pipobot Documentation

Release 1.0

Some net7iens

May 11, 2015

Contents

CHAPTER 1

Documentation

1.1 Installation/Configuration

1.1.1 Presentation

1.1.2 Requirements

1.1.3 Configuration

Pipobot configuration is centralized in a single yaml file. In this file you will configure *global* parameters, select rooms the bot will join, choose modules to use and configure them. An example of such file is distributed with pipobot under the name of *pipobot.conf.yml*. You can find in it a full example of configuration and can refer to it for syntax questions. In this documentation we will develop parameters you can use and what they mean.

Config section

In this section you will have to provide general parameters for pipobot, eg parameters independent of rooms and modules. Here is a list of parameters you can use:

- logpath: a *relative or absolute path* to a log file for pipobot
- xmpp_logpath: a *relative or absolute* path to log all debug related to XMPP communication. This will only be useful in debug mode otherwise nothing will be logged (see *General command-line options*).
- force_ipv4: the bot will try to connect to the XMPP server in IPv6 (if a DNS record exists) if this *boolean* is not provided.
- lang: the language the application will use
- modules_path: a *list* of directories (relative or absolute) where the bot will try to find modules.

Database section

Configuration of the database which will be used by the bot. Supported engines are: *MySQL*, *PostgreSQL*, *SQLite*, with their respecting configuration.

SQLite configuration

These parameters are required with this engine:

- engine: must be *sqlite*
- src: a relative or absolute path to a file where the database will be stored

MySQL or PostgreSQL configuration

For those engines here are the required parameters:

- engine: *mysql* or *postgresql*
- user: username to access the database
- password: password to access the database
- host: the server where the database is hosted (can be hostname or IP)
- name: the name of the database on the server

Room section

A room is a reference to an XMPP MUC that the bot will join. It requires these parameters:

- chat: the MUC it will join (room@domain.tld).
- address: the address of the XMPP server (optional, default is extracted from the JID of the bot).
- port: the port of the XMPP server (optional, default is 5222).
- login: the JID used to authenticate to an XMPP server.
- passwd: the password used for the authentication.
- resource: the resource of the bot
- nick: the nickname of the bot in the MUC
- modules: a list of modules or groups to load for this room. Groups must be prefixed with an underscore.

Group section

A group is a list of modules we create that can be referenced in a room configuration (see the Room section above). example:

groups:

group1:

- module1
- module2

group2:

- module1
- module3

Then in the *modules* parameter of a room can add *_group1* or *_group2*.

Module-config section

In the *module_config* section you will define modules specific configuration. You can refer to the documentation of these modules to determine how to configure them, or if there is no such documentation to messages given by the bot when it is started: it will inform you that some configuration parameters are missing, or what default parameters are used instead.

Modules configuration are defined this way:

module_config: module_name:

param1: a single value

param2:

- a list
- of items

param3: key: value

Testing section

The *testing* section is what will define which parameters and which modules the bot will use when started in testing modes (see *Unit-test mode*). You will need to provide these parameters:

- fake_nick: a nickname for the bot.
- fake_chan: a fake chan name (like XMPP MUC name).
- modules: a list of modules, just like in a real room.

1.1.4 Invocation

***pipobot* can be started in serveral modes:**

- *XMPP* mode : this is the principal mode for the bot : it will connect to a Jabber MUC and start listening for commands.
- *Testing* modes : they do not require an XMPP server : they are provided in order to easily test modules and bot functionalities.

General command-line options

When you start the bot in *XMPP* mode, you can use these options (use `pipobot -h` to retrieve them):

```
--version         show program's version number and exit
-h, --help        show this help message and exit
-q, --quiet       Log and print only critical information
-d, --debug       Log and print debug messages
-b, --background  Run in background, with reduced privileges
--pid=PID_FILE    Specify a PID file (only used in background mode)
```

You can also always specify a configuration file (default being /etc/pipobot.conf.yml):

```
pipobot /path/to/alternative/config
```

Check-modules mode

In this mode the bot will only check the configuration file, check all modules and verify that you provided all required configuration parameters.

To use this mode use:

```
--check-modules   Checks if modules' configuration is correct
```

Unit-test mode

In this mode, unit test modules will be used and started to detect errors. It will use the `testing` section of the configuration file (see *Testing section*).

If you want to learn more about unit test, you can refer to *Unit Tests*.

To use this mode use:

```
--unit-test       Run unit test defined in the config file
```

Example:

```
pipobot --unit-test

test_todo_add (todo.TodoAdd)
!todo add ... ok
test_todo_remove (todo.TodoRemove)
!todo remove ... ok
test_search (todo.TodoSearch)
!todo search ... ok

----------------------------------------------------------------------
Ran 3 tests in 1.054s

OK
```

Script mode

This mode allows you to start the bot with a pre-defined list of commands. Commands are separated with a *;*. It will generate their outputs and display them to you. Example:

```
pipobot --script=":help;http://www.google.fr;:todo list all"

--> :help
<== I can execute:
-todo
--> http://www.google.fr
<== [Lien] Titre : Google
--> :todo list all
<== TODO-list vide
```

Interactive mode

This mode is provided to simulate an XMPP room locally. You can start the bot in this mode with:

```
pipobot --interract
```

Loaded modules will be those defined in the `testing` section of the configuration file (see *Testing section*). This will start a server waiting for fake XMPP clients to connect. To create a new client you can use the **pipobot-twisted** provided application:

```
pipobot-twisted foo
```

This will create a new client called *foo* connecting to the fake server. You can then enter your commands and see the result :

```
pipobot-twisted foo

Connected to server
Welcome !
*** foo has joined
!help
<foo> !help
<Pipo-test> I can execute:
-todo
!todo add liste un test
<foo> !todo add my_list a test
<Pipo-test> TODO added
!todo list
<foo> !todo list
<Pipo-test> All TODO-lists:
my_list
!todo list my_list
<foo> !todo list my_list
<Pipo-test> my_list :
1 - a test (by foo on 2012/03/10 at 16:20)
```

You can start multiple client to the room as long as they have different nicknames.

1.2 Modules API

1.2.1 Architecture of a module

To create a new module for *pipobot* you have to write some python classes which are subclasses of pre-defined type of modules. See below for the description of all different modules. A classic structure of a module (here *date*) is :

```
modules/date/
     __init__.py
     cmd_date.py
```

`cmd_date.py` will contain the *CmdDate* class defining the command. `__init__.py` will just have to contain 'from cmd_date import CmdDate' so importing modules.date will result to the import of the command class. You can add as many cmd_[name].py as your module requires commands. You just have to import them all in `__init__.py`.

Types of modules

There are several classes of modules, depending on what you are trying to achieve.

SyncModule

A *SyncModule* is a module that can be called *explicitly* by a user (Sync stands for Synchronous). It can be used in a room like this :

```
<user> !date
<bot> Today is `insert the date of the day here !`
```

For more details, see *SyncModule*.

MultiSyncModule

A *MultiSyncModule* is very similar to a *SyncModule*, except that one *MultiSyncModule* can handle several commands in it. This is quite useful when commands are very simple, and does not require python code to be handled.

For more details, see *MultiSyncModule*.

AsyncModule

An *AsyncModule* is a module which is not related to anything said in the room. For instance, it could be a module announcing the hour every hour, or analysing mails from a mail server and announcing new messages in the room.

For more details, see *AsyncModule*.

ListenModule

A *ListenModule* is a module where the bot reacts to something that has been said in the room, but without an explicit call of a command, as in :

```
<user> Here is an awesome link : http://pipobot.xouillet.info !
<bot> [Lien] Titre : Forge xouillet
```

Every message in a room can be analysed by the bot, and he can react if the message matches some criteria (contains a URL for instance).

For more details, see *ListenModule*.

PresenceModule

A *PresenceModule* reacts to every presence message in a room, for instance joins and leaves of users. For instance:

```
*** user has joined the room
<bot> user: welcome to the room !!
```

For more details, see *PresenceModule*, or the *User Monitoring Module* which is a *PresenceModule*.

What they can return

A string

If a module returns a string, the bot will simply say it in the room.

A list of strings

If a module returns a list of string, the bot will say each element of the list one after the other. Example:

```
def some_function(self, sender, message):
    return ["msg1", "msg2", "msg3"]
```

will result to:

```
<bot> msg1
<bot> msg2
<bot> msg3
```

A dictionary

Thanks to XEP-0071, XMPP protocols allows to send XHTML messages for clients that support it. If you want your module to send XHTML messages, you can make it return a dictionary like :

```
return {"text" : "*Message for clients which don't support XHTML*",
        "xhtml" : "<b>Message for clients which do support XHTML</b>"
        }
```

Some clients do not handle monospace fonts, so if you want to had some presentation in your messages (tabulars for instance) they will not render correctly. If those clients support XHTML messages, you can create an XHTML message that will do it :

```
raw_msg =  "| Some       | tabular   |\n"
raw_msg += "| requiring  | monospace |"
return {"text" : raw_msg,
        "monospace" : True}
```

The following XHTML message will be automatically created and sent :

```
<p>
    <span style="font-family: monospace">
        | Some       | tabular   | <br />
        | requiring  | monospace |
    </span>
</p>
```

Finally, dictionaries can be used to send private message to several users. Example:

```
return { "user1" : { "text": "Message for user1",
                     "monospace": True },
         "user2" : { "text" : "raw message for user2",
                     "xhtml" : "<p> an <b> XHTML </b> message for user2 </p>"}
        }
```

Nothing, None or “”

If a module has no return statement, returns None or “”, then the bot will simply not say anything.

Using configuration parameters

Some modules may require configuration parameters that will be provided by the `pipobot`‘s main configuration file.

`pipobot` includes a syntax to define such parameters, and will automatically:

- check if required parameters are present
- replace optional parameters by a default value
- check if provided parameters are correct (type verification)

To add parameters to a module you must provide a *_config* attribute to the module class, listing them. For example if we want a module to parse the several sample of configuration:

```
modules_config:
    my_module:
        param1: True
        param2:
            - foo
            - bar
        param3:
            key1: val1
            key2: val2
        # OPTIONAL
        param4: "somestring"
```

In the corresponding module class we will add:

```
class MyModule(SyncModule):
    _config = (("param1", bool, None), ("param2", list, None),
               ("param2", dict, None), ("param4", string, "somestring))
```

Then in the code of the module we will be able to access to these parameters with *self.param1*, *self.param2*...

Possible types of parameters are defined by the yaml language:

- a boolean
- a string
- an int
- a list
- a dictionary

Each element of the _config array is a parameter constructed with (name, type, default_value), None in default_value meaning that the parameter is not optional.

1.2.2 Specific description of modules

SyncModule

Definition of module

A *SyncModule* is a module that can be called *explicitly* by a user (Sync stands for Synchronous). It can be used in a room like this :

```
<user> !date
<bot> Today is `insert the date of the day here !`
```

Some parameters must be specified to define a command :

- *name* : its name (*date* in the previous example)

- *desc* : a description of the module which will be used by the *help* module (see *Description format*.)

Writing handlers

SyncModule mother class implements a parsing method for commands. For instance a command can take several subcommands as in this example:

```
<user> !todo
<bot> This is a command to handle TODO-list
<user> !todo list
<bot> Here is the list of all TODO : ...
<user> !todo add some_list I have TODO this !
<bot> The todo 'I have TODO this !' has been successfully added to 'some_list'
```

list and *add* are subcommands for the main **todo** command. To each subcommand you want to define, you have to write a handler to the module class.

A handler is a Python method with this signature:

```
def some_name(self, sender, message):
```

The parameters are :

- *sender* is the name of the user who sent the command (*user* in the previous example).
- *message* is what the user sent, without the command name and the subcommand name.

For instance in:

```
<user> !todo add some_list I have TODO this !
```

sender will be *user* and *message* will be *some_list I have TODO this !*.

In order to define a subcommand, you have to add a descriptor to the method you write. It can be `@defaultcmd` or `@answercmd("subcommand1", "subcommand2")`. For instance the skeleton of the **todo** module will be:

```
from lib.modules import SyncModule, answercmd, defaultcmd

class CmdTodo(SyncModule):
    def __init__(self, bot):
        desc = "A TODO module"
        command_name = "todo"
        SyncModule.__init__(self, bot, desc, command_name)

    @answercmd("add")
    def add(self, sender, args):
        #what to do with !todo add some other args
        pass

    @answercmd("list")
    def list(self, sender, args):
        #what to do with !todo list some other args
        pass

    @answercmd("rm", "del")
    def rm(self, sender, args):
        #what to do with !todo rm or !todo del  some other args
        pass

    @defaultcmd
```

```
    def default(self, sender, message):
        #In any other case this will be called
        pass
```

The `@defaultcmd` decorator specify the method that will be called when *no other method* corresponds to user's input. For instance in this example, all these calls will be handled by the *default* method:

```
!todo
!todo should RTFM
!todo don't know what i am doing
```

This behaviour is interesting if you want to handle errors yourself : any use of the command that is not conform to the syntax defined by other decorators will be handled by the `default` method.

Finally you can use regular expressions in decorators to filter subcommands differently. For instance we can re-write the **todo** module like this:

```
class CmdTodo(SyncModule):
    def __init__(self, bot):
        pass

    @answercmd("^$")
    def empty(self, sender, args):
        pass

    @answercmd("list"):
    def list(self, sender, args):
        pass

    @answercmd("add (?P<list_name>\S+) (?P<desc>.*)"=
    def add(self, sender, args):
        liste = args.group("list_name")
        desc = args.group("desc")

    @answercmd("(remove|delete) (?P<ids>(\d+,?)+)")
    def remove(self, sender, args):
        ids = args.group("ids").split(",")
```

As you can see in this example, with this syntax you can do a lot of work to filter commands directly in the decorator. In the previous example, a call like :

```
!todo add somelist a new todo to add
```

will be handled by the `add` method, and a call like :

```
!todo remove 1,2,3
```

will be handled by the `remove` method.

Empty call like :

```
!todo
```

will be handled by the `empty` method.

Finally any other syntax will raise an error so the bot will return a message recommending to read the manual of the command since no `@defaultcmd` is provided.

You can use in a given module regular expression-based decorators and "classic" decorators. Just be careful of the behaviour if for instance some regular expressions are to permissive.

WARNING: Be careful not to use too permissive pattern in `@answercmd` decorator. For instance if you use this set of decorators :

```
@anwsercmd("add (?P<list_name>\S+) (?P<desc>.*)")
@answercmd("search (?P<query>.*)")
@answercmd("(remove|delete) (?P<ids>(\d+,?)+)")
@answercmd("")
```

ANY call to the corresponding command will be caught by the last one since an empty regular expression matches *a lot* of things !! If you want to define the *empty* subcommand, just use `@answercmd("^$")`.

MultiSyncModule

A *MultiSyncModule* is similar to a *SyncModule* but it contains several commands which will be handled by the same module. You initialize it with a dictionary command_name → command_description. Then you will provide some handling method with the same syntax as you would in a *SyncModule*.

AsyncModule

An *AsyncModule* is a module executing a task automatically every *n* seconds and send a message in a room with the result of this task. Its action is not related to anything said in the room.

Example:

```
<bot> You have received a new mail !!!
```

Additionally to the name and the description of the module (see *Description format*) you have to provide a *delay* which means : every *delay* seconds the action will be executed. Then you write an *action* function with no argument :

```
def action(self):
    #some_work
    self.bot.say("The message we send to the room")
```

action is the method that will be called every *delay* seconds.

ListenModule

An *ListenModule* is a module executing a task which depend on something that has been said in the room. But as opposed to *SyncModule* it is not explicitly called with a *!command* syntax.

For instance, it can be used to analyse messages with URL :

```
<user> hey, check this amazing link : http://www.nojhan.net/geekscottes/strips/geekscottes_103.png
<bot> [Lien] Type: image/png, Taille : 68270 octets
```

The parameters required for a *ListenModule* are:

- its name
- a description (see *Description format*)

The *answer* handler function will have this signature:

```
def answer(self, sender, message):
    #some work on the message
    if (re.findall(SOME_URL_REGEXP, message)):
        #handle url
        return "[Lien] Type: %s, Taille : %s octets" % (ctype, clength))
```

```
    else:
        return None
```

Then if the message contains an URL you can extract it, work on it and return some information about it. If it does not, you return *None* so the bot will not say anything in the room.

PresenceModule

A *PresenceModule* is handling XMPP Presence stanza, for instance in a MUC : an user joins/leaves the room. The handling method is named *do_answer* with this signature:

```
def do_answer(self, message):
    # some work on the message
    if join_message:
        self.bot.say("Hello %s !" % username)
```

Which will result in:

```
*** user has joined
<bot> Hello user !!!
```

1.2.3 Some internal modules

Help Module

Description format

User Monitoring Module

1.2.4 High-Level Modules

These modules are derived from `general` module presented here : *SyncModule*. They exist to simplify writing some modules executing similar tasks.

FortuneModule

This module is a `SyncModule` with some pre-defined functions. It can be used in this context : you have a website presenting some quote/fortunes and you want to write a module which, when called, will parse quotes from the website and return it. In addition to all `SyncModule` parameters, it has two more attributes you have to set : `url_random` and `url_indexed`. It provides commands with the syntax:

```
!cmd
!cmd some_number
```

In the first case, the module will use the `url_random`, and parse it. In the second case, the module will use the `url_indexed`, insert in it `some_number`, and get the corresponding page. All you need to do in your module is to override the `extract_data`, method using with your own, using the `soup` parameter which is a BeautifulSoup object created with the content of the page.

You can see some example of such `FortuneModule` in the bot (bashfr, vdm, chuck, ...).

NotifyModule

This module is the combination of a `SyncModule` and an `AsyncModule`. You have to define a `do_action` method that will be called every *n* seconds. In a `NotifyModule`, the `action` method (see *AsyncModule* for more details) is already defined and will check if the module has been *muted* or not. If it has not, the method `do_action` that you are supposed to write will be called. The `NotifyModule` will provide a `mute/unmute` method that will disable/enable the notifications. You can add to it as many `@answercmd` as you need to, like in any other `SyncModule`.

The `reminder` module is an example of such module.

1.3 Writing unit-test

1.3.1 Unit Tests

Unit tests in `pipobot` module are based on the python `unittest` library. On top of that library, a `ModuleTest` class has been written to provide some bot-related functionalities.

For more information about the ModuleTest class see *module_test Module*.

Write a ModuleTest class

Some tools are provided to you to write a unit test. First you can use the *bot_answer* method that will take a string defining what is the message that must be analysed by the bot, and retuning its answer. Then you can use *unittest* functionalities to check if the result is correct. Each test is a method that must be prefixed with *test_* Here is a first simple example:

```
class BandMTest(ModuleTest):
    def test_current(self):
        """ !b&m : check current song """
        bot_rep = self.bot_answer("!b&m")
        self.assertRegexpMatches(bot_rep, "Titre en cours : (.*)")

    def test_lyrics(self):
        """ !b&m lyrics """
        self.bot_answer("!b&m lyrics")
```

This ModuleTest contains 2 unit test : the first is asking the bot *!b&m* and expects in return a result matching a regular expression. The second test is asking *!b&m lyrics* and has no test on the output : it will only fail an exception is raised.

You can also create more complicated example : for instance to test modules that need to access to database elements :

```
class TodoRemove(ModuleTest):
    def setUp(self):
        """ Creates 3 random todo we add manually to the database """
        self.todos = []
        todos = {string_gen(8): string_gen(50),
                 string_gen(8): string_gen(50),
                 string_gen(8): string_gen(50)}
        for list_name, todo in todos.iteritems():
            todo = Todo(list_name, todo, "sender", time.time())
            self.bot.session.add(todo)
            self.bot.session.commit()
            self.todos.append(todo)
```

```
    def test_todo_remove(self):
        """ !todo remove """
        bot_rep = self.bot_answer("!todo remove %s" % ",".join([str(elt.id) for elt in self.todos]))
        expected = "\n".join(["%s a été supprimé" % todo for todo in self.todos])
        self.assertEqual(bot_rep, expected)

    def tearDown(self):
        """ In case of failure, we manually remove the todo we added """
        for todo in self.todos:
            remove = self.bot.session.query(Todo).filter(Todo.id == todo.id).first()
            if remove is not None:
                self.bot.session.delete(remove)
                self.bot.session.commit()
```

In this class, we are only testing one command with the method *test_todo_remove*. The setUp and tearDown methods are defined in the python *unittest* API :

- setUp is executed *before* the actual test
- tearDown is executed *after* the test

In the test we want to try the deletion of todo, with the "!todo remove id1,id2,id3" command. So in the setUp we manually create 3 todos with random values. Then in the test we try to remove them. The tearDown is usefull in case of the test fails : it manually removes todo added in the setUp, so we are sure that even if the test fails we will not have any generated todo remaining in the database.

Run your tests

To ask the bot to run the test you have just created, use the *–unit-test* option of pipobot, as described here: *Unit-test mode*.

1.4 Internationalisation and Localisation

1.4.1 Internationalisation

Pipobot uses the *gettext* module for internationalisation purposes. You can use the following functions to render your module translatable.

gettext(*string*)
_(*string*)

These functions take a string in argument and return the translated string. Sample usage:

```
    self.bot.say(_("Hello, World!"))
```

When one of these two functions are used, the string passed in parameter will be automatically proposed for translation.

If you want to translate a string format, pass only the format to the function:

```
    self.bot.say(_("Hello, %s") % name)
```

If you have more than one format parameter, it is better to name them explicitely because the translator may want to reverse the order:

```
    self.bot.say(_("Today is %(month)s, %(day)d") % {'month': month,
        'day': day})
```

ngettext (*singular*, *plural*, *n*)
ngettext is used to translate expressions which can be pluralised. Sample usage:

```
self.bot.say(ngettext("You have %d message", "You have %d messages",
    message_count) % message_count)
```

Always use ngettext instead of `if message_count == 1: ...` because some languages have pluralization rules different from English (for instance, in French, *0* is singular, not plural, and in Polish, there are 5 different plural forms depending on the item count)

N_ (*string*)
N_ is a no-op. It just returns the string passed in parameter. It is used to mark strings which should be translatable but cannot be directly translated because the translation system is not already active (so _, gettext and ngettext are unavailable). That may be the case for strings defined as constants in a Python module or as a class attribute.

For instance:

```
HELLO_MESSAGE = N_("Hello, World!")
[...]
def say_hello():
    print _(HELLO_MESSAGE)
```

You do not need to import anything to use these functions: they are always defined at the global level.

1.4.2 Translation handling

Pipobot uses the babel module to handle translations. If you intend to add new translations or update existing ones, you will need to install this module.

New language

To translate Pipobot to a new language (for instance *zz_ZZ*, use the following commands (from the directory containing the setup.py script):

```
python setup.py extract_messages # Extract the messages from Pipobot's sources
python setup.py init_catalog -l zz_ZZ # Create a translation catalog for the specified language
```

You can now edit pipobot/i18n/zz_ZZ/LC_MESSAGES/pipobot.po (with a standard text editor or POEdit, for instance) and translate every message. When done, run:

```
python setup.py compile_catalog # Compile the translation catalog
```

The translation can now be used by Pipobot.

Update an existing language

To update an existing translation catalog in order to take into account the changes in Pipobot's source code, run the following commands (replace *zz_ZZ* with the name of the catalog you want to update):

```
python setup.py extract_messages # Extract the messages from Pipobot's sources
python setup.py update_catalog -l zz_ZZ # Update the translation catalog for the specified language
```

You can now edit pipobot/i18n/zz_ZZ/LC_MESSAGES/pipobot.po (with a standard text editor or POEdit, for instance) and translate every message. When done, run:

```
python setup.py compile_catalog # Compile the translation catalog
```

The translation can now be used by Pipobot.

CHAPTER 2

Code documentation

2.1 Pipobot Package

2.1.1 config Module

2.1.2 bot Module

2.1.3 bot_jabber Module

2.1.4 bot_test Module

2.1.5 bot_twisted Module

2.1.6 Subpackages

lib Package

bdd Module

exceptions Module

modules Module

abstract_modules Module

parsedates Module

user Module

known_users Module

loader Module

utils Module

module_test Module

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Symbols

G

N