brutal Documentation

Release 0.3.3

Corey Bertram

December 12, 2015

Contents

Release v0.3.3. - ALPHA release

There are multiple ways to write a chat bot in python, but most thread out to support user code and mix and match client libs to support multiple protocols. brutal is an attempt to change this using the twisted framework. All network code is written using twisted libraries for native async support.

CHAPTER 1

Code

brutal is actively developed on GitHub. You can find the repo here.

CHAPTER 2

Feature Support

- IRC
- XMPP
- Everything runs async by default.
- Thread pool support exists for explicitly defined blocking code

CHAPTER 3

User Guide

3.1 Quickstart

This will guide you through the steps of getting up and running with your very own bot.

3.1.1 Installation

First you will have to make sure you have brutal.

Installing brutal is simple with pip:

```
$ pip install brutal
```

3.1.2 Create a Bot

Once you have brutal installed, you must start your new bot using the overlord.:

```
$ brutal-overlord spawn <bot_name>
```

Where `<bot_name>` is the default nick of your new bot. This will create a new directory and populate it with the base configuration that you will need.

3.1.3 Configuration

Once you have a bot, you will have to modify the `<bot_name>/config.py` file to get started.

3.2 Plugins

3.2.1 Basic Plugins

Writing brutal plugins is meant to be easy. By nature, a bot responds to events within a chat system. Everything brutal sees is an `Event` object and it is possible to write plugins that only respond to a specific `!command` in a channel or an event parser that responds to certain or all types of events.

basic commands

If you just want to make a command that the bot will respond to, all you have to do is:

```
from brutal.core.plugin import cmd
@cmd
def ping(event):
    return 'pong, got {0!r}'.format(event)

@cmd
def testargs(event):
    return 'you passed in args: {0!r}'.format(event.args)
```

basic events

If you want to register a parser to handle events, you can easily do this as well:

```
from brutal.core.plugin import event

@event
def test_event_parser(event):
    return 'EVENT!!! {0!r}'.format(event)
```

This will respond to every action a bot sees in a channel with a message.

basic matching

It is also possible to write a parser that responds to a given regex:

```
from brutal.core.plugin import match

@match(regex=r'^hi$')
def matcher(event):
    return 'Hello to you!'
```

This will respond every time someone says `hi` in the channel.

blocking code

Sometimes its hard to write async code. Rather than limit you to other bot frameworks, brutal has built in support for code that blocks. You simply have to let it know that your code is not async by passing in `thread=True`:

```
import time  # blocking lib
from brutal.core.plugin import cmd, event

@cmd(thread=True)
def sleep(event):
    time.sleep(5)  # blocks
    return 'im sleepy...'

@event(thread=True)
def sleepevent(event):
    time.sleep(7)  # blocks
    return 'SOOOOOO sleepy'
```

Every time these trigger, they get put into a global thread pool that brutal maintains for you. If your code blocks for too long you risk the chance of the thread pool getting incredibly backed up so some thought must be put into what you're blocking for. It is recommended that you try to write asynchronous code using the brutal and twisted utilities.

3.2.2 Plugin Classes

Sometimes a simple function just isn't enough. If you need to store state with your bot functionality, it is recommended that you extend the BotPlugin class. Every bot that you define within brutal will have its own instance of this class upon startup, each with its own state.

basic use of BotPlugin

```
import time
from brutal.core.plugin import BotPlugin, cmd, event, match, threaded

class TestPlugin(BotPlugin):
    def setup(self, *args, **kwargs):
        self.log.debug('SETUP CALLED')
        self.count = 0
        self.loop_task(5, self.test_loop, now=False)
        self.delay_task(10, self.future_task)

    def future_task(self):
        self.log.info('testing future task')
        return 'future!'

    @threaded
    def test_loop(self):
        self.log.info('testing looping task')
        return 'loop!'

    def say_hi(self, event=None):
        self.msg('from say_hi: {0!r}'.format(event), event=event)
        return 'hi'

    @threaded
    def say_delayed_hi(self, event=None):
        self.msg('from say_hi_threaded, sleeping for 5: {0!r}'.format(event), event=event)
        time.sleep(5)
        return 'even more delayed hi'

    @cmd
    def runlater(self, event):
        self.delay_task(5, self.say_hi, event=event)
        self.delay_task(5, self.say_delayed_hi, event=event)
        return 'will say hi in 5 seconds'

    @cmd
    def count(self, event):
        self.count += 1
        return 'count {1!r} from class! got {0!r}'.format(event, self.count)

    @cmd(thread=True)
    def inlinemsg(self, event):
        self.msg('sleeping for 5 seconds!', event=event)
```

```
        time.sleep(5)
        return 'done sleeping!'
```

3.2.3 Test Console

In order to ease some of the pain while developing plugins, brutal provides a basic test console for local development.

TODO: add details here.

CHAPTER 4

License

CHAPTER 5

Indices and tables

- genindex
- modindex
- search