helga Documentation

Release stable

January 01, 2015

Contents

CHAPTER 1

About

A python-based IRC bot using Twisted. Original inspiration came from olga. Why re-implement another bot? Because olga is written in perl, and I wanted something a bit more sane to look at.

CHAPTER 2

Requirements

All requirements for helga are listed in `requirements.txt`. However, there is a single external requirement, and that is MongoDB. You don't need it, per se, but many of the included plugins use MongoDB for storing consistent state between restarts.

CHAPTER 3

Getting Started

Start by creating a virtualenv where helga will reside:

```
$ virtualenv helga
$ cd helga
$ source bin/activate
```

Then grab the latest copy and install requirements:

```
$ git clone https://github.com/shaunduncan/helga src/helga
$ cd src/helga
$ python setup.py develop
```

Once you have performed the above steps, there will be a `helga` executable placed in the `bin` dir of your virtualenv. Run helga by calling this:

```
$ helga
```

Note that this uses the default settings file, `helga.settings` to start. You can, and should, use your own custom setttings. The behavior of a custom settings file is to preserve defaults in helga.settings and apply overrides. For this reason, you do not need to apply all configuration settings known to helga. To use a custom settings file, either export an environment variable `HELGA_SETTINGS` or use the `--settings` argument:

```
$ helga --settings=foo.bar
```

Note that this should be a string that is either an importable python module like `foo.bar.baz` or a filesystem path like `/foo/bar/baz.py`.

CHAPTER 4

Default Settings

As mentioned above, there is a default `helga.settings` module. This contains some basic helga settings, as outlined below:

- `SERVER`: A dictionary containing server connection info. At minimum, keys 'HOST' and 'PORT' are required and default to 'localhost' and '6667' respectively. Optional keys include 'SSL' which if True, will connect to 'HOST' using SSL, and 'USERNAME' and 'PASSWORD' if the IRC server requires authentication.
- `LOG_LEVEL`: String for the default logging level (default: 'DEBUG')
- `LOG_FILE`: If set, a string indicating the log file for python logs
- `PLUGIN_PRIORITY_LOW`: The value for 'low' priority plugins (see "Plugin Priorities" below). Default 25.
- `PLUGIN_PRIORITY_NORMAL`: The value for 'normal' priority plugins (see "Plugin Priorities" below). Default 50.
- `PLUGIN_PRIORITY_HIGH`: The value for 'high' priority plugins (see "Plugin Priorities" below). Default 75.
- `COMMAND_ARGS_SHLEX`: Control the behavior of argument parsing for command plugins By default this is a naive str.split(' '), however a plugin may need this behavior to be a bit more robust. By setting this value to True, shlex.split() will be used instead so that commands like `helga foo bar "baz qux"` will yield an argument list like ['bar', 'baz qux'] instead of ['bar', "'baz', 'qux"']. Shlex splits will be the default and only supported behavior in a future version. This can also be used per plugin by passing `shlex=True` to an `@command` decorator (described below in "Plugin Types")
- `CHANNEL_LOGGING`: If True, enable conversation logging on all channels (default: False)
- `CHANNEL_LOGGING_DIR`: If using channel logs, the directory to which channel logs should logs should be written. A new directory will be created for each channel in which the bot resides, so if this is set to '/foo/bar' logs for channel '#baz' will be created in '/foo/bar/#baz'. (default: '.logs')
- `CHANNEL_LOGGING_HIDE_CHANNELS`: A list of channel names (either with or without a '#' prefix) that will be hidden in the channel log browser web ui. They will still be accessible via direct URL access, but they will not immediately be shown on the full channel list.
- `NICK`: The default nick of the bot instance (default: 'helga')
- `CHANNELS`: A list of channels to automatically join. You can specify either a single channel name or a two-tuple of channel name, and password (default: ['#bots'])
- `AUTO_RECONNECT`: Should the bot automatically reconnect on connection lost? (default: True)
- `AUTO_RECONNECT_DELAY`: Time, in seconds, between reconnect attempts (default: 5)
- `RATE_LIMIT`: Message rate limit for messages sent over IRC. Default is None implying no limit.
- `OPERATORS`: List of IRC nicks that should be considered operators/administrators.

- DATABASE: A dictionary containing connection info for MongoDB. The minimum settings that should exist here are 'HOST', the MongoDB host, 'PORT, the MongoDB port, and 'DB' which should be the MongoDB database to use. These values default to 'localhost', 27017, and 'helga' respectively without any overrides. Both 'USERNAME' and 'PASSWORD' can be specified if MongoDB requires authentication.
- TIMEZONE: The default timezone for the bot instance (default: 'US/Eastern')
- ENABLED_PLUGINS: A list of plugin names that should be enabled automatically for any channel. Note that this does not mean plugins that are loaded. By default, any plugin that has been installed will be loaded and made available. This should be a list of the entry point names defined by each plugin. See below for information about this.
- ENABLED_WEBHOOKS: A list of webhook names that should be enabled on process startup. If this value is None, then all webhooks available are loaded via entry points. An empty liste will not load any webhooks. Default is None.
- PLUGIN_FIRST_RESPONDER_ONLY: If True, only the first plugin that generates a response will be sent back via IRC. If False, all plugin responses are sent. (default: True)
- COMMAND_PREFIX_BOTNICK: If set to True, command plugins can be run by asking directly, such as 'helga foo_command'. (default: True)
- COMMAND_PREFIX_CHAR: If non-empty, this char can be used to invoke a command without requiring the bot's nick. For example 'helga foo' could be run with '!foo'. (default: '!')
- FACTS_REQUIRE_NICKNAME: Boolean, if True, would require the bot's nick to show a stored fact. For example, if True, 'foo?' could only be shown with 'helga foo?'. (default: False)
- JIRA_URL: A URL format for showing JIRA links. This should contain a format parameter '{ticket}'. (default: 'http://localhost/{ticket}')
- JIRA_REST_API: If non-empty, this should be the URL for a JIRA REST API for the JIRA plugin to use. Must like JIRA_URL, this should contain a format parameter '{ticket}'. Note that this requires a minmum JIRA version to work, one that has the updated REST api. See https://docs.atlassian.com/software/jira/docs/api/REST/latest/.
- JIRA_SHOW_FULL_DESCRIPTION: Boolean, if False, only the formatted JIRA_URL will be returned. If True, a full ticket title will be shown. This requires JIRA_REST_API to be set. (default: False)
- JIRA_AUTH: A two-tuple of JIRA credentials, username and password. (default: ('', ''))
- REVIEWBOARD_URL: A URL format for showing ReviewBoard links. This should contain a format parameter '{review}'. (default: 'http://localhost/{review}')
- WIKI_URL: A URL format for showing user pages on a wiki, such as example.com/~user. This should contian a format parameter '{user}'. (default: 'http://localhost/{user}')
- WEBHOOKS_PORT: The port the webhooks plugin should listen for http requests. (default: 8080)
- WEBHOOKS_CREDENTIALS: List of two-tuple username and passwords used for http basic authentication. (default: none).

4.1 Local Development

The included Vagrantfile will let you spin up a VM to run both MongoDB and an IRC server for local development. Once you've followed the previous instructions for installing helga, simply `vagrant up`. This will forward host ports 6667 (irc) and 27017 (mongo) to the guest. At this point, simply runing `helga` from the command line will connect to this VM.

CHAPTER 5

Plugins

5.1 Overview

Helga supports plugins outside of the core source code. Plugins have a minimal API, but there are some basic rules that should be followed. All core plugin implementations can be found in `helga.plugins.core`. The basic requirement for plugins is that they have a `process` attribute that is a callable and determines if the plugin should handle a message, and a `run` method that actually performs the legwork of what the plugin should do. By convention, the `process` method should accept four arguments:

- **client**: an instance of `helga.comm.Client`
- **channel**: the channel on which the message was received
- **nick**: the current nick of the message sender
- **message**: the message string itself

The `run` is a bit different as it is up to the plugin implementation itself to decide what arguments are necessary to generate a response. This method should be called by `process` and should return one of:

- None or empty string, if no response is to be sent over IRC
- Non-empty string for a single line response
- List of strings for multiline responses

Really, as long as you follow the above conventions, you can write plugins however you wish. However, you should try to keep plugins simple and use the included decorators `command`, `match`, and `preprocessor` (explained later). However, if you prefer writing a plugin as a class, you can subclass the included `Plugin` base class, provided you have followed the above rules. Here is a simple example:

```
import time
from helga.plugins.core import Plugin

class MyPlugin(Plugin):
    def run(self, channel, nick, message):
        return u'Current timestamp: {0}'.format(time.time())

    def process(self, channel, nick, message):
        if message.startswith('!time'):
            return self.run(channel, nick, message)
```

NOTE the previous example is not the preferred way. You should use the included decorators instead (shown below).

5.1.1 A Tale of Unicode

Plugins should try to deal with unicode as much as possible. This is important as all arguments a plugin receives will be unicode strings and not byte strings. This process happens automatically as all strings received over IRC are decoded as UTF-8 an converted to unicode. If a plugin returns a string response that is unicode, it will be encoded as UTF-8 prior to being sent over IRC. To help deal with this, there are two helpful methods in `helga.util.encodings` to convert to/from unicode: `to_unicode` and `from_unicode`.

5.2 Plugin Types

For the most part, there are two main types of plugins: commands and matches. Commands are plugins that require a user to specifically ask for helga to perform some action. For example, `helga haiku` or `helga google something to search`. Matches are on the other hand are intended to be autoresponders that give some extra meaning or context to what a user has said. For example, if helga matches for a string "foo":

> <sduncan> i'm talking about foo in this message <helga> sduncan is talking about foo

For the sake of simplicity, there are two convenient decorators for authoring these types of plugins (which is usually the case). For example:

```
from helga.plugins import command, match

@command('foo', aliases=['foobar'], help="The foo command")
def foo(client, channel, nick, message, cmd, args):
    # This is run on "helga foo" or "helga foobar"
    return u"Running the foo command"

@match(r'bar')
def bar(client, channel, nick, message, matches):
    # This will run whenever a user mentions the word 'bar'
    return u"{0} said bar!".format(nick)
```

You may notice in the above example that each decorated function accepts different arguments. For commands, there are two additional arguments `cmd` and `args`. The former is the parsed command that was used to run the method (which could be "foo" in the above case, or the alias "foobar"). The latter is a list of whitespace delimited strings that follow the parsed commend. For example `helga foo a b c` would mean the args param would be `['a', 'b', 'c']`.

For the match plugin, the single additional argument is `matches` which is for the most part, the result of `re.findall`. However, the `@match` decorator accepts a callable in place of a regex string. This callable should accept one argument: the message being processed. It should return a value that can be evaluated for truthiness and will be passed to the decorated function as the `matches` parameter.

Command plugins currently have multiple ways of parsing argument strings. The old way that will be deprecated in a future version is naive whitespace splitting. For example, the command `helga foo bar "baz qux"` would produce args `['bar', '"baz', 'qux"']`. This may not be ideal for some plugins. To get around this, you can optionally pass `shlex=True` to your command decorator like `@command('foo', shlex=True)`. With this enabled, the command `helga foo bar "baz qux"` would produce args `['bar', 'baz qux']`. This behavior will become the default in a future version. You can also enable this behavior globally by setting COMMAND_ARGS_SHLEX to True in your settings file.

5.3 Preprocessors

Plugins can also be message preprocessors. These are callables that may perform some modification on an incoming message prior to that message being delivered to any plugins. Preprocessors should accept arguments (in order) for `client`, `channel`, `nick`, and `message` and should return a three-tuple consisting of (in order) `channel`, `nick`, and `message`. To declare a function as a preprocessor, a convenient decorator can be used:

```
from helga.plugins import preprocessor

@preprocessor
def blank_message(client, channel, nick, message):
    return channel, nick, u''
```

5.4 Complex plugins

Some plugins do both matching and act as a command. For this reason, plugin decorators are chainable. However, remember that different plugin types expect decorated functions to accept different arguments. It is best to accept `*args` for these:

```
from helga.plugins import command, match, preprocessor

@preprocessor
@match(r'bar')
@command('foo')
def complex(client, channel, nick, message, *args):
    # len(args) == 0 for preprocessors
    # len(args) == 1 for matches
    # len(args) == 2 for commands
```

5.5 Plugin Priorities

You can control the priority in which a plugin is run. Note though, that preprocessors will always run first. A priority value should be an integer value. There are no limits or bounds for this value, but know that a higher value will mean a higher priority. If you are writing `Plugin` subclass style plugins, you will need to set a `priority` attribute of your object. This is done automatically if you call `super(MyClass, self).__init__(priority=some_value)` in your class's `__init__`.

However, if you are using the preferred decorator style for writing plugins, you can supply a `priority` keyword argument to the decorator:

```
from helga import command, match, preprocessor

@preprocessor(priority=10)
def foo_preprocess(*args):
    pass

@command('foo', priority=20)
def foo_command(*args):
    pass

@match(r'foo', priority=30)
def foo_match(*args):
    pass
```

For convenience, there are constants that can be used for setting priorities:

- **PRIORITY_LOW** = 25
- **PRIORITY_NORMAL** = 50
- **PRIORITY_HIGH** = 75

Also, each decorator/plugin type has its own default value for priority:

- Preprocessors have default priority of `PRIORITY_NORMAL`
- Commands have default priority of `PRIORITY_NORMAL`
- Matches have default priority of `PRIORITY_LOW`

The above priority values are defaults and can be tuned if necessary via settings overrides (see default settings above).

5.6 Publishing plugins

Helga uses setuptools entry points for plugin loading. Once you've written a plugin you wish to use, you will need to make sure your python package's setup.py contains an entry_point under the group name `helga_plugins`. For example:

```
entry_points = {
    'helga_plugins': [
        'plugin_name = mylib.mymodule:MyPluginClass',
    ],
},
```

Note that if you are using decorated function for a plugin, you will want to specify the method name for your entry point, i.e. `mylib.mymodule:myfn`.

5.7 Webhooks

As of helga version 1.3, there is an included plugin for exposing an HTTP server to support webhooks. This might be useful if you need to have a public facing HTTP service that you would like to use to perform some sort of announcement on a particular channel. This is also very extensible and should allow you to create new webhooks in a very similar way plugins are created. This plugin is enabled by default and requires two settings: `WEBHOOKS_PORT` and `WEBHOOKS_CREDENTIALS`. The former is of course the port on which to run this service. The latter should be a list of tuples in the form of (username, password). These are used to perform HTTP basic authentication on any webhook that requires it.

Webhook plugins work by declaring routes. This will not only feel similar to helga's decorator style plugins, but it will also feel very similar to anyone who has used something like Flask. There are two primary decorators you will need to get started: `route`, which declares a function as a route endpoint, and `authenticated`, which ensures that the route function cannot be called without proper HTTP basic authentication. Both of these can be imported from `helga.plugins.webhooks`. For example:

```
from helga.plugins.webhooks import authenticated, route

@route(r'/foo/(?P<id>[0-9]+)')
@authenticated
def foo(request, irc_client, id):
    # This will require auth
    pass
```

```
@route('/bar', methods=['POST'])
def bar(request, irc_client):
    # This will not require auth, and will only accept POST
    pass
```

NOTE: For authenticated routes, you MUST specify `@authenticated` as the first decorator. This may be changed in the future.

The route decorator accepts two arguments: 1) a path regular expression and 2) an optional list of HTTP methods to accept. If you do not specify a list of HTTP methods, only GET requests will be served. All regex paths must be named groups and they will be passed as keyword arguments.

Webhooks should return a string response, which will be returned to the requesting client as the content body of the response. You can arbitrarily set response headers using the passed `request` argument. For example:

```
def foo(request, irc_client):
    request.setResponseHeader(404)
    return '404 Not Found'
```

For convenience, if you would like to simplify settings resopnse status codes with an optional message, simply raise `helga.plugins.webhooks.HttpError`:

```
from helga.plugins.webhooks import HttpError

def foo(request, irc_client):
    raise HttpError(404)
```

To register a new webhook plugin, you must declare an entry_point much in the same way normal plugins are done. However, the entry_point group name is `helga_webhooks`. For example:

```
entry_points = {
    'helga_webhooks': [
        'name = mylib.mymodule:myhook',
    ],
},
```

The webhook plugin itself has some commands for IRC interaction: start/stop to control the running HTTP listener, and routes, which will show all the route paths and the HTTP methods they accept.

5.8 Third Party Plugins

Here are some plugins that have been written that you can use:

Plugin	Description	Link
excuses	Generate a response from http://developerexcuses.com	https://github.com/alfredodeza/helga-excuses
haskell	Evaluate Haskell expressions.	https://github.com/carymrobbins/helga-haskell
isup	Check downforeveryoneorjustme.com	https://github.com/shaunduncan/helga-isup
karma	Dish out karma points to other people	https://github.com/coddingtonbear/helga-karma
norris	Generate Chuck Norris facts for users	https://github.com/alfredodeza/helga-norris
updates	List and record IRC channel updates.	https://github.com/cobbdb/helga-contrib-updates
zen	The Zen of Python	https://github.com/shaunduncan/helga-zen

Written a plugin? Send a pull request to be listed in the above table!

CHAPTER 6

Tests

All tests are written to be run via `tox`. To run the test suite, inside your virtualenv:

```
$ cd src/helga
$ tox
```

CHAPTER 7

Contributing

Contributions are welcomed, as well as any bug reports! Please note that any pull request will be denied if tests run via tox do not pass

CHAPTER 8

License

Dual licensed under the MIT and GPL licenses.