helga Documentation

Release 1.7.5

Shaun Duncan

Aug 12, 2017

Contents

CHAPTER 1

About

Helga is a full-featured chat bot for Python 2.6/2.7 using Twisted. Helga originally started as a python fork of a perl-based IRC bot olga, but has grown considerably since then. Early versions limited to support to IRC, but now include other services like XMPP and HipChat.

CHAPTER 2

Supported Backends

As of version 1.7.0, helga supports IRC, XMPP, and HipChat out of the box. Note, however, that helga originally started as an IRC bot, so much of the terminology will reflect that. The current status of XMPP and HipChat support is very limited and somewhat beta. In the future, helga may have a much more robust and pluggable backend system to allow connections to any number of chat services.

CHAPTER 3

Features

- A simple and robust plugin api
- HTTP webhooks support and webhook plugins
- Channel logging and browsable web UI
- Event driven behavior for plugins
- Support for IRC, XMPP, and HipChat

CHAPTER 4

Contributing

Contributions are **always** welcomed, whether they be in the form of bug fixes, enhancements, or just bug reports. To report any issues, please create a ticket on github. For code changes, please note that any pull request will be denied a merge if the test suite fails.

If you are looking to get help with helga, join the #helgabot IRC channel on freenode.

CHAPTER 5

License

Helga is open source software, dual licensed under the MIT and GPL licenses. Dual licensing was chosen for this project so that plugin authors can create plugins under their choice of license that is compatible with this project.

CHAPTER 6

Contents

Getting Started

Requirements

All python requirements for running helga are listed in `requirements.txt`. Helga supports SSL connections to a chat server (currently IRC, XMPP, and HipChat); in order to compile SSL support, you will need to install both `openssl` and `libssl-dev` packages, as well as `libffi6` and `libffi-dev` (the latter are required for `cffi`, needed by `pyOpenSSL` version 0.14 or later).

Optionally, you can have Helga configured to connect to a MongoDB server. Although this is not strictly required, many plugins require a connection to operate, so it is highly recommended. "Why MongoDB", you ask? Since MongoDB is a document store, it is much more flexible for changing schema definitions within plugins. This completely eliminates the need for Helga to manage schema migrations for different plugin versions.

Important: Helga is currently **only** supported and tested for Python versions 2.6 and 2.7

Installing

Helga is hosted in PyPI. For the latest version, simply install:

Note, that if you follow the development instructions below and wish to install helga in a virtualenv, you will need to activate it prior to installing helga using pip. In the future, there may be a collection of .rpm or .deb packages for specific systems, but for now, pip is the only supported means of install.

Development Setup

To setup helga for development, start by creating a virtualenv and activating it:

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

Then grab the latest copy of the helga source:

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

Installing helga this way creates a `helga` console script in the virtualenv's `bin` directory that will start the helga process. Run it like this:

```
$ helga
```

Using Vagrant

Alternatively, if you would like to setup helga to run entirely in a virtual machine, there is a Vagrantfile for you:

```
$ git clone https://github.com/shaunduncan/helga
$ cd helga
$ vagrant up
```

This will provision an ubuntu 12.04 virtual machine with helga fully installed. It will also ensure that IRC and MongoDB servers are running as well. The VM will have ports 6667 and 27017 for IRC and MongoDB respectively forwarded from the host machine, as well as private network IP 192.168.10.101. Once this VM is up and running, simply:

```
$ vagrant ssh
$ helga
```

The source directory includes an irssi configuration file that connects to the IRC server at localhost:6667 and auto-joins the `#bots` channel; to use this simply run from the git clone directory:

```
$ irssi --home=.irssi
```

Running Tests

Helga has a full test suite for its various components. Since helga is supported for multiple python versions, tests are run using tox, which can be run entirely with helga's setup.py.

```
$ python setup.py test
```

Alternatively, if you would like to run tox directly:

```
$ pip install tox
$ tox
```

Helga uses pytest as it's test runner, so you can run individual tests if you like, but you will need to install test requirements:

```
$ pip install pytest mock pretend freezegun
$ py.test
```

Building Docs

Much like the test suite, helga's documentation is built using tox:

```
$ tox -e docs
```

Or alternatively (with installing requirements):

```
$ pip install sphinx alabaster
$ cd docs
$ make html
```

Configuring Helga

As mentioned in *Getting Started*, when you install helga, a `helga` console script is created that will run the bot process. This is the simplest way to run helga, however, it will assume various default settings like assuming that both an IRC and MongoDB server to which you wish to connect run on your local machine. This may not be ideal for running helga in a production environment. For this reason you may wish to create your own configuration for helga.

Custom Settings

Helga settings files are essentially executable python files. If you have ever worked with django settings files, helga settings will feel very similar. Helga does assume some configuration defaults, but you can (and should) use a custom settings file. The behavior of any custom settings file you use is to overwrite any default configuration helga uses. For this reason, you do not need to apply all of the configuration settings (listed below) known. For example, a simple settings file to connect to an IRC server at `example.com` on port `6667` would be:

```
SERVER = {
    'HOST': 'example.com',
    'PORT': 6667,
}
```

There are two ways in which you can use a custom settings file. First, you could export a `HELGA_SETTINGS` environment variable. Alternatively, you can indicate this via a `--settings` argument to the `helga` console script. For example:

```
$ export HELGA_SETTINGS=foo.settings
$ helga
```

Or:

```
$ helga --settings=/etc/helga/settings.py
```

In either case, this value should be an absolute filesystem path to a python file like `/path/to/foo.py`, or a python module string available on `$PYTHONPATH` like `path.to.foo`.

Default Configuration

Running the `helga` console script with no arguments will run helga using a default configuration, which assumes that you are wishing to connect to an IRC server. For a full list of the included default settings, see *helga.settings*.

XMPP Configuration

Helga was originally written as an IRC bot, but now includes XMPP support as well. Since its background as an IRC bot, much of the language in the documentation and API are geared towards that. For instance, multi user chat rooms are referred to as 'channels' and users are referred to by a 'nick'. The default helga configuration will assume that you want to connect to an IRC server. To enable XMPP connections, you must specify a `TYPE` value of `xmpp` in your `SERVER` settings:

```
SERVER = {
    'HOST': 'example.com',
    'PORT': 5222,
    'TYPE': 'xmpp',
    'USERNAME': 'helga',
    'PASSWORD': 'hunter2',
}
```

Note above that you also **must** specify a value for `USERNAME` and `PASSWORD`, which will result in a Jabber ID (JID) of something like `helga@example.com`. The also assumes that the multi user chat (MUC) domain for your xmpp server is `conference.example.com`. This might not always be desirable. For this reason, you can also specify specific JID and MUC values using the keys `JID` and `MUC_HOST` respectively. In this instance, the specific JID is used to authenticate and username is not required:

```
SERVER = {
    'HOST': 'example.com',
    'PORT': 5222,
    'TYPE': 'xmpp',
    'PASSWORD': 'hunter2',
    'JID': 'someone@example.com',
    'MUC_HOST': 'chat.example.com',
}
```

Also, just like IRC support, helga can automatically join chat rooms configured in the setting *`CHANNELS`*. You can configure this a couple of different ways, the easiest being a shorthand version of the room name, prefixed with a '#'. For example, given a room with a JID of `bots@conf.example.com`, the setting might look like:

```
CHANNELS = [
    '#bots',
]
```

Alternatively, you *can* specify the full JID:

```
CHANNELS = [
    'bots@conf.example.com',
]
```

Just like IRC, you can specify a room password using a two-tuple:

```
CHANNELS = [
    ('#bots', 'room_password'),
]
```

HipChat Support

HipChat allows for clients to connect to its service using XMPP. If you are intending to use helga as a HipChat bot, you will first need to take note of the settings needed to connect (see HipChat XMPP Settings). This also applies to anyone using the self-hosted HipChat server. A server configuration for connecting to HipChat might look like:

```
SERVER = {
    'HOST': 'chat.hipchat.com',
    'PORT': 5222,
    'JID': '00000_00000@chat.hipchat.com',
    'PASSWORD': 'hunter2',
    'MUC_HOST': 'conf.hipchat.com',
    'TYPE': 'xmpp',
}
```

HipChat makes a few assumtions that are different from standard XMPP clients. First, you **must** specify the *NICK* setting as the user's first name and last name:

```
NICK = 'Helga Bot'
```

Also, if you want @ mentions to work with command plugins so that this will work:

```
@HelgaBot do something
```

Set *COMMAND_PREFIX_BOTNICK* as the string '@?' + the @ mention name of the user. For example, if the @ mention name is 'HelgaBot':

```
COMMAND_PREFIX_BOTNICK = '@?HelgaBot'
```

Finally, HipChat does not require that room members have unique JID values. Considering a user in a room might have a JID of `room@host/user_nick`, the default XMPP backend assumes that `user_nick` is unique. HipChat does something a little different and assumes that the resource portion of the JID is the user's full name like `room@host/Jane Smith`, which may not be unique. This means that replies from the bot that include a nick will say 'Jane Smith' rather than an @ mention like '@JaneSmith'. To enable @ mentions for bot replies, you should install the hipchat_nicks plugin and add `HIPCHAT_API_TOKEN` to your settings file:

```
$ pip install helga-hipchat-nicks
$ echo 'HIPCHAT_API_TOKEN = "your_token"' >> path/to/your/settings.py
```

Plugins

One of the most prominent features of helga is its support for plugins and plugin architecture. At their core, plugins are essentially standalone, installable python packages. There are few small rules to follow, but creating custom plugins is an incredibly easy process.

Plugin Types

Plugins have a notion of type. This essentially means that they have predefined expectations for how they behave. At this time, there are three types of plugins:

Commands

Plugins of this type require a user to specifically ask to perform some action. For example, a command plugin behave like this:

```
<sduncan> helga google something
<helga> no results found for "something"
```

(see *Command Plugins*)

Matches

Plugins of this type are intended to be a form of autoresponder that aim to provide some extra meaning or context to what a user has said in a chat. For example, a match plugin could provide extra details if someone says 'foo':

```
<sduncan> I'm talking about foo in this message
<helga> sduncan just said 'foo'
```

(see *Match Plugins*)

Preprocessors

Plugins of this type generally don't respond. However, they can modify the original message that will be received by command or match plugins.

(see *Preprocessor Plugins*)

Plugin Priorities

Plugins also have a notion of priority that affect the order in which the plugin manager will process them. Priorities can be any numerical value, but as a rule of thumb, the higher the number, the more *important* a plugin will be. More important plugins will be processed first. Note, however, that preprocessor type plugins will *always* run before command and match plugins. Therefore, preprocessors will only be weighted against other preprocessors. Commands and matches are weighted against other commands and matches.

The *helga.plugins* module has three values that may be useful for indicating the priority of a plugin:

- *PRIORITY_LOW*
- *PRIORITY_NORMAL*
- *PRIORITY_HIGH*

The actual values of these priorities can be fine tuned via custom settings (see *Configuring Helga*). Unless specifically indicated, each plugin type assumes a default priority:

- Preprocessors have a default priority of *PRIORITY_NORMAL*
- Commands have a default priority of *PRIORITY_NORMAL*
- Matches have a default priority of *PRIORITY_LOW*

Creating Plugins with Decorators

Helga comes with an easy-to-use decorator API for writing simple plugins. For the most part, this is the preferred way of creating custom plugins. In a nutshell, there are decorators in *helga.plugins* that correspond to each plugin type:

- *@command*
- *@match*
- *@preprocessor*

Command Plugins

Command plugins are those which require you to ask in order to perform some action. For these types of plugins, you will use the *@command* decorator:

helga.plugins.**command**(*command*, *aliases=None*, *help=''*, *priority=50*, *shlex=False*)

A decorator for creating command plugins

Parameters

- **command** – The command string, i.e. 'search' for a command 'helga search foo'
- **aliases** – A list of command aliases. If a command 'search' has an alias list ['s'], then 'helga search foo' and 'helga s foo' will both run the command.
- **help** – An optional help string for the command. This is used by the builtin help plugin.
- **priority** – The priority of the plugin. Default is *PRIORITY_NORMAL*.
- **shlex** – A boolean indicating whether to use shlex arg string parsing rather than naive whitespace splitting.

Decorated functions should follow this pattern:

func(*client*, *channel*, *nick*, *message*, *cmd*, *args*)

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server
- **cmd** – The parsed command string which could be the registered command or one of the command aliases
- **args** – The parsed command arguments as a list, i.e. any content following the command. For example: `helga foo bar baz` would be `['bar', 'baz']`

Returns String or list of strings to return via chat. None or empty string or list for no response

For example:

```
from helga.plugins import command

@command('foo', aliases=['f'], help='The foo command')
def foo(client, channel, nick, message, cmd, args):
    return u'You said "helga foo"'
```

For argument parsing, there are currently two supported behaviors. The default is to perform whitespace splitting on the argument string. For example, given a command:

```
helga foo bar "baz qux"
```

the resulting args list to the command function would be:

```
['bar', '"baz', 'qux"']
```

For some plugins, this may be less than ideal. Therefore, you can optionally pass `shlex=True` to the *`@command`* decorator. This changes the behavior in such a way that in the previous example, the resulting args list would be:

```
['bar', 'baz qux']
```

This behavior can also be configured globally by configuring `COMMAND_ARGS_SHLEX = True` in your settings file (see *Default Configuration*)

Important: Shlex argument parsing will become the default behavior in a future version of helga.

Match Plugins

Match plugins are those that are intended to be a form of autoresponder. They are meant to provide some extra meaning or context to what a user has said in chat. For these types of plugins, you will use the *`@match`* decorator:

`helga.plugins.`**`match`**(*pattern*, *priority=25*)

A decorator for creating match plugins

Parameters

- **`pattern`** – A regular expression string used to match against a chat message. Optionally, this argument can be a callable that accepts a chat message string as its only argument and returns a value that can be evaluated for truthiness.
- **`priority`** – The priority of the plugin. Default is *`PRIORITY_LOW`*

Decorated match functions should follow this pattern:

`func`(*client*, *channel*, *nick*, *message*, *matches*)

Parameters

- **`client`** – an instance of *`helga.comm.irc.Client`* or *`helga.comm.xmpp.Client`*
- **`channel`** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **`nick`** – The nick of the user sending the message
- **`message`** – The full message string received from the server
- **`matches`** – The result of `re.findall` if decorated with a regular expression, otherwise the return value of the callable passed

Returns String or list of strings to return via chat. None or empty string or list for no response

For example:

```
from helga.plugins import match

@match(r'foo')
def foo(client, channel, nick, message, matches):
    return u'{0} said foo'.format(nick)
```

In most cases, this decorator will have a single regular expression as its argument. However, it can also accept a callable. This callable should accept a single argument: the message contents received from the chat server. There is no explicit return value type, but the return value should be able to be evaluated for truthiness. When that return value has truth, then the decorated function will be called. For example:

```
import time
from helga.plugins import match

def match_even(message):
    if int(time.time()) % 2 == 0:
        return 'Even Time!'

@match(match_even)
def even(client, channel, nick, message, matches):
    # Will send 'Match: Even Time!' to the server
    return u'Match: {0}'.format(matches)
```

Preprocessor Plugins

Preprocessor plugins generally don't respond. Instead, they are intended to potentially modify the original chat message that will be received by command or match plugins. For these types of plugins, you will use the *@preprocessor* decorator:

helga.plugins.**preprocessor**(*priority=50*)
A decorator for creating preprocessor plugins

Parameters **priority** – The priority of the plugin. Default is *PRIORITY_NORMAL*

Decorated preprocessor functions should follow this pattern:

func(*client*, *channel*, *nick*, *message*, *matches*)

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server

Returns a three-tuple (channel, nick, message) containing any modifications

For example:

```
from helga.plugins import preprocessor

@preprocessor
def foo(client, channel, nick, message):
    # Other plugins will think the message argument is what is returned
    return channel, nick, 'NOT THE ORIGINAL MESSAGE'
```

Decorator Chaining

The decorators for commands, matches, and preprocessors can be chained for more complex behavior. For example, should you wish to have a command that could add or remove patterns for a match, you could chain both *@command* and *@match*. Note, however, that each plugin type decorator expects that decorated functions accept a specific number of arguments. For this reason, it is best to use `*args` and argument length checking (this may change in the future).

For example, let's say we want a plugin that will match a dynamic set of patterns, but also gives the ability to add or remove patterns and modifies the incoming message by prepending text to indicate that it has been processed:

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

PATTERNS = set()

def check(message):
    global PATTERNS
    return re.findall('({0})'.format('|'.join(PATTERNS)))

@preprocessor
@match(check)
@command('matcher', help='Usage: helga (add|remove) <pattern>')
def matcher(client, channel, nick, message, *args):
    global PATTERNS

    if len(args) == 0:
        # Preprocessor
        return channel, nick, u'[matcher] {0}'.format(message)
    elif len(args) == 1:
        # Match - args[0] is return value of check(), re.findall
        found_list = args[0]
        return u'What you said matched: {0}'.format(found_list)
    elif len(args) == 2:
        # Command: args[1] is the parsed argument string
        command, pattern = args[1][:2]
        if command == 'add':
            PATTERNS.add(pattern)
            return u'Added {0}'.format(pattern)
        else:
            PATERNS.discard(pattern)
            return u'Removed {0}'.format(pattern)
```

Note, decorator chaining is only one way to create complex behavior for plugins. There is also a class-based plugin API (see *Class-Based Plugins*)

Handling Unicode

Plugins should try to deal exclusively with unicode as much as possible. This is important to keep in mind since all plugins that accept string arguments will receive unicode strings specifically and not byte strings. For the most part, helga's client connection assumes a UTF-8 encoding for all incoming messages. Note, though, that plugins that don't explicitly return unicode responses will not fail; the internal plugin manager will implicitly handle convertng all responses to the correct format (unicode or byte strings) needed by the chat server. There are also useful utilities for dealing with unicode support in plugins found in *helga.util.encodings*:

- *from_unicode*
- *from_unicode_args*
- *to_unicode*
- *to_unicode_args*

Accessing The Database

As mentioned in *Requirements*, MongoDB is highly recommended, but not required dependency. Having a MongoDB server that helga can use means that plugins can store data for use across restarts. This may be incredibly useful depending on the needs of your plugin. If your MongoDB connection is configured properly according to *Core Settings*, two pymongo objects in *helga.db* will be available for use:

- *helga.db.client*: A pymongo MongoClient object, the connection client to MongoDB
- *helga.db.db*: A pymongo Database object, the default MongoDB database to use

Using this database connection in a plugin is very simple:

```
from helga.db import db

db.my_collection.insert({'foo': 'bar'})
db.my_collection.find()
```

For more information on using this, see the pymongo API documentation.

Note: Should helga not be configured properly for MongoDB, or should a connection to MongoDB fail, the database object *helga.db.db* will explicitly be `None`. Therefore, it may be important for plugins that depend on MongoDB to check for this condition.

Requiring Settings

In many instances, plugins may require some configurable setting in a custom helga settings file (see *Custom Settings*). As a general rule of thumb, configurable settings should be documented by a plugin but in no way should expect that they be present in `helga.settings`. Plugins should use `getattr` for retrieving custom settings and assume some default value:

```
from helga import settings

my_setting = getattr(settings, 'MY_SETTING_VALUE', 42)
```

Also, although not explicitly required, settings names should be prefixed with the name of the plugin. This should help in organizing custom settings. For example, if a plugin `foo` uses a custom setting `SOME_VALUE`, then try to expect a setting `FOO_SOME_VALUE`.

Communicating Asynchronously

In some cases, plugins may need to perform some blocking action such as communicating with an external API. If a plugin were to perform this action and directly return a string response, this may block other plugins from processing. To get around this concern, plugins can, instead of returning a response, raise *ResponseNotReady*. This will indicate to helga's plugin manager that a response may be sent at some point in the future. In this instance, helga will continue to process other plugins, unless configured to only return first response, in which case no other plugins will be processed (see *Default Configuration*). For example:

```
from helga.plugins import command, ResponseNotReady

@command('foo')
def foo(client, channel, nick, message, cmd, args):
    # Run some async action
    raise ResponseNotReady
```

In order to actually invoke some asynchronous action, most plugins can and should utilize the fact that helga is built using Twisted by calling `twisted.internet.reactor.callLater`. For example:

```
from twisted.internet import reactor

def do_something(arg, kwarg=None):
    print arg or kwarg

# Have the event loop run `do_something` in 30 seconds
reactor.callLater(30, do_something, None, kwarg='foo')
```

For more details on this see the Twisted Documentation. To revisit the previous plugin example:

```
from twisted.internet import reactor
from helga.plugins import command, ResponseNotReady

def foo_async(client, channel, args):
    client.msg(channel, 'someone ran the foo command with args: {0}'.format(args))

@command('foo')
def foo(client, channel, nick, message, cmd, args):
    reactor.callLater(5, foo_async, client, channel, args)
    raise ResponseNotReady
```

Notice above that the callback function `foo_async` takes the client connection as an argument. Should a plugin need to respond asynchronously to the server, it is generally a good idea for deferred callbacks to accept at a minimum the client and the channel of the message. In addition, there are several useful methods of both *helga.comm.irc.Client* and *helga.comm.xmpp.Client* that can be used for asynchronous communication:

- *helga.comm.irc.Client.msg()*
- *helga.comm.irc.Client.me()*
- *helga.comm.xmpp.Client.msg()*
- *helga.comm.xmpp.Client.me()*

Signals/Notifications of Helga Events

Helga makes heavy use of signals for events provided by smokesignal. In this way, plugins can receive notifications when some event occurs and perform some action such as loading data from the database or setting some preferred state. At this time, there are several included signals that fire on given events and provide callbacks with certain arguments:

started Fired when the helga process starts. Callbacks should accept no arguments.

signon Fired when helga successfully connects to the chat server. Callbacks should follow:

func(*client*)

Parameters **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client* depending on the server type

join Fired when helga joins a channel. Callbacks should follow:

func(*client*, *channel*)

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client* depending on the server type

- **channel** – the name of the channel

left Fired when helga leaves a channel. Callbacks should follow:

func(*client*, *channel*)

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client* depending on the server type
- **channel** – the name of the channel

user_joined Fired when a user joins a channel helga is in. Callbacks should follow:

func(*client*, *nick*, *channel*)

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client* depending on the server type
- **nick** – the nick of the user that joined
- **channel** – the name of the channel

user_left Fired when a user leaves a channel helga is in. Callbacks should follow:

func(*client*, *nick*, *channel*)

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client* depending on the server type
- **nick** – the nick of the user that left
- **channel** – the name of the channel

Packaging and Distribution

If you have created a simple helga plugin, you may be asking "What now?". Helga, rather than using plugin directories containing lots of one-off scripts, makes use of proper python packaging to manage plugin installation. This may be a bit of an advanced topic if you are new to python packaging, but for the most part, you can follow a small number of repeatable steps for simple plugins.

Basic Project Structure

For the most part, simple plugins will follow the same basic project structure:

```
helga_my_plugin
- helga_my_plugin.py
- LICENSE
- MANIFEST.in
- README.rst
- setup.py
- tests.py
- tox.ini
```

helga_my_plugin.py This is the actual plugin script. This can be named whatever you feel like naming it, but it is good practice to name this something like `helga_<name of plugin>.py`.

LICENSE Since helga is dual-licensed MIT and GPL, this can be either MIT or GPL

MANIFEST.in If you wish to include any non-python files with your plugin, you should include this file. For example, if you wish to include the README and LICENSE, the contents of this file would be:

```
LICENSE
README.rst
```

README.rst Not required to be a reStructuredText document, but it is good practice to describe what the plugin does, how to use it, and if there are any custom settings that should be set.

setup.py setuptools setup script (see *setuptools and entry_points*)

tests.py If you write any unit tests for your plugin

tox.ini If you write any unit tests for your plugin and use tox to run them. It is generally a good idea to use tox to run tests against python 2.6 and 2.7 since helga supports both of those versions.

setuptools and entry_points

Not only does a plugin's setup.py file declare project information and allow it to be installed with pip, it is also how helga loads plugins at runtime. To do this, helga uses a setuptools feature known as entry_points. To understand how to use this, take the above project structure as an example. Let's say that the contents of `helga_my_plugin.py` looks like this:

```
from helga.plugins import match

@match(r'foo')
def foo(client, channel, nick, message, matches):
    return u'{0} said foo'.format(nick)
```

A basic `setup.py` file for this project might look like:

```
from setuptools import setup, find_packages

setup(
    name='helga_my_plugin',
    version='0.0.1',
    description='A foo plugin',
    author="Jane Smith"
    author_email="jane.smith@example.com",
    packages=find_packages(),
    py_modules=['helga_my_plugin'],
    include_package_data=True,
    zip_safe=True,
    entry_points=dict(
        helga_plugins=[
            'my_plugin = helga_my_plugin:foo',
        ],
    ),
)
```

Before talking about `entry_points`, take note of some other important lines.

py_modules If your plugin is a single python file, you will need include it without the '.py' extension in a string list.

include_package_data If you intend on including files specified in a `MANIFEST.in` file, you will need to set this to True.

Now, let's talk about the `entry_points` line. The helga plugin loader will look for any installed python package that declares `helga_plugins` entry points. These should be list of strings of the form:

```
plugin_name = module.path:decorated_function
```

The 'plugin_name' portion should be a simple name for the plugin, such as 'my_plugin' in the 'helga_my_plugin' example above. The latter half must be colon delimited containing a module path and the function decorated using *@command*, *@match*, or *@preprocessor*. So if a file `helga_my_plugin.py` contains:

```
from helga.plugins import match

@match(r'foo')
def foo(*args):
    return 'foo'
```

the entry point would be `helga_my_plugin:foo`. For more information and details on how entry points work, see the entry_points documentation.

Distributing Plugins

The preferred distribution channel for helga plugins is PyPI so that plugins can be installed using pip. Once you have properly packaged your plugin, submit it to PyPI:

```
$ python setup.py sdist register upload
```

Using A Project Template

If you use cookiecutter for managing project templates, there is a third-party helga plugin cookiecutter template here: https://github.com/bigjust/cookiecutter-helga-plugin

Installing Plugins

If plugins are properly packaged and distributed according to *Packaging and Distribution*, then any new plugins for helga to use can be installed using pip. If helga has been installed into a virtualenv as mentioned in *Getting Started*, activate that virtualenv prior to installing the new plugin:

```
$ source bin/activate
$ pip install helga-my-plugin
```

Note, however, that you will need to full restart any running helga process in order to use the new plugins. This behavior may change in future versions of helga. If a plugin is not distributed using PyPI, but is available via some source repository, you can still install it with a little more work:

```
$ source bin/activate
$ git clone git@example.com:janedoe/helga-my-plugin.git src/helga-my-plugin
$ cd src/helga-my-plugin
$ python setup.py develop
```

Note, that installing a plugin will mean that it will be loaded when helga starts unless it is not included in the plugins whitelist *helga.settings.ENABLED_PLUGINS* or it is listed in the plugins blacklist *helga.settings.DISABLED_PLUGINS* The default behavior is that all plugins installed on the system are loaded and made available for use in IRC.

With this in mind, installed plugins are available for use, but they may not immediately be so. Helga maintains a list of plugin names that indicate which plugins should be enabled by default in a channel, which is configured via *helga.settings.DEFAULT_CHANNEL_PLUGINS*. If a plugin name does not appear in this list, a user in a channel will not be able to use it until it is enabled with the *manager* plugin:

```
<sduncan> !plugins enable my_plugin
```

Class-Based Plugins

All of the above documentation for creating plugins makes use of helga's simple decorator API. Generally speaking, the decorator API is the preferred way of authoring plugins. However, simple decorated functions may not be robust enough for all plugins. For this reason, there is a class-based API that can be used instead. In fact, this is what is used behind the scenes for the decorator API.

Base Plugin Class

At a high level, plugin objects should be some form of a sublass of *helga.plugins.Plugin*:

class helga.plugins.**Plugin**(*priority=50*)

The base class for helga plugins. There are three main methods of this base class that are important for creating class-based plugins.

`preprocess`

Run by the plugin registry as a preprocessing mechanism. Allows plugins to modify the channel, nick, and/or message that other plugins will receive.

`process`

Run by the plugin registry to allow a plugin to process a chat message. This is the primary entry point for plugins according to the plugin manager, so it should either return a response or not.

`run`

Run internally by the plugin, generally from within the `process` method. This should do the actual work to generate a response. In other words, `process` should handle checking if the plugin should handle a message and then return whatever `run` returns.

preprocess(*client*, *channel*, *nick*, *message*)

A preprocessing filter for plugins. This allows a plugin to modify a received message prior to that message being handled by this plugin's or other plugin's `process` method.

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server

Returns a three-tuple (channel, nick, message) containing any modifications

process(*client*, *channel*, *nick*, *message*)

This method of a plugin is called by helga's plugin registry to process an incoming chat message. This

should determine whether or not the plugin `run` method should be called. If so, it should return whatever return value `run` generates. If not, `None` should be returned.

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server

Returns None if the plugin should not run, otherwise the return value of the `run` method

run(*client*, *channel*, *nick*, *message*, **args*, ***kwargs*)

Executes this plugin with a given message to generate a response. This should run without regard to whether it should or should not for a given message. Note, that this is where the actual work for the plugin should occur. Subclasses should implement this method.

A return value of None, an empty string, or empty list implies that no response should be sent via chat. A non-empty string, list of strings, or raised *ResponseNotReady* implies a response to be sent.

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server

Returns None if no response is to be sent back to the server, a non-empty string or list of strings if a response is to be returned

Plugin implementations can subclass this base class directly, but there are convenience subclasses for each plugin type that already do a lot of the heavy lifting.

Command Subclasses

To create a class-based command plugin, subclass *helga.plugins.Command*. For example:

```
from helga.plugins import Command

class FooCommand(Command):
    command = 'foo'
    aliases = ['f']
    help = 'Return the foo count. Usage: helga (foo|f)'

    def __init__(self, *args, **kwargs):
        super(FooCommand, self).__init__(*args, **kwargs)
        self.foo_count = 0

    def run(self, client, channel, nick, message, cmd, args):
```

```
        self.foo_count += 1
        return u'Foo count is {0}'.format(self.foo_count)
```

Match Subclasses

To create a class-based match plugin, subclass *helga.plugins.Match*. For example:

```
from helga.plugins import Match

class FooMatch(Match):
    pattern = r'foo (\w+)'

    def run(self, client, channel, nick, message, matches):
        return u"{0} said 'foo' followed by '{1}'".format(nick, matches[0])
```

Or in the case of using a callable as a pattern:

```
import time

from helga.plugins import Match

class FooMatch(Match):

    def __init__(self, *args, **kwargs):
        super(FooMatch, self).__init__(*args, **kwargs)
        self.pattern = self.match_foo

    def match_foo(self, message):
        if 'foo' in message:
            return time.time()

    def run(self, client, channel, nick, message, matches):
        return u"{0} said 'foo' at {0}".format(nick, matches)
```

Preprocessor Subclasses

There is no direct *Plugin* subclass for preprocessor plugins. Preprocessors using the decorator API are merely instances of the base *Plugin* class (see *Base Plugin Class*). However, to create a preprocessor plugin using a class-based approach:

```
from helga.plugins import Plugin

def FooPreprocessor(Plugin):

    def preprocess(self, client, channel, nick, message):
        # Ignore anything from nicks that start with a vowel
        if nick[0] in 'aeiou':
            return channel, nick, u''
        return channel, nick, message
```

Packaging Class-Based Plugins

Class-based plugins are packaged in exactly the same manner as those using the decorator API (see *Packaging and Distribution*). The only difference is with respect to entry points. Whereas with decorator plugins, the entry point

follows a 'module:function' pattern, class-based plugins follow a 'module:class' pattern. For example, given this plugin in a file `helga_foo.py`:

```
from helga.plugins import Command

class FooCommand(Command):
    pass
```

The respective entry point string might look something like this:

```
foo = helga_foo:FooCommand
```

Supporting XMPP

You shouldn't need to make any special changes to plugins if you follow the documenation above. However, remember that helga was started as an IRC bot, so things work a bit more to that favor. Plugins will still receive `client`, `channel`, `nick`, and `message` arguments.

Note, though, that values for `channel` will **never** be the full JID of a chat room. Instead, they will be the `user` portion of the room JID, prepended with a '#'. For example:

```
bots@conf.example.com
```

would become a channel named `#bots` and private messages from:

```
user@host.com
```

would become a channel named `user`.

Nick values operate in a similar manner, only using the `resource` portion of the JID for group chat. For example:

```
bots@conf.example.com/foo
```

would become a nick named:

```
foo
```

and a private message from:

```
foo@host.com
```

would become a nick named:

```
foo
```

For more information about how this works see *helga.comm.xmpp.Client.parse_channel()* and *helga.comm.xmpp.Client.parse_nick()*.

Webhooks

As of helga version 1.3, helga includes support for pluggable webhooks that can interact with the running bot or communicate via IRC. The webhook architecture is extensible much in the way that plugins work, allowing you to create new or custom HTTP services.

Overview

The webhooks system has two important aspects and core concepts: the HTTP server and routes.

HTTP Server

The webhooks system consists of an HTTP server that is managed by a command plugin named *webooks*. This plugin is enabled by default and handles starting the HTTP server is started when helga successfully signs on to IRC. The server process is configured to listen on a port specified by the setting *WEBHOOKS_PORT*.

The actual implementation of this HTTP server is a combination of a TCP listner using the Twisted reactor, and `twisted.web.server.Site` with a single root resource (see *WebhookRoot*) that manages each registered URL route.

Note: This server is managed via a plugin only so it can be controlled via IRC.

Routes

Routes are the plugins of the webhook system. They are essentially registered URL paths that have some programmed behavior. For example, `http://localhost:8080/github`, or `/github` specifically, might be the registered route for a webhook that announces github code pushes on an IRC channel. Routes are declared using a decorator (see *The @route Decorator*), which will feel familiar to anyone with flask experience. At this time, routes also support HTTP basic authentication, which is configurable with a setting *WEBHOOKS_CREDENTIALS*.

The `@route` Decorator

Much like the plugin system, webhook routes are created using an easy to use decorator API. At the core of this API is a single decorator *@route*, which will feel familiar to anyone with flask experience:

`helga.plugins.webhooks.`**`route`**(*path*, *methods=None*)

Decorator to register a webhook route. This requires a path regular expression, and optionally a list of HTTP methods to accept, which defaults to accepting `GET` requests only. Incoming HTTP requests that use a non-allowed method will receive a 405 HTTP response.

Parameters

- **`path`** – a regular expression string for the URL path of the route
- **`methods`** – a list of accepted HTTP methods for this route, defaulting to `['GET']`

Decorated routes must follow this pattern:

`func`(*request*, *client*)

Parameters

- **`request`** – The incoming HTTP request, `twisted.web.http.Request`
- **`client`** – The client connection. An instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*

Returns a string HTTP response

For example:

```
from helga.plugins.webhooks import route

@route(r'/foo')
def foo(request, client):
    client.msg('#foo', 'someone hit the /foo endpoint')
    return 'message sent'
```

Routes can be configured to also support URL parameters, which act similarly to django's URL routing mechanisms. By introducing named pattern groups in the regular expression string. These will be passed as keyword arguments to the decorated route handler:

```
from helga.plugins.webhooks import route

@route(r'/foo/(?P<bar>[0-9]+)')
def foo(request, client, bar):
    client.msg('#foo', 'someone hit the /foo endpoint with bar {0}'.format(bar))
    return 'message sent'
```

Authenticated Routes

The webhooks system includes mechanisms for restricting routes to authenticated users. Note, that this is only supported to handle HTTP basic authentication. Auth credentials are currently limited to hard-coded username and password pairs configured as a list of two-tuples, the setting *WEBHOOKS_CREDENTIALS*. Routes are declared as requiring authentication using the *@authenticated* decorator:

For example:

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

@route(r'/foo')
@authenticated
def foo(request, client):
    client.msg('#foo', 'someone hit the /foo endpoint')
    return 'message sent'
```

Important: The *@authenticated* decorator **must** be the first decorator used for a route handler, otherwise the authentication check will not happen prior to a route being handled. This requirement may change in the future.

Sending Non-200 Responses

By default, route handlers will send a 200 response to any incoming request. However, in some cases it may be necessary to explicitly return a non-200 response. In order to accomplish this, a route handler can manually set the response status code on the request object:

```
from helga.plugins.webhooks import route

@route(r'/foo')
def foo(request, client):
    request.setResponseCode(404)
    return 'foo is always 404'
```

In addition to this, route handlers can also raise *helga.plugins.webhooks.HttpError*:

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

@route(r'/foo')
def foo(request, client):
    raise HttpError(404, 'foo is always 404')
```

Using Templates

When installed, helga will have pystache installed as well, which can be used for templating webhooks that produce HTML responses. It is important though that any webhooks be packaged so that any external `.mustache` templates are packaged as well, which can be done by adding to a `MANIFEST.in` file (see *Packaging and Distribution*):

```
recursive-include . *.mustache
```

Handling Unicode

Handling unicode for webhooks is not as strict as with plugins, but the same guidelines should follow. For example, webhooks should return unicode, but know that unicode strings are explicitly encoded as UTF-8 byte strings. See the plugin documentation *Handling Unicode*.

Accessing The Database

Database access for webhooks follows the same rules as for plugins. See the plugin documentation *Accessing The Database*

Requiring Settings

Requiring settings for webhooks follows the same rules as for plugins. See the plugin documentation *Requiring Settings*

Packaging and Distribution

Much like plugins, webhooks are also installable python modules. For that reason, the rules for packaging and distributing webhooks are the same as with plugins (see plugin *Packaging and Distribution*). However, there is one minor difference with respect to declaring the webhook entry point. Rather than indicating the webhook as a `helga_plugins` entry point, it should be placed in an entry point section named `helga_webhooks`. For example:

```
setup(
    entry_points=dict(
        helga_webhooks=[
            'api = myapi:decorated_route'
        ]
    )
)
```

Installing Webhooks

Webhooks are installed in the same manner that plugins are installed (see plugin *Installing Plugins*). And much like plugins, there are settings to control both a whitelist and blacklist for loading webhook routes (see `ENABLED_PLUGINS` and `DISABLED_PLUGINS`). To explicitly whitelist webhook routes to be loaded, use `ENABLED_WEBHOOKS`. To explicitly blacklist webhook routes from being loaded, use `DISABLED_WEBHOOKS`.

_builtin:

Builtin Features

Helga comes with many builtin plugins, webhooks, and features.

Supported Backends

As of version 1.7.0, helga supports IRC, XMPP, and HipChat out of the box. Note, however, that helga originally started as an IRC bot, so much of the terminology will reflect that. The current status of XMPP and HipChat support is very limited and somewhat beta. In the future, helga may have a much more robust and pluggable backend system to allow connections to any number of chat services.

The default configuration assumes that you wish to connect to an IRC server. However, if you wish to connect to an XMPP or HipChat server, see *XMPP Configuration*.

Builtin Plugins

Helga comes with several builtin plugins. Generally speaking, it is better to have independently maintained plugins rather than modifying helga core. In fact, many of the plugins listed here may be retired as core plugins and moved to externally maintained locations. This is mainly due to the fact that some are either not useful as core plugins or would require more maintenance for helga core than should be needed.

Important: Some builtin plugins may be deprecated and removed in a future version of helga. They will be moved and maintained elsewhere as independent plugins.

---

help

A command plugin to show help strings for any installed command plugin. Usage:

```
helga (help|halp) [<plugin>]
```

With no arguments, all command plugin help strings are returned to the requesting user in a private message.

manager

Important: This plugin requires database access for some features

---

A command plugin that acts as an IRC-based plugin manager. Usage:

```
helga plugins (list|(enable|disable) (<name> ...))
```

The 'list' subcommand will list out both enabled and disabled plugins for the current channel. For example:

```
<sduncan> !plugins list
<helga> Enabled plugins: foo, bar
<helga> Disabled plugins: baz
```

Both enable and disable will respectively move a plugin between enabled and disabled status on the current channel. If a database connection is configured, both enable and disable will record plugins as either automatically enabled for the current channel or not. For example:

```
<sduncan> !plugins enable baz
<sduncan> !plugins list
<helga> Enabled plugins: foo, bar, baz
<sduncan> !plugins disable baz
<helga> Enabled plugins: foo, bar
<helga> Disabled plugins: baz
```

operator

Important: This plugin requires database access for some features

A command plugin that exposes some capabilities exclusively for helga operators. Operators are nicks with elevated privileges configured via the `OPERATORS` setting (see *Core Settings*). Usage:

```
helga (operator|oper|op) (reload <plugin>|(join|leave|autojoin (add|remove)) <channel>
↪).
```

Each subcommand acts as follows:

`reload <plugin>` Experimental. Given a plugin name, perform a call to the python builtin `reload()` of the loaded module. Useful for seeing plugin code changes without restarting the process.

`(join|leave) <channel>` Join or leave a specified channel

`autojoin (add|remove) <channel>` Add or remove a channel from a set of autojoin channels. This features requries database access.

ping

A simple command plugin to ping the bot, which will always respond with 'pong'. Usage:

```
helga ping
```

webhooks

A special type of command plugin that enables webhook support (see *Webhooks*). This command is more of a high-level manager of the webhook system. Usage:

```
helga webhooks (start|stop|routes)
```

Both `start` and `stop` are privileged actions and can start and stop the HTTP listener for webhooks respectively. To use them, a user must be configured as an operator. The `routes` subcommand will list all of the URL routes known to the webhook listener.

Builtin Webhooks

Helga also includes some builtin webhooks for use out of the box.

announcements

The announcements webhook exposes a single HTTP endpoint for allowing the ability to post a message in an IRC channel via an HTTP request. This webhook **only** supports POST requests and requires HTTP basic authentication (see *Authenticated Routes*). Requests must be made to a URL path `/announce/<channel>` such as `/announce/bots` and made with a POST parameter `message` containing the IRC message contents. The endpoint will respond with 'Message Sent' on a successful message send.

logger

The logger webhook is a browsable web frontend for helga's builtin channel logger (see *Channel Logging*). This webhook is enabled by default but requires that channel logging is enabled for it to be of any use. Logs are shown in a dated order, grouped by channel.

Without any configuration, this web frontend will allow browsing all channels in which the bot resides or has resided. This behavior can be changed with the setting *CHANNEL_LOGGING_HIDE_CHANNELS* which should be a list of channel names that should be hidden from the browsable web UI. NOTE: they can still be accessed directly.

This webhook exposes a root `/logger` URL endpoint that serves as a channel listing. The webhook will support any url of the form `/logger/<channel>/YYYY-MM-DD` such as `/logger/foo/2014-12-31`.

Channel Logging

As of the 1.6 release, helga includes support for a simple channel logger, which may be useful for those wanting to helga to, in addition to any installed plugins, monitor and save conversations that occur on any channel in which the bot resides. This is a helga core feature and not managed by a plugin, mostly to ensure that channel logging *always* happens with some level of confidence that no preprocess plugins have modified the message. Channel logging feature can be either enabled or disabled via the setting *CHANNEL_LOGGING*.

Channel logs are kept in UTC time and stored in dated logfiles that are rotated automatically. These log files are written to disk in a configurable location indicated by *CHANNEL_LOGGING_DIR* and are organized by channel name. For example, message that occurred on Dec 31 2014 on channel #foo would be written to a file `/path/to/logs/#foo/2014-12-31.txt`

The channel logger also includes a web frontend for browsing any logs on disk, documented as the builtin webhook *logger*.

Note: Non-public channels (i.e. those not beginning with a '#') will be ignored by helga's channel logger. No conversations via private messages will be logged.

API Documentation

helga.comm.irc

Twisted protocol and communication implementations for IRC

class helga.comm.irc.**Client**(*factory=None*)

An implementation of *twisted.words.protocols.irc.IRCClient* with some overrides derived from helga settings (see *Configuring Helga*). Some methods are overridden to provide additional functionality.

channels = set()

A set containing all of the channels the bot is currently in

operators = set()

A set containing all of the configured operators (setting *OPERATORS*)

last_message = dict()

A channel keyed dictionary containing dictionaries of nick -> message of the last messages the bot has seen a user send on a given channel. For instance, if in the channel #foo:

```
<sduncan> test
```

The contents of this dictionary would be:

```
self.last_message['#foo']['sduncan'] = 'test'
```

channel_loggers = dict()

A dictionary of known channel loggers, keyed off the channel name

action(*user*, *channel*, *message*)

Handler for an IRC message. This method handles logging channel messages (if it occurs on a public channel) as well as allowing the plugin manager to send the message to all registered plugins. Should the plugin manager yield a response, it will be sent back over IRC.

Parameters

- **user** – IRC user string of the form {nick}!~{user}@{host}
- **channel** – the channel from which the message came
- **message** – the message contents

alterCollidedNick(*nickname*)

Called when the bot has a nickname collision. This will generate a new nick containing the perferred nick and the current timestamp.

Parameters **nickname** – the nickname that was already taken

encoding = 'UTF-8'

The assumed encoding of IRC messages

erroneousNickFallback = None

A backup nick should the preferred *nickname* be taken. This defaults to a string in the form of the preferred nick plus the timestamp when the process was started (i.e. helga_12345)

get_channel_logger(*channel*)

Gets a channel logger, keeping track of previously requested ones. (see *Channel Logging*)

Parameters **channel** – A channel name

Returns a python logger suitable for channel logging

irc_unknown(*prefix*, *command*, *params*)
Handler for any unknown IRC commands. Currently handles /INVITE commands

Parameters

- **prefix** – any command prefix, such as the IRC user
- **command** – the IRC command received
- **params** – list of parameters for the given command

is_public_channel(*channel*)
Checks if a given channel is public or not. A channel is public if it starts with '#' and is not the bot's nickname (which occurs when a private message is received)

Parameters **channel** – the channel name to check

join(*channel*, *key=None*)
Join a channel, optionally with a passphrase required to join.

Parameters

- **channel** – the name of the channel to join
- **key** – an optional passphrase used to join the given channel

joined(*channel*)
Called when the client successfully joins a new channel. Adds the channel to the known channel list and sends the `join` signal (see *Signals/Notifications of Helga Events*)

Parameters **channel** – the channel that has been joined

leave(*channel*, *reason=None*)
Leave a channel, optionally with a reason for leaving

Parameters

- **channel** – the name of the channel to leave
- **reason** – an optional reason for leaving

left(*channel*)
Called when the client successfully leaves a channel. Removes the channel from the known channel list and sends the `left` signal (see *Signals/Notifications of Helga Events*)

Parameters **channel** – the channel that has been left

lineRate = None
An integer, in seconds, if IRC messages should be sent at a limit of once per this many seconds. `None` implies no limit. (setting *RATE_LIMIT*)

log_channel_message(*channel*, *nick*, *message*)
Logs one or more messages by a user on a channel using a channel logger. If channel logging is not enabled, nothing happens. (see *Channel Logging*)

Parameters

- **channel** – A channel name
- **nick** – The nick of the user sending an IRC message
- **message** – The IRC message

me(*channel*, *message*)
Equivalent to: /me message

Parameters

- **channel** – The IRC channel to send the message to. A channel not prefixed by a '#' will be sent as a private message to a user with that nick.
- **message** – The message to send

msg(*channel*, *message*)
Send a message over IRC to the specified channel

Parameters

- **channel** – The IRC channel to send the message to. A channel not prefixed by a '#' will be sent as a private message to a user with that nick.
- **message** – The message to send

nickname = None
The preferred IRC nick of the bot instance (setting *NICK*)

on_invite(*inviter*, *invitee*, *channel*)
Handler for /INVITE commands. If the invitee is the bot, it will join the requested channel.

Parameters

- **inviter** – IRC user string of the form `{nick}!~{user}@{host}`
- **invitee** – the nick of the user receiving the invite
- **channel** – the channel to which invitee has been invited

parse_nick(*full_nick*)
Parses a nick from a full IRC user string. For example from `me!~myuser@localhost` would return `me`.

Parameters **full_nick** – the full IRC user string of the form `{nick}!~{user}@{host}`

Returns The nick portion of the IRC user string

password = None
A password should the IRC server require authentication (setting *SERVER*)

privmsg(*user*, *channel*, *message*)
Handler for an IRC message. This method handles logging channel messages (if it occurs on a public channel) as well as allowing the plugin manager to send the message to all registered plugins. Should the plugin manager yield a response, it will be sent back over IRC.

Parameters

- **user** – IRC user string of the form `{nick}!~{user}@{host}`
- **channel** – the channel from which the message came
- **message** – the message contents

signedOn()
Called when the client has successfully signed on to IRC. Establishes automatically joining channels. Sends the `signon` signal (see *Signals/Notifications of Helga Events*)

sourceURL = 'http://github.com/shaunduncan/helga'
The URL where the source of the bot is found

userJoined(*user*, *channel*)
Called when a user joins a channel in which the bot resides. Responsible for sending the `user_joined` signal (see *Signals/Notifications of Helga Events*)

Parameters

- **user** – IRC user string of the form `{nick}!~{user}@{host}`
- **channel** – the channel in which the event occurred

userLeft (*user*, *channel*)
Called when a user leaves a channel in which the bot resides. Responsible for sending the `user_left` signal (see *Signals/Notifications of Helga Events*)

Parameters

- **user** – IRC user string of the form `{nick}!~{user}@{host}`
- **channel** – the channel in which the event occurred

userRenamed (*oldname*, *newname*)

Parameters

- **oldname** – the nick of the user before the rename
- **newname** – the nick of the user after the rename

username = None
A username should the IRC server require authentication (setting *SERVER*)

class `helga.comm.irc.`**Factory**
The client factory for twisted. Ensures that a client is properly created and handles auto reconnect if helga is configured for it (see settings *AUTO_RECONNECT* and *AUTO_RECONNECT_DELAY*)

buildProtocol (*address*)
Build the helga protocol for twisted, or in other words, create the client object and return it.

Parameters **address** – an implementation of *twisted.internet.interfaces.IAddress*

Returns an instance of *Client*

clientConnectionFailed (*connector*, *reason*)
Handler for when the IRC connection fails. Handles auto reconnect if helga is configured for it (see settings *AUTO_RECONNECT* and *AUTO_RECONNECT_DELAY*)

clientConnectionLost (*connector*, *reason*)
Handler for when the IRC connection is lost. Handles auto reconnect if helga is configured for it (see settings *AUTO_RECONNECT* and *AUTO_RECONNECT_DELAY*)

helga.comm.xmpp

class `helga.comm.xmpp.`**Client** (*factory*)
The XMPP client that has predetermined behavior for certain events. This client assumes some default behavior for multi user chat (MUC) by setting the conference host as `conference.HOST` using `HOST` in *SERVER*. A specific MUC host can be specified using the key `MUC_HOST` in *SERVER*.

This client is also a bit opinionated when it comes to chat rooms and how room names and nicks are delivered to plugins. Since helga originally started as an IRC bot, channels are sent to plugins as the user portion of the room JID prefixed with '#'. For example, if a message is received from `bots@conference.example.com/some_user`, the channel will be `#bots`. In this instance, plugins would see the user nick as `some_user`. For private messages, a message received from `some_user@example.com` would result in an identical channel and nick `some_user`.

factory
An instance of *Factory* used to create this client instance.

jid
The Jabber ID used by the client. A copy of the factory `jid` attribute.

nickname
The current nickname of the bot. Generally this is the user portion of the `jid` attribute, and the resource portion of chat room JIDs, but the value is obtained via the setting *NICK*. For HipChat support, this should be set to the user account's **Full Name**.

stream
The raw data stream. An instance of *twisted.words.protocols.jabber.xmlstream.XmlStream*

channels = set()
A set containing all of the channels the bot is currently in

operators = set()
A set containing all of the configured operators (setting *OPERATORS*)

last_message = dict()
A channel keyed dictionary containing dictionaries of nick -> message of the last messages the bot has seen a user send on a given channel. For instance, if in the channel `#foo`:

```
<sduncan> test
```

The contents of this dictionary would be:

```
self.last_message['#foo']['sduncan'] = 'test'
```

channel_loggers = dict()
A dictionary of known channel loggers, keyed off the channel name

format_channel (*channel*)
Formats a channel as a valid JID string. This will operate with a fallback of `channel@conference_host` should any of the following conditions happen:

- Parsing the channel as a JID fails with *twisted.words.protocols.jabber.jid.InvalidFormat*
- Either the `user` or `host` portion of the parsed JID is empty

Any prefixed '#' characters are removed. For example, assuming a conference host of 'conf.example.com':

- *#bots* would return *bots@conf.example.com*
- *bots* would return *bots@conf.example.com*
- *bots@rooms.example.com* would return *bots@rooms.example.com*
- *bots@rooms.example.com/resource* would return *bots@rooms.example.com*

Parameters **channel** – The channel to format as a full JID. Can be a simple string, '#' prefixed string, or full room JID.

Returns The full user@host JID of the room

get_channel_logger (*channel*)
Gets a channel logger, keeping track of previously requested ones. (see *Channel Logging*)

Parameters **channel** – A channel name

Returns a python logger suitable for channel logging

is_public_channel (*channel*)
Checks if a given channel is public or not. A channel is public if it starts with '#'

Parameters **channel** – the channel name to check

join(*channel*, *password=None*)
Join a channel, optionally with a passphrase required to join. Channels can either be a full, valid JID or a simple channel name like '#bots', which will be expanded into something like *bots@conference.example.com* (see `format_channel()`)

Parameters

- **channel** – the name of the channel to join
- **key** – an optional passphrase used to join the given channel

joined(*channel*)
Called when the client successfully joins a new channel. Adds the channel to the known channel list and sends the `join` signal (see *Signals/Notifications of Helga Events*)

Parameters **channel** – the channel that has been joined

leave(*channel*, *reason=None*)
Leave a channel, optionally with a reason for leaving

Parameters

- **channel** – the name of the channel to leave
- **reason** – an optional reason for leaving

left(*channel*)
Called when the client successfully leaves a channel. Removes the channel from the known channel list and sends the `left` signal (see *Signals/Notifications of Helga Events*)

Parameters **channel** – the channel that has been left

log_channel_message(*channel*, *nick*, *message*)
Logs one or more messages by a user on a channel using a channel logger. If channel logging is not enabled, nothing happens. (see *Channel Logging*)

Parameters

- **channel** – A channel name
- **nick** – The nick of the user sending an IRC message
- **message** – The IRC message

me(*channel*, *message*)
Equivalent to: /me message. This is more compatibility with existing IRC plugins that use this method. Channels prefixed with '#' are assumed to be multi user chat rooms, otherwise, they are assumed to be individual users.

Parameters

- **channel** – The XMPP channel to send the message to. A channel not prefixed by a '#' will be sent as a private message to a user with that nick.
- **message** – The message to send, which will be prefixed with '/me'

msg(*channel*, *message*)
Send a message over XMPP to the specified channel. Channels prefixed with '#' are assumed to be multi user chat rooms, otherwise, they are assumed to be individual users.

Parameters

- **channel** – The XMPP channel to send the message to. A channel not prefixed by a '#' will be sent as a private message to a user with that nick.
- **message** – The message to send

on_authenticated(*stream*)
Handler for successful authentication to the XMPP server. Establishes automatically joining channels. Sends the `signon` signal (see *Signals/Notifications of Helga Events*)

Parameters **stream** – An instance of *twisted.words.protocols.jabber.xmlstream.XmlStream*

on_connect(*stream*)
Handler for a successful connection to the server. Sets the client xml stream and starts the heartbeat service.

Parameters **stream** – An instance of *twisted.words.protocols.jabber.xmlstream.XmlStream*

on_disconnect(*stream*)
Handler for an unexpected disconnect. Logs the disconnect and stops the heartbeat service.

Parameters **stream** – An instance of *twisted.words.protocols.jabber.xmlstream.XmlStream*

on_init_failed(*failure*)
Handler for when client initialization fails. This should end contact with the server by sending the xml footer.

Parameters **failure** – The element of the failure

on_invite(*element*)
Handler that responds to channel invites from other users. This will acknowledge the request by joining the room indicated in the xml payload.

Parameters **element** – A <message/> element, instance of *twisted.words.xish.domish.Element*

on_message(*element*)
Handler for an XMPP message. This method handles logging channel messages (if it occurs on a public channel) as well as allowing the plugin manager to send the message to all registered plugins. Should the plugin manager yield a response, it will be sent back.

Parameters **message** – A <message/> element, instance of *twisted.words.xish.domish.Element*

on_nick_collision(*element*)
Handler called when the server responds of nick collision with the bot. This will generate a new nick containing the preferred nick and the current timestamp and attempt to rejoin the room it failed to join.

Parameters **element** – A <presence/> element, instance of *twisted.words.xish.domish.Element*

on_ping(*el*)
Handler for server IQ pings. Automatically responds back with a PONG.

Parameters **el** – A <iq/> PING message, instance of *twisted.words.xish.domish.Element*

on_subscribe(*element*)
Handler that responds to 'buddy requests' from other users. This will acknowledge the request by approving it.

Parameters **element** – A <presence/> element, instance of *twisted.words.xish.domish.Element*

on_user_joined(*element*)
Handler called when a user enters a public room. Responsible for sending the `user_joined` signal (see *Signals/Notifications of Helga Events*)

Parameters **element** – A <presence/> element, instance of *twisted.words.xish.domish.Element*

on_user_left(*element*)
Handler called when a user leaves a public room. Responsible for sending the `user_left` signal (see *Signals/Notifications of Helga Events*)

Parameters **element** – A <presence/> element, instance of *twisted.words.xish.domish.Element*

parse_channel(*element*)
Parses a channel name from an element. This follows a few rules to determine the right channel to use. Assuming a 'from' jid of `user@host/resource`:

- If the element tag is 'presence', the user portion of the jid is returned with '#' prefix
- If the element type is 'groupchat', the user portion of the jid is returned with a '#' prefix
- If the element type is 'chat', but the host is the conference host name, the resource portion of the jid is returned
- Otherwise, the user portion of the jid is returned

Parameters **element** – An instance of *twisted.words.xish.domish.Element*

Returns The channel portion of the XMPP jid, prefixed with '#' if it's a chat room

parse_message(*message*)
Parses the message body from a <message/> element, ignoring any delayed messages. If a message is indeed a delayed message, an empty string is returned

Parameters **message** – A <message/> element, instance of *twisted.words.xish.domish.Element*

Returns The contents of the message, empty string if the message is delayed

parse_nick(*message*)
Parses a nick from a full XMPP jid. This will also take special care to parse a nick as a user jid or a resource from a room jid. For example from `me@jabber.local` would return `me` and `bots@conference.jabber.local/me` would return `me`.

Parameters **message** – A <message/> element, instance of *twisted.words.xish.domish.Element*

Returns The nick portion of the XMPP jid

ping()
Sends an IQ PING to the host server. Useful for establishing a heartbeat/keepalive

set_presence(*presence*)
Sends a <presence/> element to the connected server. Used to indicate online or available status

Parameters **presence** – The presence status string to send to the server

class `helga.comm.xmpp.`**Factory**
XMPP client factory. following *twisted.words.protocols.jabber.client.XMPPClientFactory*. Ensures that a client is properly created and handles auto reconnect if helga is configured for it (see settings *AUTO_RECONNECT* and *AUTO_RECONNECT_DELAY*).

By default the Jabber ID is set using the form `USERNAME@HOST` from *SERVER*, but a specific value can be used with the `JID` key instead.

jid
The Jabber ID used by the client. Configured directly via `JID` in *SERVER* or indirectly as `USERNAME@HOST` from *SERVER*. An instance of *twisted.words.protocols.jabber.jid.JID*.

auth
An instance of *twisted.words.protocols.jabber.client.XMPPAuthenticator* used for password authentication of the server connection.

client
The client instance of *Client*

clientConnectionFailed(*connector*, *reason*)
Handler for when the XMPP connection fails. Handles auto reconnect if helga is configured for it (see settings *AUTO_RECONNECT* and *AUTO_RECONNECT_DELAY*)

Parameters

- **connector** – The twisted conntector
- **reason** – A twisted Failure instance

Raises The given reason unless AUTO_RECONNECT is enabled

clientConnectionLost(*connector*, *reason*)
Handler for when the XMPP connection is lost. Handles auto reconnect if helga is configured for it (see settings *AUTO_RECONNECT* and *AUTO_RECONNECT_DELAY*)

Parameters

- **connector** – The twisted conntector
- **reason** – A twisted Failure instance

Raises The given reason unless AUTO_RECONNECT is enabled

protocol
alias of `XmlStream`

helga.db

pymongo connection objects and utilities

helga.db.**client**
A *pymongo.mongo_client.MongoClient* instance, the connection client to MongoDB

helga.db.**db**
A *pymongo.database.Database* instance, the default MongoDB database to use

helga.db.**connect**()
Connect to a MongoDB instance, if helga is configured to do so (see setting *DATABASE*). This will return the MongoDB client as well as the default database as configured.

Returns A two-tuple of (*pymongo.MongoClient*, *pymongo.database.Database*)

helga.log

Logging utilities for helga

class helga.log.**ChannelLogFileHandler**(*basedir*)
A rotating file handler implementation that will create UTC dated log files suitable for channel logging.

compute_next_rollover()
Based on UTC now, computes the next rollover date, which will be 00:00:00 of the following day. For example, if the current datetime is 2014-10-31 08:15:00, then the next rollover will be 2014-11-01 00:00:00.

current_filename()
Returns a UTC dated filename suitable as a log file. Example: 2014-12-15.txt

doRollover()
Perform log rollover. Closes any open stream, sets a new log filename, and computes the next rollover time.

shouldRollover(*record*)
Returns True if the current UTC datetime occurs on or after the next scheduled rollover datetime. False otherwise.

Parameters **record** – a python log record

class helga.log.**UTCTimeLogFilter**(*name=''*)
A log record filter that will add an attribute `utcnow` and `utctime` to a log record. The former is a utcnow datetime object, the latter is the formatted time of day for utcnow.

filter(*record*)
Filter the log record and add two attributes:

- `utcnow`: the value of *datetime.datetime.utcnow*
- `utctime`: the time formatted string of `utcnow` in the form `HH:MM:SS`

helga.log.**getLogger**(*name*)
Obtains a named logger and ensures that it is configured according to helga's log settings (see *Log Settings*). Use of this is generally intended to mimic *logging.getLogger* with the exception that it takes care of formatters and handlers.

Parameters **name** – The name of the logger to get

helga.log.**get_channel_logger**(*channel*)
Obtains a python logger configured to operate as a channel logger.

Parameters **channel** – the channel name for the desired logger

helga.plugins

Helga's core plugin library containing base implementations for creating plugins as well as utilities for managing plugins at runtime

helga.plugins.**registry**
A singleton instance of *helga.plugins.Registry*

helga.plugins.**ACKS = ['roger', '10-4', 'no problem', 'will do', 'you got it', 'anything you say', 'sure thing', 'ok', 'right-o',**
A collection of pre-canned acknowledgement type responses

class helga.plugins.**Command**(*command=''*, *aliases=None*, *help=''*, *priority=50*, *shlex=False*)
A subclass of *Plugin* for command type plugins (see *Plugin Types*). Command plugins have a default priority of *PRIORITY_NORMAL*

aliases = []
A list of command aliases. If a command 'search' has an alias list ['s'], then 'helga search foo' and 'helga s foo' will both run the command

command = ''
The command string, i.e. 'search' for a command 'helga search foo'

help = ''
An optional help string for the command. This is used by the builtin *help* plugin

parse(*botnick*, *message*)
Parse the incoming message using the current nick of the bot, the defined command string of this object, plus any aliases. Will return the actual command parsed (which could be an alias), plus either whitespaced delimited list of strings that follow the parsed command, or shlex argument list if `shlex` is True.

Generally, this does not need to be implemented by subclasses

Parameters

- **botnick** – the current bot nickname
- **message** – the incoming chat message

Returns two-tuple consisting of the string of parsed command, and an argument list of strings either whitespace delimited or shlex split.

process(*client*, *channel*, *nick*, *message*)
Parses the incoming message and determins if this command should run (i.e. if the primary command or one of the aliases match).

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server

Returns None if the plugin should not run, otherwise the return value of the `run` method

run(*client*, *channel*, *nick*, *message*, *command*, *args*)
Executes this plugin with a given message to generate a response. This should run without regard to whether it should or should not for a given message. Note, that this is where the actual work for the plugin should occur. Subclasses should implement this method.

A return value of None, an empty string, or empty list implies that no response should be sent via chat. A non-empty string, list of strings, or raised *ResponseNotReady* implies a response to be sent.

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server
- **cmd** – The parsed command string which could be the registered command or one of the command aliases
- **args** – The parsed command arguments as a list, i.e. any content following the command. For example: `helga foo bar baz` would be `['bar', 'baz']`

Returns String or list of strings to return via chat. None or empty string or list for no response

shlex = False
A boolean indicating whether or not to use shlex arg string parsing rather than naive whitespace splitting

class helga.plugins.**Match**(*pattern=''*, *priority=25*)
A subclass of *Plugin* for match type plugins (see *Plugin Types*). Matches have a default priority of *PRIORITY_LOW*

match(*message*)
Matches a message against the pattern defined for this class. If the `pattern` attribute is a callable, it is called with the message as its only argument and that value is returned. Otherwise, the pattern attribute is used as a regular expression string argument to `re.findall` and that value is returned.

Parameters **message** – the message received from the server

Returns the result of `re.findall` if pattern is a string, otherwise the return value of calling the `pattern` attribute with the message as a parameter

pattern = ''
A regular expression string used to match against a chat message. Optionally, this attribute can be a callable that accepts a chat message string as its only argument and returns a value that can be evaluated for truthiness.

process(*client*, *channel*, *nick*, *message*)
Processes a message sent by a user on a given channel. This will return None if the message does not match the plugin's pattern, or the return value of `run` if it does match. For this plugin to match an incoming message, the return value of `self.match()` must return value that can be evaluated for truth. Generally, subclasses should not have to worry about this method, and instead, should focus on the implementation of `run`.

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server

Returns None if the plugin should not run, otherwise the return value of the `run` method

run(*client*, *channel*, *nick*, *message*, *matches*)
Executes this plugin with a given message to generate a response. This should run without regard to whether it should or should not for a given message. Note, that this is where the actual work for the plugin should occur. Subclasses should implement this method.

A return value of None, an empty string, or empty list implies that no response should be sent via chat. A non-empty string, list of strings, or raised *ResponseNotReady* implies a response to be sent.

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message

- **message** – The full message string received from the server
- **matches** – The result of `re.findall` if decorated with a regular expression, otherwise the return value of the callable passed to *helga.plugins.match()*

Returns String or list of strings to return via chat. None or empty string or list for no response

`helga.plugins.`**PRIORITY_HIGH = 75**
The value for high priority plugins. Configurable via setting *PLUGIN_PRIORITY_HIGH*

`helga.plugins.`**PRIORITY_LOW = 25**
The value for low priority plugins. Configurable via setting *PLUGIN_PRIORITY_LOW*

`helga.plugins.`**PRIORITY_NORMAL = 50**
The value for normal priority plugins. Configurable via setting *PLUGIN_PRIORITY_NORMAL*

class `helga.plugins.`**Plugin**(*priority=50*)
The base class for helga plugins. There are three main methods of this base class that are important for creating class-based plugins.

`preprocess`

Run by the plugin registry as a preprocessing mechanism. Allows plugins to modify the channel, nick, and/or message that other plugins will receive.

`process`

Run by the plugin registry to allow a plugin to process a chat message. This is the primary entry point for plugins according to the plugin manager, so it should either return a response or not.

`run`

Run internally by the plugin, generally from within the `process` method. This should do the actual work to generate a response. In other words, `process` should handle checking if the plugin should handle a message and then return whatever `run` returns.

decorate(*fn*, *preprocessor=False*)
A helper for decorating a function to handle this plugin. This essentially just monkey patches the `run` or `preprocess` method whith the given function. Decorated functions should accept whatever arguments the subclass implementation sends to its `run` method. Also, instances/subclasses of *Plugin* are kept in a list attribute of the decorated function. This allows chainable decorators that function as intended. Example usage:

```
def my_plugin(*args, **kwargs):
    pass

p = Plugin()
p.decorate(my_plugin)
assert p in my_plugin._plugins
```

Parameters

- **fn** – function to decorate
- **preprocessor** – True if the function should be decorated as a preprocessor

preprocess(*client*, *channel*, *nick*, *message*)
A preprocessing filter for plugins. This allows a plugin to modify a received message prior to that message being handled by this plugin's or other plugin's `process` method.

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server

Returns a three-tuple (channel, nick, message) containing any modifications

priority = 50
The registered priority of the plugin

process(*client*, *channel*, *nick*, *message*)
This method of a plugin is called by helga's plugin registry to process an incoming chat message. This should determine whether or not the plugin `run` method should be called. If so, it should return whatever return value `run` generates. If not, `None` should be returned.

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server

Returns None if the plugin should not run, otherwise the return value of the `run` method

run(*client*, *channel*, *nick*, *message*, **args*, ***kwargs*)
Executes this plugin with a given message to generate a response. This should run without regard to whether it should or should not for a given message. Note, that this is where the actual work for the plugin should occur. Subclasses should implement this method.

A return value of None, an empty string, or empty list implies that no response should be sent via chat. A non-empty string, list of strings, or raised *ResponseNotReady* implies a response to be sent.

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server

Returns None if no response is to be sent back to the server, a non-empty string or list of strings if a response is to be returned

class `helga.plugins.`**Registry**
Simple plugin registry that handles dispatching messages to registered plugins. Plugins can be disabled or enabled per channel. By default, all plugins are loaded, but not enabled on a channel unless it exists

in *DEFAULT_CHANNEL_PLUGINS*. This is done so that potentially annoying plugins can be enabled on-demand. Plugin loading can be limited to a whitelist via *ENABLED_PLUGINS* or restricted to a blacklist via *DISABLED_PLUGINS*.

plugins = {}
A dictionary mapping plugin names to decorated functions or *Plugin* subclasses

enabled_plugins = {}
A dictionary of enabled plugin names per channel, keyed by channel name

all_plugins
A set of all registered plugin names

disable(*channel*, **plugins*)
Disable a plugin or plugins on a desired channel

Parameters

- **channel** – the desired chat channel
- ***plugins** – a list of plugin names to disable

enable(*channel*, **plugins*)
Enable a plugin or plugins on a desired channel

Parameters

- **channel** – the desired chat channel
- ***plugins** – a list of plugin names to enable

get_plugin(*name*)
Get a plugin by name

Parameters **name** – the name of the plugin

Returns a plugin implementation (decorated function or *Plugin* sublclass)

load()
Load all plugins registered via setuptools entry point named `helga_plugins` and initialize them. For example:

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

Note that this loading honors plugin whitelists and blacklists from the settings *ENABLED_PLUGINS* and *DISABLED_PLUGINS* respectively. If there are no whitelisted plugins, nothing is loaded. If a plugin is in the blacklist, it is not loaded. If a plugin is not listed in the whitelist, it is not loaded.

preprocess(*client*, *channel*, *nick*, *message*)
Invoke the `preprocess` method for each plugin on a given channel according to plugin priority. Any exceptions from plugins will be suppressed and logged.

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – the channel from which the message came
- **nick** – the nick of the user sending the message

- **message** – the original message received

Returns a three-tuple (channel, nick, message) containing modifications all preprocessor plugins have made

prioritized(*channel*, *high_to_low=True*)
Obtain a list of enabled plugins for a given channel ordered according to their priority (see *Plugin Priorities*). The default action is to return a list ordered from most important to least important.

Parameters

- **channel** – the chat channel for the enabled plugin list
- **high_to_low** – priority ordering, True for most important to least important.

process(*client*, *channel*, *nick*, *message*)
Invoke the `process` method for each plugin on a given channel according to plugin priority. Any exceptions from plugins will be suppressed and logged. All return values from plugin `process` methods are collected unless the setting *PLUGIN_FIRST_RESPONDER_ONLY* is set to True or a plugin raises *ResponseNotReady*, in which case the first plugin to return a response or raise *ResponseNotReady* will prevent others from processing. All response strings are explicitly converted to unicode.

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – the channel from which the message came
- **nick** – the nick of the user sending the message
- **message** – the original message received

Returns a list of non-empty unicode response strings

register(*name*, *fn_or_cls*)
Register a decorated plugin function or *Plugin* subclass with a given name

Parameters

- **name** – the name of the plugin
- **fn_or_cls** – a decorated plugin function or *Plugin* subclass

Raises TypeError if the `fn_or_cls` argument is not a decorated plugin function or *Plugin* subclass

reload(*name*)
Reloads a plugin with a given name. This is equivalent to finding the registered entry point module and using the python builtin `reload()`.

Parameters **name** – the desired plugin to reload

Returns True if reloaded, False if an exception occurred

exception `helga.plugins.`**ResponseNotReady**
Exception raised by plugins that perform some async operation using twisted deferreds. If the bot is configured to only allow the first plugin response (by default), then any plugin raising this will prevent further plugin execution

(see *Communicating Asynchronously*)

`helga.plugins.`**command**(*command*, *aliases=None*, *help=''*, *priority=50*, *shlex=False*)
A decorator for creating command plugins

Parameters

- **command** – The command string, i.e. 'search' for a command 'helga search foo'
- **aliases** – A list of command aliases. If a command 'search' has an alias list ['s'], then 'helga search foo' and 'helga s foo' will both run the command.
- **help** – An optional help string for the command. This is used by the builtin help plugin.
- **priority** – The priority of the plugin. Default is *PRIORITY_NORMAL*.
- **shlex** – A boolean indicating whether to use shlex arg string parsing rather than naive whitespace splitting.

Decorated functions should follow this pattern:

`helga.plugins.`**`func`**(*client*, *channel*, *nick*, *message*, *cmd*, *args*)

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server
- **cmd** – The parsed command string which could be the registered command or one of the command aliases
- **args** – The parsed command arguments as a list, i.e. any content following the command. For example: `helga foo bar baz` would be `['bar', 'baz']`

Returns String or list of strings to return via chat. None or empty string or list for no response

`helga.plugins.`**`match`**(*pattern*, *priority=25*)

A decorator for creating match plugins

Parameters

- **pattern** – A regular expression string used to match against a chat message. Optionally, this argument can be a callable that accepts a chat message string as its only argument and returns a value that can be evaluated for truthiness.
- **priority** – The priority of the plugin. Default is *PRIORITY_LOW*

Decorated match functions should follow this pattern:

`helga.plugins.`**`func`**(*client*, *channel*, *nick*, *message*, *matches*)

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server
- **matches** – The result of `re.findall` if decorated with a regular expression, otherwise the return value of the callable passed

Returns String or list of strings to return via chat. None or empty string or list for no response

helga.plugins.**preprocessor**(*priority=50*)
A decorator for creating preprocessor plugins

Parameters **priority** – The priority of the plugin. Default is *PRIORITY_NORMAL*

Decorated preprocessor functions should follow this pattern:

helga.plugins.**func**(*client*, *channel*, *nick*, *message*, *matches*)

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **channel** – The channel from which the message was received. This could be a public channel like '#foo', or in the event of a private message, could be the nick of the user sending the message
- **nick** – The nick of the user sending the message
- **message** – The full message string received from the server

Returns a three-tuple (channel, nick, message) containing any modifications

helga.plugins.**random_ack**()
Returns a random choice from *ACKS*

helga.plugins.webhooks

Webhook HTTP server plugin and core webhook API

Webhooks provide a way to expose HTTP endpoints that can interact with helga. A command plugin manages an HTTP server that is run on a port specified by setting *helga.settings.WEBHOOKS_PORT* (default 8080). An additional, optional setting that can be used for routes requiring HTTP basic auth is *helga.settings.WEBHOOKS_CREDENTIALS*, which should be a list of tuples, where each tuple is a pair of (username, password).

Routes are URL path endpoints. On the surface they are just python callables decorated using *@route*. The route decorated must be given a path regex, and optional list of HTTP methods to accept. Webhook plugins must be registered in the same way normal plugins are registered, using setuptools entry_points. However, they must belong to the entry_point group `helga_webhooks`. For example:

```
setup(entry_points={
    'helga_webhooks': [
        'api = myapi.decorated_route'
    ]
})
```

For more information, see *Webhooks*

exception helga.plugins.webhooks.**HttpError**(*code*, *message=None*, *response=None*)
A basic HTTP error.

@type status: L{bytes} @ivar status: Refers to an HTTP status code, for example C{http.NOT_FOUND}.

@type message: L{bytes} @param message: A short error message, for example "NOT FOUND".

@type response: L{bytes} @ivar response: A complete HTML document for an error page.

class helga.plugins.webhooks.**WebhookPlugin**(**args*, ***kwargs*)
A command plugin that manages running an HTTP server for webhook routes and services. Usage:

```
helga webhooks (start|stop|routes)
```

Both `start` and `stop` are privileged actions and can start and stop the HTTP listener for webhooks respectively. To use them, a user must be configured as an operator. The `routes` subcommand will list all of the URL routes known to the webhook listener.

Webhook routes are generally loaded automatically if they are installed. There are whitelist and blacklist controls to limit loading webhook routes (see *ENABLED_WEBHOOKS* and *DISABLED_WEBHOOKS*)

add_route(*fn*, *path*, *methods*)
Adds a route handler function to the root web resource at a given path and for the given methods

Parameters

- **fn** – the route handler function
- **path** – the URL path of the route
- **methods** – list of HTTP methods that the route should respond to

control(*action*)
Control the running HTTP server. Intended for helga operators.

Parameters **action** – the action to perform, either 'start' or 'stop'

list_routes(*client*, *nick*)
Messages a user with all webhook routes and their supported HTTP methods

Parameters

- **client** – an instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*
- **nick** – the nick of the chat user to message

class helga.plugins.webhooks.**WebhookRoot**(**args*, ***kwargs*)
The root HTTP resource the webhook HTTP server uses to respond to requests. This manages all registered webhook route handlers, manages running them, and manages returning any responses generated.

add_route(*fn*, *path*, *methods*)
Adds a route handler function to the root web resource at a given path and for the given methods

Parameters

- **fn** – the route handler function
- **path** – the URL path of the route
- **methods** – list of HTTP methods that the route should respond to

chat_client = None
An instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*

render(*request*)
Renders a response for an incoming request. Handles finding and dispatching the route matching the incoming request path. Any response string generated will be explicitly encoded as a UTF-8 byte string.

If no route patch matches the incoming request, a 404 is returned.

If a route is found, but the request uses a method that the route handler does not support, a 405 is returned.

Parameters **request** – The incoming HTTP request, `twisted.web.http.Request`

Returns a string with the HTTP response content

routes = None
A dictionary of regular expression URL paths as keys, and two-tuple values of allowed methods, and the route handler function

helga.plugins.webhooks.**authenticated**(*fn*)
Decorator for declaring a webhook route as requiring HTTP basic authentication. Incoming requests validate a supplied basic auth username and password against the list configured in the setting *WEBHOOKS_CREDENTIALS*. If no valid credentials are supplied, an HTTP 401 response is returned.

Parameters **fn** – the route handler to decorate

helga.plugins.webhooks.**route**(*path*, *methods=None*)
Decorator to register a webhook route. This requires a path regular expression, and optionally a list of HTTP methods to accept, which defaults to accepting `GET` requests only. Incoming HTTP requests that use a non-allowed method will receive a 405 HTTP response.

Parameters

- **path** – a regular expression string for the URL path of the route
- **methods** – a list of accepted HTTP methods for this route, defaulting to `['GET']`

Decorated routes must follow this pattern:

helga.plugins.webhooks.**func**(*request*, *client*)

Parameters

- **request** – The incoming HTTP request, `twisted.web.http.Request`
- **client** – The client connection. An instance of *helga.comm.irc.Client* or *helga.comm.xmpp.Client*

Returns a string HTTP response

helga.settings

Default settings and configuration utilities

Chat Settings

Settings that pertain to how helga operates with and connects to a chat server

helga.settings.**NICK = 'helga'**
The preferred nick of the bot instance. For XMPP clients, this will be used when joining rooms.

helga.settings.**CHANNELS = [('#bots',)]**
A list of channels to automatically join. You can specify either a single channel name or a two-tuple of channel name, and password. For example:

```
CHANNELS = [
    '#bots',
    ('#foo', 'password'),
]
```

Note that this setting is only for hardcoded autojoined channels. Helga also responds to /INVITE commands as well offers a builtin plugin to configure autojoin channels at runtime (see *operator*)

For XMPP/HipChat support, channel names should either be the full room JID in the form of `room@host` or a simple channel name prefixed with a '#' such as `#room`. Depending on the configuration, the room JID will be constructed using the `MUC_HOST` value of the `SERVER` setting or by prefixing 'conference.' to the `HOST` value.

helga.settings.**SERVER = {'HOST': 'localhost', 'TYPE': 'irc', 'PORT': 6667}**
Dictionary of connection details. At a minimum this should contain keys `HOST` and `PORT` which default to 'localhost' and 6667 respectively for irc. Optionally, you can specify a boolean key `SSL` if you require helga to connect via SSL. You may also specify keys `USERNAME` and `PASSWORD` if your server requires authentication. For example:

```
SERVER = {
    'HOST': 'localhost',
    'PORT': 6667,
    'SSL': False,
    'USERNAME': 'user',
    'PASSWORD': 'pass',
}
```

Additional, optional keys are supported for different chat backends:

- `TYPE`: the backend type to use, 'irc' or 'xmpp'
- `MUC_HOST`: the MUC group chat domain like 'conference.example.com' for group chat
- `JID`: A full jabber ID to use instead of USERNAME (xmpp only)

helga.settings.**AUTO_RECONNECT = True**
A boolean indicating if the bot automatically reconnect on connection lost

helga.settings.**AUTO_RECONNECT_DELAY = 5**
An integer for the time, in seconds, to delay between reconnect attempts

helga.settings.**RATE_LIMIT = None**
IRC Only. An integer indicating the rate limit, in seconds, for messages sent over IRC. This may help to prevent flood, but may degrade the performance of the bot, as it applies to every message sent to IRC.

Core Settings

Settings that pertain to core helga features.

helga.settings.**OPERATORS = []**
A list of chat nicks that should be considered operators/administrators

helga.settings.**DATABASE = {'HOST': 'localhost', 'DB': 'helga', 'PORT': 27017}**
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 collection to use. These values default to 'localhost', 27017, and 'helga' respectively. Both 'USERNAME' and 'PASSWORD' can be specified if MongoDB requires authentication. For example:

```
DATABASE = {
    'HOST': 'localhost',
    'PORT': 27017,
    'DB': 'helga',
    'USERNAME': 'foo',
    'PASSWORD': 'bar',
}
```

Log Settings

helga.settings.**LOG_LEVEL = 'DEBUG'**
A string for the logging level helga should use for process logging

helga.settings.**LOG_FILE = None**
A string, if set, a string indicating the log file for python logs. By default helga will log directly to stdout

helga.settings.**LOG_FORMAT = '%(asctime)-15s [%(levelname)s] [%(name)s:%(lineno)d]: %(message)s'**
A string that is compatible with configuring a python logging formatter.

Channel Log Settings

See *Channel Logging*

helga.settings.**CHANNEL_LOGGING = False**
A boolean, if True, will enable conversation logging on all channels

helga.settings.**CHANNEL_LOGGING_DIR = '.logs'**
If *CHANNEL_LOGGING* is enabled, this is a string of the directory to which channel logs should be written.

helga.settings.**CHANNEL_LOGGING_HIDE_CHANNELS = []**
A list of channel names (either with or without a '#' prefix) that will be hidden in the browsable channel log web ui.

Plugin and Webhook Settings

Settings that control plugin and/or webhook behaviors. See *Plugins* or *Webhooks*

helga.settings.**ENABLED_PLUGINS = True**
A list of plugin names that should be loaded by the plugin manager. This effectively serves as a mechanism for explicitly including plugins that have been installed on the system. If this value is True, the plugin manager will load any plugin configured with an entry point and make it available for use. If it is None, or an empty list, no plugins will be loaded. See *Plugins* for more information.

helga.settings.**DISABLED_PLUGINS = []**
A list of plugin names that should NOT be loaded by the plugin manager. This effectively serves as a mechanism for explicitly excluding plugins that have been installed on the system. If this value is True, the plugin manager will NOT load any plugin configured with an entry point. If it is None, or an empty list, no plugins will be blacklisted. See *Plugins* for more information.

helga.settings.**DEFAULT_CHANNEL_PLUGINS = True**
A list of plugin names that should be enabled automatically for any channel. If this value is True, all plugins installed will be enabled by default. If this value is None, or an empty list, no plugins will be enabled on channels by default. See *Plugins* for more information.

helga.settings.**ENABLED_WEBHOOKS = True**
A list of whitelisted webhook names that should be loaded and enabled on process startup. If this value is True, then all webhooks available are loaded and made available. An empty list or None implies that no webhooks will be made available. See *Webhooks* for more details.

helga.settings.**DISABLED_WEBHOOKS = None**
A list of blacklisted webhook names that should NOT be loaded and enabled on process startup. If this value is True, then all webhooks available are loaded and made available. An empty list or None implies that no webhooks will be made available. See *Webhooks* for more details.

helga.settings.**PLUGIN_PRIORITY_LOW = 25**
Integer value for 'low' priority plugins (see *Plugin Priorities*)

helga.settings.**PLUGIN_PRIORITY_NORMAL = 50**
Integer value for 'normal' priority plugins (see *Plugin Priorities*)

helga.settings.**PLUGIN_PRIORITY_HIGH = 75**
Integer value for 'high' priority plugins (see *Plugin Priorities*)

helga.settings.**PLUGIN_FIRST_RESPONDER_ONLY = True**
A boolean, if True, the first response received from a plugin will be the only message sent back to the chat server. If False, all responses are sent.

helga.settings.**COMMAND_PREFIX_BOTNICK = True**
If a boolean and True, command plugins can be run by asking directly, such as 'helga foo_command'. This can also be a string for specifically setting a nick type prefix (such as @NickName for HipChat)

helga.settings.**COMMAND_PREFIX_CHAR = '!'**
A string char, if non-empty, that can be used to invoke a command without requiring the bot's nick. For example 'helga foo' could be run with '!foo'.

helga.settings.**COMMAND_ARGS_SHLEX = False**
A boolean that controls the behavior of argument parsing for command plugins. If False, command plugin arguments are parsed using a naive whitespace split. If True, they will be parsed using *shlex.split*. See *Command Plugins* for more information. The default is False, but this shlex parsing will be the only supported means of argument string parsing in a future version.

helga.settings.**WEBHOOKS_PORT = 8080**
The integer port the webhooks plugin should listen for http requests.

helga.settings.**WEBHOOKS_CREDENTIALS = []**
List of two-tuple username and passwords used for http webhook basic authentication

helga.settings.**configure**(*overrides*)
Applies custom configuration to global helga settings. Overrides can either be a python import path string like 'foo.bar.baz' or a filesystem path like 'foo/bar/baz.py'

Parameters **overrides** – an importable python path string like 'foo.bar' or a filesystem path to a python file like 'foo/bar.py'

helga.util.encodings

Utilities for working with unicode and/or byte strings

helga.util.encodings.**from_unicode**(*unistr*, *errors='ignore'*)
Safely convert unicode to a byte string by first checking if it already is a byte string before encoding. This function assumes UTF-8 for byte strings and by default will ignore any encoding errors.

Parameters

- **unistr** – either unicode or a byte string
- **errors** – a string indicating how encoding errors should be handled (i.e. 'strict', 'ignore', 'replace')

helga.util.encodings.**from_unicode_args**(*fn*)
Decorator used to safely convert a function's positional arguments from unicode to byte strings

helga.util.encodings.**to_unicode**(*bytestr*, *errors='ignore'*)
Safely convert a byte string to unicode by first checking if it already is unicode before decoding. This function assumes UTF-8 for byte strings and by default will ignore any decoding errors.

Parameters

- **bytestr** – either a byte string or unicode string
- **errors** – a string indicating how decoding errors should be handled (i.e. 'strict', 'ignore', 'replace')

helga.util.encodings.**to_unicode_args**(*fn*)
Decorator used to safely convert a function's positional arguments from byte strings to unicode

CHAPTER 7

Indices and Tables

- genindex
- modindex
- search

Python Module Index

h

Index

R

S

T

U

W