, ...])	A system chat, where usually only the User Themself and the other participant (system chat member) are in the chat.
GroupChat(*[, channel, middleware, ...])	A group chat, where there are usually multiple members present.
ChatMember(chat, *[, name, alias, uid, id, ...])	Member of a chat.
SelfChatMember(chat, *[, name, alias, id, ...])	The User Themself as member of a chat.
SystemChatMember(chat, *[, name, alias, id, ...])	A system account/prompt as member of a chat.
ChatNotificationState(value)	Indicates the notifications settings of a chat in its slave channel or middleware.

Classes

class `ehforwarderbot.chat.`**BaseChat**(*, *channel=None*, *middleware=None*, *module_name=''*, *channel_emoji=''*, *module_id=''*, *name=''*, *alias=None*, *uid=''*, *id=''*, *vendor_specific=None*, *description=''*)

Bases: `abc.ABC`

Base chat class, this is an abstract class sharing properties among all chats and members. No instance can be created directly from this class.

Note: `BaseChat` objects are picklable, thus it is RECOMMENDED to keep any object of its subclass also picklable.

module_id

Unique ID of the module.

Type *ModuleID* (str)

channel_emoji

Emoji of the channel, empty string if the chat is from a middleware.

Type str

module_name

Name of the module.

Type *ModuleID* (str)

name

Name of the chat.

Type str

alias

Alternative name of the chat, usually set by user.

Type Optional[str]

uid

Unique ID of the chat. This MUST be unique within the channel.

Type *ChatID* (str)

description
A text description of the chat, usually known as "bio", "description", "purpose", or "topic" of the chat.

Type str

vendor_specific
Any vendor specific attributes.

Type Dict[str, Any]

__init__(**, channel=None, middleware=None, module_name='', channel_emoji='', module_id='', name='', alias=None, uid='', id='', vendor_specific=None, description=''*)

Parameters

- **channel** (Optional[*SlaveChannel*]) – Provide the channel object to fill *module_name*, *channel_emoji*, and *module_id* automatically.
- **middleware** (Optional[*Middleware*]) – Provide the middleware object to fill *module_name*, and *module_id* automatically.
- **module_id** (`NewType()`(`ModuleID`, `str`)) – Unique ID of the module.
- **channel_emoji** (*str*) – Emoji of the channel, empty string if the chat is from a middleware.
- **module_name** (*str*) – Name of the module.
- **name** (*str*) – Name of the chat.
- **alias** (*Optional[str]*) – Alternative name of the chat, usually set by user.
- **uid** (`NewType()`(`ChatID`, `str`)) – Unique ID of the chat. This MUST be unique within the channel.
- **description** (*str*) – A text description of the chat, usually known as "bio", "description", "purpose", or "topic" of the chat.
- **vendor_specific** (*Dict[str, Any]*) – Any vendor specific attributes.

copy()
Return a shallow copy of the object.

Return type `TypeVar`(`_BaseChatSelf`, bound= *BaseChat*, covariant=True)

property display_name: str
Shortcut property, equivalent to `alias or name`

Return type `str`

property long_name: str
Shortcut property, if alias exists, this will provide the alias with name in parenthesis. Otherwise, this will return the name

Return type `str`

abstract verify()
Verify the completeness of the data.

Raises **AssertionError** – When this chat is invalid.

class ehforwarderbot.chat.**Chat**(**, channel=None, middleware=None, module_name='', channel_emoji='', module_id='', name='', alias=None, id='', uid='', vendor_specific=None, description='', members=None, notification=ChatNotificationState.ALL, with_self=True*)

Bases: `ehforwarderbot.chat.BaseChat`, `abc.ABC`

A chat object, indicates a user, a group, or a system chat. This class is abstract. No instance can be created directly from this class.

If your IM platform is providing an ID of the User Themself, and it is using this ID to indicate the author of a message, you SHOULD update `Chat.self.uid` accordingly.

```
>>> channel.my_chat_id
"david_divad"
>>> chat = Chat(channel=channel, name="Alice", uid=ChatID("alice123"))
>>> chat.self.uid = channel.my_chat_id
```

By doing so, you can get the author in one step:

```
author = chat.get_member(author_id)
```

… instead of using a condition check:

```
if author_id == channel.my_chat_id:
    author = chat.self
else:
    author = chat.get_member(author_id)
```

Note: `Chat` objects are picklable, thus it is RECOMMENDED to keep any object of its subclass also picklable.

module_id

Unique ID of the module.

Type `ModuleID` (str)

channel_emoji

Emoji of the channel, empty string if the chat is from a middleware.

Type str

module_name

Name of the module.

Type str

name

Name of the chat.

Type str

alias

Alternative name of the chat, usually set by user.

Type Optional[str]

uid

Unique ID of the chat. This MUST be unique within the channel.

Type `ChatID` (str)

description
A text description of the chat, usually known as “bio”, “description”, “purpose”, or “topic” of the chat.

Type str

notification
Indicate the notification settings of the chat in its slave channel (or middleware), defaulted to `ALL`.

Type *ChatNotificationState*

members
Provide a list of members in the chat. Defaulted to an empty `list`.

You can extend this object and implement a `@property` method set for loading members on demand.

Note that this list may include members created by middlewares when the object is a part of a message, and these members MAY not appear when trying to retrieve from the slave channel directly. These members would have a different *module_id* specified from the chat.

Type list of *ChatMember*

vendor_specific
Any vendor specific attributes.

Type Dict[str, Any]

self
the User as a member of the chat (if available).

Type Optional[*SelfChatMember*]

__init__(**, channel=None, middleware=None, module_name='', channel_emoji='', module_id='', name='', alias=None, id='', uid='', vendor_specific=None, description='', members=None, notification=ChatNotificationState.ALL, with_self=True*)

Keyword Arguments

- **module_id** (*str*) – Unique ID of the module.
- **channel_emoji** (*str*) – Emoji of the channel, empty string if the chat is from a middleware.
- **module_name** – Name of the module.
- **name** (*str*) – Name of the chat.
- **alias** (*Optional[str]*) – Alternative name of the chat, usually set by user.
- **id** – Unique ID of the chat. This MUST be unique within the channel.
- **description** (*str*) – A text description of the chat, usually known as “bio”, “description”, “purpose”, or “topic” of the chat.
- **notification** (`ChatNotificationState`) – Indicate the notification settings of the chat in its slave channel (or middleware), defaulted to `ALL`.
- **members** (MutableSequence[*ChatMember*]) – Provide a list of members of the chat. Defaulted to an empty `list`.
- **vendor_specific** (*Dict[str, Any]*) – Any vendor specific attributes.
- **with_self** (*bool*) – Initialize the chat with the User Themself as a member.

add_member(*name*, *uid*, *alias=None*, *id=''*, *vendor_specific=None*, *description=''*, *middleware=None*)
Add a member to the chat.

Tip: This method does not check for duplicates. Only add members with this method if you are sure that they are not added yet. To check if the member is already added before adding, you can do something like this:

```
with contextlib.suppress(KeyError):
    return chat.get_member(uid)
return chat.add_member(name, uid, alias=..., vendor_specific=...)
```

Parameters

- **name** (str) – Name of the member.
- **uid** (NewType()(ChatID, str)) – ID of the member.

Keyword Arguments

- **alias** (Optional[str]) – Alias of the member.
- **vendor_specific** (Dict[str, Any]) – Any vendor specific attributes.
- **description** (str) – A text description of the chat, usually known as "bio", "description", "purpose", or "topic" of the chat.
- **middleware** (Optional[Middleware]) – Initialize this chat as a part of a middleware.

Return type ChatMember

add_self()
Add self to the list of members.

Raises AssertionError – When there is already a self in the list of members.

Return type SelfChatMember

add_system_member(*name=''*, *alias=None*, *id=''*, *uid=''*, *vendor_specific=None*, *description=''*, *middleware=None*)
Add a system member to the chat.

Useful for slave channels and middlewares to create an author of a message from a system member when the "system" member is intended to become a member of the chat.

Tip: This method does not check for duplicates. Only add members with this method if you are sure that they are not added yet.

Keyword Arguments

- **name** (str) – Name of the member.
- **uid** – ID of the member.
- **alias** (Optional[str]) – Alias of the member.
- **vendor_specific** (Dict[str, Any]) – Any vendor specific attributes.
- **description** (str) – A text description of the chat, usually known as "bio", "description", "purpose", or "topic" of the chat.

- **middleware** (Optional[*Middleware*]) – Initialize this chat as a part of a middleware.

Return type *SystemChatMember*

get_member (*member_id*)
Find a member of chat by its ID.

Parameters **member_id** (NewType()(ChatID, str)) – ID of the chat member.

Return type *ChatMember*

Returns the chat member.

Raises **KeyError** – when the ID provided is not found.

property has_self: bool
Indicate if this chat has yourself.

Return type bool

make_system_member (*name='', alias=None, id='', uid='', vendor_specific=None, description='', middleware=None*)
Make a system member for this chat.

Useful for slave channels and middlewares to create an author of a message from a system member when the "system" member is NOT intended to become a member of the chat.

Keyword Arguments

- **name** (*str*) – Name of the member.
- **uid** – ID of the member.
- **alias** (*Optional[str]*) – Alias of the member.
- **vendor_specific** (*Dict[str, Any]*) – Any vendor specific attributes.
- **description** (*str*) – A text description of the chat, usually known as "bio", "description", "purpose", or "topic" of the chat.
- **middleware** (Optional[*Middleware*]) – Initialize this chat as a part of a middleware.

Return type *SystemChatMember*

self: Optional[*ehforwarderbot.chat.SelfChatMember*]
The user as a member of the chat (if available).

class ehforwarderbot.chat.**ChatMember** (*chat, *, name='', alias=None, uid='', id='', vendor_specific=None, description='', middleware=None*)
Bases: *ehforwarderbot.chat.BaseChat*

Member of a chat. Usually indicates a member in a group, or the other participant in a private chat. Chat bots created by the users of the IM platform is also considered as a plain *ChatMember*.

To represent the User Themself, use *SelfChatMember*.

To represent a chat member that is a part of the system, the slave channel, or a middleware, use *SystemChatMember*.

*ChatMember*s MUST be created with reference of the chat it belongs to. Different objects MUST be created even when the same person appears in different groups or in a private chat.

*ChatMember*s are RECOMMENDED to be created using *Chat.add_member()* method.

Note: `ChatMember` objects are picklable, thus it is RECOMMENDED to keep any object of its subclass also picklable.

`__init__`(*chat*, ***, *name=''*, *alias=None*, *uid=''*, *id=''*, *vendor_specific=None*, *description=''*, *middleware=None*)

Parameters **`chat`** (`Chat`) – Chat associated with this member.

Keyword Arguments

- **`name`** (`str`) – Name of the member.
- **`alias`** (`Optional[str]`) – Alternative name of the member, usually set by user.
- **`uid`** (`ChatID` (str)) – Unique ID of the member. This MUST be unique within the channel. This ID can be the same with a private chat of the same person.
- **`description`** (`str`) – A text description of the member, usually known as "bio", "description", "summary" or "introduction" of the member.
- **`middleware`** (`Middleware`) – Initialize this chat as a part of a middleware.

`verify`()

Verify the completeness of the data.

Raises **`AssertionError`** – When this chat is invalid.

class `ehforwarderbot.chat.`**`ChatNotificationState`**(*value*)

Bases: `enum.Enum`

Indicates the notifications settings of a chat in its slave channel or middleware. If an exact match is not available, choose the most similar one.

`ALL = -1`

All messages in the chat triggers notifications.

`MENTIONS = 1`

Notifications are sent only when the User is mentioned in the message, in the form of @-references or quote-reply (message target).

`NONE = 0`

No notification is sent to slave IM channel at all.

class `ehforwarderbot.chat.`**`GroupChat`**(***, *channel=None*, *middleware=None*, *module_name=''*, *channel_emoji=''*, *module_id=''*, *name=''*, *alias=None*, *id=''*, *uid=''*, *vendor_specific=None*, *description=''*, *notification=ChatNotificationState.ALL*, *with_self=True*)

Bases: `ehforwarderbot.chat.Chat`

A group chat, where there are usually multiple members present.

Members can be added with the `add_member()` method.

If the `with_self` argument is `True` (which is the default setting), the User Themself would also be initialized as a member of the chat.

Examples

```
>>> group = GroupChat(channel=slave_channel, name="Wonderland", uid=ChatID(
↪"wonderland001"))
>>> group.add_member(name="Alice", uid=ChatID("alice"))
ChatMember(chat=<GroupChat: Wonderland (wonderland001) @ Example slave channel>,␣
↪name='Alice', alias=None, uid='alice', vendor_specific={}, description='')
>>> group.add_member(name="bob", alias="Bob James", uid=ChatID("bob"))
ChatMember(chat=<GroupChat: Wonderland (wonderland001) @ Example slave channel>,␣
↪name='bob', alias='Bob James', uid='bob', vendor_specific={}, description='')
>>> from pprint import pprint
>>> pprint(group.members)
[SelfChatMember(chat=<GroupChat: Wonderland (wonderland001) @ Example slave␣
↪channel>, name='You', alias=None, uid='__self__', vendor_specific={},␣
↪description=''),
 ChatMember(chat=<GroupChat: Wonderland (wonderland001) @ Example slave channel>,␣
↪name='Alice', alias=None, uid='alice', vendor_specific={}, description=''),
 ChatMember(chat=<GroupChat: Wonderland (wonderland001) @ Example slave channel>,␣
↪name='bob', alias='Bob James', uid='bob', vendor_specific={}, description='')]
```

Note: `GroupChat` objects are picklable, thus it is RECOMMENDED to keep any object of its subclass also picklable.

verify()

Verify the completeness of the data.

Raises `AssertionError` – When this chat is invalid.

class ehforwarderbot.chat.**PrivateChat**(**, channel=None, middleware=None, module_name='', channel_emoji='', module_id='', name='', alias=None, id='', uid='', vendor_specific=None, description='', notification=ChatNotificationState.ALL, with_self=True, other_is_self=False*)

Bases: *ehforwarderbot.chat.Chat*

A private chat, where usually only the User Themself and the other participant are in the chat. Chat bots SHOULD also be categorized under this type.

There SHOULD only be at most one non-system member of the chat apart from the User Themself, otherwise it might lead to unintended behavior.

This object is by default initialized with the other participant as its member.

If the `with_self` argument is `True` (which is the default setting), the User Themself would also be initialized as a member of the chat.

Parameters **other** – the other participant of the chat as a member

Note: `PrivateChat` objects are picklable, thus it is RECOMMENDED to keep any object of its subclass also picklable.

verify()

Verify the completeness of the data.

Raises `AssertionError` – When this chat is invalid.

class ehforwarderbot.chat.**SelfChatMember**(*chat*, ***, *name=''*, *alias=None*, *id=''*, *uid=''*, *vendor_specific=None*, *description=''*, *middleware=None*)

Bases: `ehforwarderbot.chat.ChatMember`

The User Themself as member of a chat.

`SelfChatMember`s are RECOMMENDED to be created together with a chat object by setting `with_self` value to `True`. The created object is accessible at `Chat.self`.

The default ID of a `SelfChatMember` object is `SelfChatMember.SELF_ID`, and the default name is a translated version of the word "You".

You are RECOMMENDED to change the ID of this object if provided by your IM platform, and you MAY change the name or alias of this object depending on your needs.

Note: `SelfChatMember` objects are picklable, thus it is RECOMMENDED to keep any object of its subclass also picklable.

SELF_ID

The default ID of a `SelfChatMember`.

__init__(*chat*, ***, *name=''*, *alias=None*, *id=''*, *uid=''*, *vendor_specific=None*, *description=''*, *middleware=None*)

Parameters **chat** (`Chat`) – Chat associated with this member.

Keyword Arguments

- **name** (`str`) – Name of the member.
- **alias** (`Optional[str]`) – Alternative name of the member, usually set by user.
- **uid** (`ChatID` (str)) – Unique ID of the member. This MUST be unique within the channel. This ID can be the same with a private chat of the same person.
- **description** (`str`) – A text description of the member, usually known as "bio", "description", "summary" or "introduction" of the member.
- **middleware** (`Middleware`) – Initialize this chat as a part of a middleware.

class ehforwarderbot.chat.**SystemChat**(***, *channel=None*, *middleware=None*, *module_name=''*, *channel_emoji=''*, *module_id=''*, *name=''*, *alias=None*, *id=''*, *uid=''*, *vendor_specific=None*, *description=''*, *notification=ChatNotificationState.ALL*, *with_self=True*)

Bases: `ehforwarderbot.chat.Chat`

A system chat, where usually only the User Themself and the other participant (system chat member) are in the chat. This object is used to represent system chat where the other participant is neither a user nor a chat bot of the remote IM.

Middlewares are RECOMMENDED to create chats with this type when they want to send messages in this type.

This object is by default initialized with the system participant as its member.

If the `with_self` argument is `True` (which is the default setting), the User Themself would also be initialized as a member of the chat.

Parameters **other** – the other participant of the chat as a member

Note: `SystemChat` objects are picklable, thus it is RECOMMENDED to keep any object of its subclass also picklable.

verify()

Verify the completeness of the data.

Raises `AssertionError` – When this chat is invalid.

class ehforwarderbot.chat.**SystemChatMember**(*chat*, ***, *name=''*, *alias=None*, *id=''*, *uid=''*, *vendor_specific=None*, *description=''*, *middleware=None*)

Bases: *ehforwarderbot.chat.ChatMember*

A system account/prompt as member of a chat.

Use this chat to send messages that is not from any specific member. Middlewares are RECOMMENDED to use this member type to communicate with the User in an existing chat.

Chat bots created by the users of the IM platform SHOULD NOT be created as a *SystemChatMember*, but a plain *ChatMember* instead.

*SystemChatMember*s are RECOMMENDED to be created using *Chat.add_system_member()* or *Chat.make_system_member()* method.

Note: `SystemChatMember` objects are picklable, thus it is RECOMMENDED to keep any object of its subclass also picklable.

SYSTEM_ID

The default ID of a *SystemChatMember*.

__init__(*chat*, ***, *name=''*, *alias=None*, *id=''*, *uid=''*, *vendor_specific=None*, *description=''*, *middleware=None*)

Parameters **chat** (*Chat*) – Chat associated with this member.

Keyword Arguments

- **name** (*str*) – Name of the member.
- **alias** (*Optional[str]*) – Alternative name of the member, usually set by user.
- **uid** (*ChatID* (str)) – Unique ID of the member. This MUST be unique within the channel. This ID can be the same with a private chat of the same person.
- **description** (*str*) – A text description of the member, usually known as "bio", "description", "summary" or "introduction" of the member.
- **middleware** (*Middleware*) – Initialize this chat as a part of a middleware.

10.3 Constants

class `ehforwarderbot.constants.`**`MsgType`**(*value*)

An enumeration.

`Animation = 'Animation'`

Message with an animation, usually in the form of GIF or soundless video.

`Audio = 'Voice'`

Audio messages (deprecated).

Deprecated since version Use: *`Voice`* if the message has a voice message (usually recorded). Use *`File`* if the message has a music file (usually uploaded).

`File = 'File'`

File message.

`Image = 'Image'`

Image (picture) message.

Notes

Animated GIF images must use *`Animation`* type instead.

`Link = 'Link'`

Message that is mainly one specific link, or a text message with one link preview.

`Location = 'Location'`

Location message.

`Status = 'Status'`

Status from a user in a chat, usually typing and uploading.

`Sticker = 'Sticker'`

Pictures sent with few text caption, usually a transparent background, and a limited number of options that is usually not from the user's photo gallery.

`Text = 'Text'`

Text message

`Unsupported = 'Unsupported'`

Any type of message that is not listed above. A text representation is required.

`Video = 'Video'`

Video message

`Voice = 'Voice'`

Voice messages, usually recorded right before sending.

10.4 Coordinator

Coordinator among channels.

`ehforwarderbot.coordinator.`**`profile`**

Name of current profile..

Type str

`ehforwarderbot.coordinator.`**`mutex`**

Global interaction thread lock.

Type threading.Lock

`ehforwarderbot.coordinator.`**`master`**

The running master channel object.

Type *Channel*

`ehforwarderbot.coordinator.`**`slaves`**

Dictionary of running slave channel object. Keys are the unique identifier of the channel.

Type Dict[str, EFBChannel]

`ehforwarderbot.coordinator.`**`middlewares`**

List of middlewares

Type List[*Middleware*]

`ehforwarderbot.coordinator.`**`add_channel`**`(`*channel*`)`

Register the channel with the coordinator.

Parameters **`channel`** (`Channel`) – Channel to register

`ehforwarderbot.coordinator.`**`add_middleware`**`(`*middleware*`)`

Register a middleware with the coordinator.

Parameters **`middleware`** (`Middleware`) – Middleware to register

`ehforwarderbot.coordinator.`**`get_module_by_id`**`(`*module_id*`)`

Return the module instance of a provided module ID

Parameters **`module_id`** (`NewType()` (`ModuleID`, `str`)) – Module ID, with instance ID if available.

Return type `Union`[*`Channel`*, *`Middleware`*]

Returns Module instance requested.

Raises **`NameError`** – When the module is not found.

`ehforwarderbot.coordinator.`**`master:`** ***`ehforwarderbot.channel.MasterChannel`***

The instance of the master channel.

`ehforwarderbot.coordinator.`**`master_thread: Optional[threading.Thread] = None`**

The thread running poll() of the master channel.

`ehforwarderbot.coordinator.`**`middlewares:`**
`List[`*`ehforwarderbot.middleware.Middleware`*`] = []`

Instances of middlewares. Sorted in the order of execution.

`ehforwarderbot.coordinator.`**`mutex: _thread.allocate_lock = <unlocked _thread.lock object>`**

Mutual exclusive lock for user interaction through CLI interface

ehforwarderbot.coordinator.**profile: str = 'default'**
Current running profile name

ehforwarderbot.coordinator.**send_message**(*msg*)
Deliver a new message or edited message to the destination channel.

Parameters **msg** (Message) – The message

Return type Optional[*Message*]

Returns The message processed and delivered by the destination channel, includes the updated message ID if sent to a slave channel. Returns None if the message is not sent.

ehforwarderbot.coordinator.**send_status**(*status*)
Deliver a status to the destination channel.

Parameters **status** (Status) – The status

ehforwarderbot.coordinator.**slave_threads: Dict[ModuleID, threading.Thread] = {}**
Threads running poll() from slave channels. Keys are the channel IDs.

ehforwarderbot.coordinator.**slaves: Dict[ModuleID, *ehforwarderbot.channel.SlaveChannel*] = {}**
Instances of slave channels. Keys are the channel IDs.

ehforwarderbot.coordinator.**translator: gettext.NullTranslations = <gettext.NullTranslations object>**
Internal GNU gettext translator.

10.5 Exceptions

exception ehforwarderbot.exceptions.**EFBException**
Bases: Exception

A general class to indicate that the exception is from EFB framework.

exception ehforwarderbot.exceptions.**EFBChatNotFound**
Bases: *ehforwarderbot.exceptions.EFBException*

Raised by a slave channel when a chat indicated is not found.

Can be raised by any method that involves a chat or a message.

exception ehforwarderbot.exceptions.**EFBChannelNotFound**
Bases: *ehforwarderbot.exceptions.EFBException*

Raised by the coordinator when the message sent delivers to a missing channel.

exception ehforwarderbot.exceptions.**EFBMessageError**
Bases: *ehforwarderbot.exceptions.EFBException*

Raised by slave channel for any other error occurred when sending a message or a status.

Can be raised in *Channel.send_message()* and *Channel.send_status()*.

exception ehforwarderbot.exceptions.**EFBMessageNotFound**
Bases: *ehforwarderbot.exceptions.EFBMessageError*

Raised by a slave channel when a message indicated is not found.

Can be raised in *Channel.send_message()* (edited message / target message not found) and in *Channel.send_status()* (message to delete is not found).

exception ehforwarderbot.exceptions.**EFBMessageTypeNotSupported**
Bases: *ehforwarderbot.exceptions.EFBMessageError*

Raised by a slave channel when the indicated message type is not supported.

Can be raised in *Channel.send_message()*.

exception ehforwarderbot.exceptions.**EFBOperationNotSupported**
Bases: *ehforwarderbot.exceptions.EFBMessageError*

Raised by slave channels when a chat operation is not supported. E.g.: cannot edit message, cannot delete message.

Can be raised in *Channel.send_message()* and *Channel.send_status()*.

exception ehforwarderbot.exceptions.**EFBMessageReactionNotPossible**
Bases: *ehforwarderbot.exceptions.EFBException*

Raised by slave channel when a message reaction request from master channel is not possible to be processed.

Can be raised in *Channel.send_status()*.

10.6 Message

Summary

Message(*[, attributes, chat, author, ...])	A message.
LinkAttribute(title[, description, image, url])	Attributes for link messages.
LocationAttribute(latitude, longitude)	Attributes for location messages.
StatusAttribute(status_type[, timeout])	Attributes for status messages.
MessageCommands(commands)	Message commands.
MessageCommand(name, callable_name[, args, ...])	A message command.
Substitutions(substitutions)	Message text substitutions, or "@-references".

Classes

class ehforwarderbot.message.**Message**(**, attributes=None, chat=None, author=None, commands=None, deliver_to=None, edit=False, edit_media=False, file=None, filename=None, is_system=False, mime=None, path=None, reactions=None, substitutions=None, target=None, text='', type=MsgType.Unsupported, uid=None, vendor_specific=None*)

A message.

Note: `Message` objects are picklable, thus it is strongly RECOMMENDED to keep any object of its subclass also picklable.

Keyword Arguments

- **attributes** (Optional[*MessageAttribute*]) – Attributes used for a specific message type. Only specific message type requires this attribute. Defaulted to `None`.
  - Link: *LinkAttribute*

   - Location: *LocationAttribute*
   - Status: Typing/Sending files/etc.: *StatusAttribute*

Note: Do NOT use object of the abstract class *MessageAttribute* for `attributes`, but object of specific class instead.

- **chat** (*Chat*) – Sender of the message.
- **author** (*ChatMember*) – Author of this message. Author of the message MUST be indicated as a part of the same `chat` this message is from. If the message is sent from the User Themself, this MUST be an object of *SelfChatMember*.

  Note that the author MAY not be inside *members* of the chat of this message. The author MAY have a different *module_id* from the `chat`, and could be unretrievable otherwise.
- **commands** (Optional[*MessageCommands*]) – Commands attached to the message

  This attribute will be ignored in _Status_ messages.
- **deliver_to** (*Channel*) – The channel that the message is to be delivered to.
- **edit** (*bool*) – Flag this up if the message is edited. Flag only this if no multimedia file is modified, otherwise flag up both this one and `edit_media` as well.

  If no media file is modified, the edited message MAY carry no information about the file.

  This attribute will be ignored in _Status_ messages.
- **edit_media** (*bool*) – Flag this up if any file attached to the message is modified. If this value is true, `edit` MUST also be `True`. This attribute is ignored if the message type is not supposed to contain any media file, e.g. `Text`, `Location`, etc.

  This attribute will be ignored in _Status_ messages.
- **file** (*Optional[BinaryIO]*) – File object to multimedia file, type "rb". `None` if N/A. Recommended to use `NamedTemporaryFile`. The file SHOULD be able to be safely deleted (or otherwise discarded) once closed. All file object MUST be sought back to 0 (`file.seek(0)`) before sending.
- **filename** (*Optional[str]*) – File name of the multimedia file. `None` if N/A
- **is_system** (*bool*) – Mark as true if this message is a system message.
- **mime** (*Optional[str]*) – MIME type of the file. `None` if N/A
- **path** (*Optional[Path]*) – Local path of multimedia file. `None` if N/A
- **reactions** (Dict[str, Collection[`Chat`]]) – Indicate reactions to the message. Dictionary key is the canonical name of reaction, usually an emoji. Value is a collection of users who reacted to the message with that certain emoji. All `Chat` objects in this dict MUST be members in the chat of this message.

  This attribute will be ignored in _Status_ messages.
- **substitutions** (Optional[*Substitutions*]) – Substitutions of messages, usually used when the some parts of the text of the message refers to another user or chat.

  This attribute will be ignored in _Status_ messages.
- **target** (Optional[*Message*]) – Target message (usually for messages that "replies to" another message).

  This attribute will be ignored in _Status_ messages.

Note: This message MAY be a “minimum message”, with only required fields:

– `Message.chat`

– `Message.author`

– `Message.text`

– `Message.type`

– `Message.uid`

- **`text`** (*`str`*) – Text of the message.

  This attribute will be ignored in _Status_ messages.

- **`type`** (*`MsgType`*) – Type of message

- **`uid`** (*`str`*) – Unique ID of message. Usually stores the message ID from slave channel. This ID MUST be unique among all chats in the same channel.

Note: Some channels may not support message editing. Some channels may issue a new uid for edited message.

- **`vendor_specific`** (*`Dict[str, Any]`*) – A series of vendor specific attributes attached. This can be used by any other channels or middlewares that is compatible with such information. Note that no guarantee is provided for information in this section.

`property link: Optional[ehforwarderbot.message.LinkAttribute]`
Get the link attributes of the current message, if available.

Return type `Optional`[*`LinkAttribute`*]

`property location: Optional[ehforwarderbot.message.LocationAttribute]`
Get the location attributes of the current message, if available.

Return type `Optional`[*`LocationAttribute`*]

`property status: Optional[ehforwarderbot.message.StatusAttribute]`
Get the status attributes of the current message, if available.

Return type `Optional`[*`StatusAttribute`*]

`verify``()`
Verify the validity of message.

Raises **`AssertionError`** – when the message is not valid

`class` `ehforwarderbot.message.`**`MessageAttribute`**
Bases: `abc.ABC`

Abstract class of a message attribute.

`class` `ehforwarderbot.message.`**`LinkAttribute`**(*title*, *description=None*, *image=None*, *url=''*)
Bases: *`ehforwarderbot.message.MessageAttribute`*

Attributes for link messages.

`title`
Title of the link.

Type str

description
Description of the link.

Type str, optional

image
Image/thumbnail URL of the link.

Type str, optional

url
URL of the link.

Type str

__init__(*title*, *description=None*, *image=None*, *url=''*)

Parameters

- **title** (*str*) – Title of the link.
- **description** (*str, optional*) – Description of the link.
- **image** (*str, optional*) – Image/thumbnail URL of the link.
- **url** (*str*) – URL of the link.

class ehforwarderbot.message.**LocationAttribute**(*latitude*, *longitude*)
Bases: *ehforwarderbot.message.MessageAttribute*

Attributes for location messages.

latitude
Latitude of the location.

Type float

longitude
Longitude of the location.

Type float

__init__(*latitude*, *longitude*)

Parameters

- **latitude** (*float*) – Latitude of the location.
- **longitude** (*float*) – Longitude of the location.

class ehforwarderbot.message.**MessageCommand**(*name*, *callable_name*, *args=None*, *kwargs=None*)
Bases: `object`

A message command.

This object records a way to call a method in the module object. In case where the message has an `author` from a different module from the `chat`, this function MUST be called on the `author`'s module.

The method specified MUST return either a `str` as result or `None` if this message will be edited or deleted for further interactions.

name
Human-friendly name of the command.

Type str

callable_name
Callable name of the command.

Type str

args
Arguments passed to the function.

Type Collection[Any]

kwargs
Keyword arguments passed to the function.

Type Mapping[str, Any]

__init__(*name*, *callable_name*, *args=None*, *kwargs=None*)

Parameters

- **name** (`str`) – Human-friendly name of the command.
- **callable_name** (`str`) – Callable name of the command.
- **args** (`Optional[Collection[Any]]`) – Arguments passed to the function. Defaulted to empty list;
- **kwargs** (`Optional[Mapping[str, Any]]`) – Keyword arguments passed to the function. Defaulted to empty dict.

class `ehforwarderbot.message.`**MessageCommands**(*commands*)
Bases: `List`[`ehforwarderbot.message.MessageCommand`]

Message commands.

Message commands allow user to take action to a specific message, including vote, add friends, etc.

commands
Commands for the message.

Type list of `MessageCommand`

__init__(*commands*)

Parameters **commands** (list of `MessageCommand`) – Commands for the message.

class `ehforwarderbot.message.`**StatusAttribute**(*status_type*, *timeout=5000*)
Bases: `ehforwarderbot.message.MessageAttribute`

Attributes for status messages.

Message with type `Status` notifies the other end to update a chat-specific status, such as typing, send files, etc.

status_type
Type of status, possible values are defined in the `StatusAttribute`.

timeout
Number of milliseconds for this status to expire. Default to 5 seconds.

Type Optional[int]

Types
List of status types supported

class Types(*value*)
Bases: `enum.Enum`

TYPING
Used in *status_type*, represent the status of typing.

UPLOADING_FILE
Used in *status_type*, represent the status of uploading file.

UPLOADING_IMAGE
Used in *status_type*, represent the status of uploading image.

UPLOADING_VOICE
Used in *status_type*, represent the status of uploading voice.

UPLOADING_VIDEO
Used in *status_type*, represent the status of uploading video.

__init__ (*status_type*, *timeout=5000*)

Parameters

- **status_type** (*Types*) – Type of status.
- **timeout** (*Optional[int]*) – Number of milliseconds for this status to expire. Default to 5 seconds.

class ehforwarderbot.message.**Substitutions** (*substitutions*)
Bases: Dict[Tuple[int, int], Union[*ehforwarderbot.chat.Chat*, *ehforwarderbot.chat.ChatMember*]]

Message text substitutions, or “@-references”.

This is for the case when user “@-referred” a list of users in the message. Substitutions here is a dict of correspondence between the index of substring used to refer to a user/chat in the message and the chat object it referred to.

Values of the dictionary MUST be either a member of the chat (`self` or the other for private chats, group members for group chats) or another chat of the slave channel.

A key in this dictionary is a tuple of two `int`s, where first of it is the starting position in the string, and the second is the ending position defined similar to Python’s substring. A tuple of `(3, 15)` corresponds to `msg.text[3:15]`. The value of the tuple `(a, b)` MUST satisfy $0 \leq a < b \leq l$, where l is the length of the message text.

Type: Dict[Tuple[int, int], *Chat*]

property is_mentioned: bool
Returns `True` if you are mentioned in this message.

In the case where a chat (private or group) is mentioned in this message instead of a group member, you will also be considered mentioned if you are a member of the chat.

Return type `bool`

10.6.1 Examples

Prelude: Defining related chats

```
master: MasterChannel = coordinator.master
slave: SlaveChannel = coordinator.slave['demo.slave']
alice: PrivateChat = slave.get_chat("alice101")
bob: PrivateChat = slave.get_chat("bobrocks")
wonderland: GroupChat = slave.get_chat("thewonderlandgroup")
wonderland_alice: ChatMember = wonderland.get_member("alice101")
```

Initialization and marking chats

1. A message delivered from slave channel to master channel

```
message = Message(
    deliver_to=master,
    chat=wonderland,
    author=wonderland_alice,
    # More attributes go here...
)
```

2. A message delivered from master channel to slave channel

```
message = Message(
    deliver_to=slave,
    chat=alice,
    author=alice.self,
    # More attributes go here...
)
```

Quoting a previous message (targeted message)

Data of the quoted message SHOULD be retrieved from recorded historical data. `Message.deliver_to` is not required for quoted message, and complete data is not required here. For details, see `Message.target`.

You MAY use the `Channel.get_message()` method to get the message object from the sending channel, but this might not always be possible depending on the implementation of the channel.

```
message.target = Message(
    chat=alice,
    author=alice.other,
    text="Hello, world.",
    type=MsgType.Text,
    uid=MessageID("100000002")
)
```

Edit a previously sent message

Message ID MUST be the ID from the slave channel regardless of where the message is delivered to.

```
message.edit = True
message.uid = MessageID("100000003")
```

Type-specific Information

1. Text message

   ```
   message.type = MsgType.Text
   message.text = "Hello, Wonderland."
   ```

2. Media message

   Information related to media processing is described in *Media processing*.

   The example below is for image (picture) messages. Audio, file, video, sticker works in the same way.

   In non-text messages, the `text` attribute MAY be an empty string.

   ```
   message.type = MsgType.Image
   message.text = "Image caption"
   message.file = NamedTemporaryFile(suffix=".png")
   message.file.write(binary_data)
   message.file.seek(0)
   message.filename = "holiday photo.png"
   message.mime = "image/png"
   ```

3. Location message

   In non-text messages, the `text` attribute MAY be an empty string.

   ```
   message.type = MsgType.Location
   message.text = "I'm here! Come and find me!"
   message.attributes = LocationAttribute(51.4826, -0.0077)
   ```

4. Link message

   In non-text messages, the `text` attribute MAY be an empty string.

   ```
   message.type = MsgType.Link
   message.text = "Check it out!"
   message.attributes = LinkAttribute(
       title="Example Domain",
       description="This domain is established to be used for illustrative␣
   ↪examples in documents.",
       image="https://example.com/thumbnail.png",
       url="https://example.com"
   )
   ```

5. Status

   In status messages, the `text` attribute is disregarded.

   ```
   message.type = MsgType.Status
   message.attributes = StatusAttribute(StatusAttribute.TYPING)
   ```

6. Unsupported message

   `text` attribute is required for this type of message.

```
message.type = MsgType.Unsupported
message.text = "Alice requested USD 10.00 from you. "
               "Please continue with your Bazinga App."
```

Additional information

1. Substitution

   @-reference the User Themself, another member in the same chat, and the entire chat in the message text.

```
message.text = "Hey @david, @bob, and @all. Attention!"
message.substitutions = Substitutions({
    # text[4:10] == "@david", here David is the user.
    (4, 10): wonderland.self,
    # text[12:16] == "@bob", Bob is another member of the chat.
    (12, 16): wonderland.get_member("bob"),
    # text[22:26] == "@all", this calls the entire group chat, hence the
    # chat object is set as the following value instead.
    (22, 26): wonderland
})
```

2. Commands

```
message.text = "Carol sent you a friend request."
message.commands = MessageCommands([
    EFBCommand(name="Accept", callable_name="accept_friend_request",
               kwargs={"username": "carol_jhonos", "hash": "2a9329bd93f"}
↪),
    EFBCommand(name="Decline", callable_name="decline_friend_request",
               kwargs={"username": "carol_jhonos", "hash": "2a9329bd93f"})
])
```

10.7 Middleware

class ehforwarderbot.**Middleware**(*instance_id=None*)

Middleware class.

middleware_id

Unique ID of the middleware. Convention of IDs is specified in *Packaging and Publish*. This ID will be appended with its instance ID when available.

Type str

middleware_name

Human-readable name of the middleware.

Type str

instance_id

The instance ID if available.

Type str

`__init__`(*instance_id=None*)

Initialize the middleware. Inherited initializer MUST call the "super init" method at the beginning.

Parameters **`instance_id`** (`Optional`[`NewType()`(`InstanceID`, `str`)]) – Instance ID of the middleware.

`get_extra_functions`()

Get a list of additional features

Returns A dict of methods marked as additional features. Method can be called with `get_extra_functions()["methodName"]()`.

Return type Dict[str, Callable]

`process_message`(*message*)

Process a message with middleware

Parameters **`message`** (*`Message`*) – Message object to process

Returns Processed message or None if discarded.

Return type Optional[*`Message`*]

`process_status`(*status*)

Process a status update with middleware

Parameters **`status`** (*`Status`*) – Message object to process

Returns Processed status or None if discarded.

Return type Optional[*`Status`*]

10.7.1 About Middleware ID

With the introduction of instance IDs, it is required to use the `self.middleware_id` or equivalent instead of any hard-coded ID or constants while referring to the middleware ID (e.g. while retrieving the path to the configuration files, etc).

10.7.2 Accept commands from user through Master Channel

Despite we do not limit how the User interact with your middleware, there are 2 common ways to do it through a master channel.

Capture messages

If the action is chat-specific, you can capture messages with a specific pattern. Try to make the pattern easy to type but unique enough so that you don't accidentally catch messages that were meant to sent to the chat.

You may also construct a virtual chat or chat member of type "System" to give responses to the User.

“Additional features”

If the action is not specific to any chat, but to the system as a whole, we have provided the same command line-like interface as in slave channels to middlewares as well. Details are available at *Additional features*.

10.7.3 Chat-specific interactions

Middlewares can have chat-specific interactions through capturing messages and reply to them with a chat member created by the middleware.

The following code is an example of a middleware that interact with the user by capturing messages.

When the master channel sends a message with a text starts with `time``, the middleware captures this message and reply with the name of the chat and current time on the server. The message captured is not delivered to any following middlewares or the slave channel.

```python
def process_message(self: Middleware, message: Message) -> Optional[Message]:
    if message.deliver_to != coordinator.master and \  # sent from master channel
        text.startswith('time`'):

        # Make a system chat object.
        # For difference between `make_system_member()` and `add_system_member()`,
        # see their descriptions above.
        author = message.chat.make_system_member(
            uid="__middleware_example_time_reporter__",
            name="Time reporter",
            middleware=self
        )

        # Make a reply message
        reply = Message(
            uid=f"__middleware_example_{uuid.uuid4()}__",
            text=f"Greetings from chat {message.chat.name} on {datetime.now().
↪strftime('%c')}.",
            chat=chat,
            author=author,  # Using the new chat we created before
            type=MsgType.Text,
            target=message,  # Quoting the incoming message
            deliver_to=coordinator.master  # message is to be delivered to master
        )
        # Send the message back to master channel
        coordinator.send_message(reply)

        # Capture the message to prevent it from being delivered to following␣
↪middlewares
        # and the slave channel.
        return None

    # Continue to deliver messages not matching the pattern above.
    return message
```

10.8 Status

class ehforwarderbot.status.**Status**

Abstract class of a status

destination_channel

The channel that this status is sent to, usually the master channel.

Type *Channel*

class ehforwarderbot.status.**ChatUpdates**(*channel*, *new_chats=()*, *removed_chats=()*, *modified_chats=()*)

Inform the master channel on updates of slave chats.

channel

Slave channel that issues the update

Type *SlaveChannel*

new_chats

Unique ID of new chats

Type Optional[Collection[str]]

removed_chats

Unique ID of removed chats

Type Optional[Collection[str]]

modified_chats

Unique ID of modified chats

Type Optional[Collection[str]]

__init__(*channel*, *new_chats=()*, *removed_chats=()*, *modified_chats=()*)

Parameters

- **channel** (*SlaveChannel*) – Slave channel that issues the update
- **new_chats** (*Optional[Collection[str]]*) – Unique ID of new chats
- **removed_chats** (*Optional[Collection[str]]*) – Unique ID of removed chats
- **modified_chats** (*Optional[Collection[str]]*) – Unique ID of modified chats

class ehforwarderbot.status.**MemberUpdates**(*channel*, *chat_id*, *new_members=()*, *removed_members=()*, *modified_members=()*)

Inform the master channel on updates of members in a slave chat.

channel

Slave channel that issues the update

Type *SlaveChannel*

chat_id

Unique ID of the chat.

Type str

new_members

Unique ID of new members

Type Optional[Collection[str]]

removed_members
Unique ID of removed members

Type Optional[Collection[str]]

modified_members
Unique ID of modified members

Type Optional[Collection[str]]

__init__(*channel*, *chat_id*, *new_members=()*, *removed_members=()*, *modified_members=()*)

Parameters

- **channel** (*SlaveChannel*) – Slave channel that issues the update
- **chat_id** (*str*) – Unique ID of the chat.
- **new_members** (*Optional[Collection[str]]*) – Unique ID of new members
- **removed_members** (*Optional[Collection[str]]*) – Unique ID of removed members
- **modified_members** (*Optional[Collection[str]]*) – Unique ID of modified members

class ehforwarderbot.status.**MessageReactionsUpdate**(*chat*, *msg_id*, *reactions*)
Update reacts of a message, issued from slave channel to master channel.

Parameters

- **chat** (*Chat*) – The chat where message is sent
- **msg_id** (*str*) – ID of the message for the reacts
- **reactions** (Mapping[NewType()(ReactionName, str), Collection[*ChatMember*]]) – Indicate reactions to the message. Dictionary key represents the reaction name, usually an emoji. Value is a collection of users who reacted to the message with that certain emoji. All Chat objects in this dict MUST be members in the chat of the message.
- **destination_channel** (*MasterChannel*) – Channel the status is issued to, which is always the master channel.

__init__(*chat*, *msg_id*, *reactions*)

Parameters

- **chat** (*Chat*) – The chat where message is sent
- **msg_id** (*str*) – ID of the message for the reacts
- **reactions** (Mapping[NewType()(ReactionName, str), Collection[*ChatMember*]]) – Indicate reactions to the message. Dictionary key represents the reaction name, usually an emoji. Value is a collection of users who reacted to the message with that certain emoji. All Chat objects in this dict MUST be members in the chat of the message.

class ehforwarderbot.status.**MessageRemoval**(*source_channel*, *destination_channel*, *message*)
Inform a channel to remove a certain message.

This is usually known as "delete from everyone", "delete from recipient", "recall a message", "unsend", or "revoke a message" as well, depends on the IM platform.

Some channels MAY not support removal of messages, and raises a *exceptions.EFBOperationNotSupported* exception.

Feedback by sending another `MessageRemoval` back is not required when this object is sent from a master channel. Master channels SHOULD treat a successful delivery of this status as a successful removal.

source_channel
Channel issued the status

Type *Channel*

destination_channel
Channel the status is issued to

Type *Channel*

message
Message to remove. This MAY not be a complete *message.Message* object.

Type *Message*

Raises .exceptions.EFBOperationNotSupported – When message removal is not supported in the channel.

__init__(*source_channel*, *destination_channel*, *message*)
Create a message removal status

Try to provided as much as you can, if not, provide a minimum information in the channel:

- Slave channel ID and chat ID (*message.chat.module_id* and *message.chat.uid*)
- Message unique ID from the slave channel (`message.uid`)

Parameters

- **source_channel** (*Channel*) – Channel issued the status
- **destination_channel** (*Channel*) – Channel the status is issued to
- **message** (*Message*) – Message to remove.

class ehforwarderbot.status.**ReactToMessage**(*chat*, *msg_id*, *reaction*)
Created when user react to a message, issued from master channel.

When this status is sent, a *status.MessageReactionsUpdate* is RECOMMENDED to be issued back to master channel.

Parameters

- **chat** (`Chat`) – The chat where message is sent
- **msg_id** (*str*) – ID of the message to react to
- **reaction** (*Optional[str]*) – The reaction name to be sent, usually an emoji. Set to `None` to remove reaction.
- **destination_channel** (*SlaveChannel*) – Channel the status is issued to, extracted from the chat object.

Raises

- **`.exceptions.EFBMessageReactionNotPossible`** – Raised when the reaction is not valid (e.g. the specific reaction is not in the list of possible reactions).
- **`.exceptions.EFBOperationNotSupported`** – Raised when reaction in any form is not supported to the message at all.

`__init__`(*chat*, *msg_id*, *reaction*)

Parameters

- **`chat`** (*`Chat`*) – The chat where message is sent
- **`msg_id`** (*`str`*) – ID of the message to react to
- **`reaction`** (`Optional`[`NewType()`(`ReactionName`, `str`)]) – The reaction name to be sent, usually an emoji

10.9 Custom Type Hints

A list of type aliases when no separate class is defined for some types of values. Types for user-facing values (display names, descriptions, message text, etc.) are not otherwise defined.

Most of types listed here are defined under the "NewType" syntax in order to clarify some ambiguous values not covered by simple type checking. This is only useful if you are using static type checking in your development. If you are not using type checking of any kind, you can simply ignore values in this module.

`ehforwarderbot.types.`**`ChatID`**
Chat ID from slave channel or middleware, applicable to both chat and chat members.

alias of `str`

`ehforwarderbot.types.`**`ExtraCommandName`**
Command name of additional features, in the format of `^[A-Za-z][A-Za-z0-9_]{0,19}$`.

alias of `str`

`ehforwarderbot.types.`**`InstanceID`**
Instance ID of a module.

alias of `str`

`ehforwarderbot.types.`**`MessageID`**
Message ID from slave channel or middleware.

alias of `str`

`ehforwarderbot.types.`**`ModuleID`**
Module ID, including instance ID after `#` if available.

alias of `str`

`ehforwarderbot.types.`**`ReactionName`**
Canonical representation of a reaction, usually an emoji.

alias of `str`

`ehforwarderbot.types.`**`Reactions`**
Reactions to a message.

alias of `Mapping`[*`ReactionName`*, `Collection`[`ChatMember`]]

10.10 Utilities

`ehforwarderbot.utils.`**`extra`**`(`*name*, *desc*`)`

Decorator for slave channel's "additional features" interface.

Parameters

- **name** (`str`) – A human readable name for the function.
- **desc** (`str`) – A short description and usage of it. Use `{function_name}` in place of the function name in the description.

Return type `Callable[..., Optional[str]]`

Returns The decorated method.

Example

```
@extra(name="Echo", desc="Return the text entered.\n\nUsage:\n    {function_name}␣
↪text")
def echo(self, text: str) -> Optional[str]:
    return text
```

`ehforwarderbot.utils.`**`get_base_path`**`()`

Get the base data path for EFB. This can be defined by the environment variable `EFB_DATA_PATH`.

If `EFB_DATA_PATH` is not defined, this gives `~/.ehforwarderbot`.

This method creates the queried path if not existing.

Return type `Path`

Returns The base path.

`ehforwarderbot.utils.`**`get_config_path`**`(`*module_id=None*, *ext='yaml'*`)`

Get path for configuration file. Defaulted to `~/.ehforwarderbot/profiles/`*`profile_name`*`/`*`module_id`*`/config.yaml`.

This method creates the queried path if not existing. The config file will not be created, however.

Parameters

- **module_id** (`Optional[NewType()(ModuleID, str)]`) – Module ID.
- **ext** (`str`) – Extension name of the config file. Defaulted to `"yaml"`.

Return type `Path`

Returns The path to the configuration file.

`ehforwarderbot.utils.`**`get_custom_modules_path`**`()`

Get the path to custom channels

Return type `Path`

Returns The path for custom channels.

`ehforwarderbot.utils.`**`get_data_path`**`(`*module_id*`)`

Get the path for permanent storage of a module.

This method creates the queried path if not existing.

Parameters **module_id** (`NewType()(ModuleID, str)`) – Module ID

Return type `Path`

Returns The data path of indicated module.

`ehforwarderbot.utils.`**`locate_module`**(*module_id*, *module_type=None*)

Locate module by module ID

Parameters

- **module_id** (`NewType()`(`ModuleID`, `str`)) – Module ID
- **module_type** (`Optional`[`str`]) – Type of module, one of `'master'`, `'slave'` and `'middleware'`

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

CHAPTER 12

Feel like contributing?

Everyone is welcomed to raise an issue or submit a pull request, just remember to read through and follow the *contribution guideline* before you do so.

CHAPTER 13

Related articles

- Idea: Group Chat Tunneling (Sync) with EH Forwarder Bot
- What's so new in EH Forwarder Bot 2 (and its modules)

For tips, tricks and community contributed articles, see project wiki.

CHAPTER 14

License

EFB framework is licensed under GNU Affero General Public License 3.0 or later versions.

Donate.

Python Module Index

Index