ZenIRCBot Documentation

Release 2.2.1

Wraithan

February 26, 2014

Contents

Contents:

CHAPTER 1

Install

Installing ZenIRCBot will one day be much simpler, but for now there are a number of steps to it.

To start with, the bot is written in both JavaScript using the Node platform, and there is a version that is written in Python. Two of the services are written in or rely on Python the rest are written in JavaScript.

If you are on Ubuntu, then the quickest and easiest way to get a recent version of node is to use a version from a PPA (Personal Package Archive). This PPA containes the latest stable versions of nodejs.

To install from the PPA make sure you have `python-software-properties` installed first; otherwise, you might be missing the `add-apt-repository` command. (Install `software-properties-common` on Quantal).

Then run:

```
$ sudo add-apt-repository ppa:chris-lea/node.js
$ sudo apt-get update
$ sudo apt-get install nodejs npm
```

If you don't want to use the PPA then you can always compile node and npm yourself.

1.1 Configuring the bot

The config for the bot can be found in *bot.json* which is a JSON file that both the Node.js and Python bots use.

If you don't have a *bot.json*, copy *bot.json.dist* into its place like so:

```
$ cp bot.json{.dist,}
```

Then modify it and fill in the values with your own.

Note: Despite the option being *servers* ZenIRCBot currently only supports 1 server. It is named as such for future compatibility

1.2 Getting the node.js bot running

To start with you'll need to install Node, npm and Redis. Once you have those you'll need to use npm to install the dependencies:

```
$ npm install
```

Then all you need to do is start the bot:

```
$ node bot.js
```

Voilà, your bot should connect, join the channels in the config and go forth on its merry way. It wont do anything interesting until you start up services. You can find information on starting up *Services*.

1.3 Getting the python bot running

Warning: This version of the bot is less battle tested than the node version. This doesn't mean you shouldn't use it, just know that these instructions may change in the near future. Also it doesn't use all of the options in the `bot.json`

To start with you'll need to install Python, virtualenv, libevent (libevent-dev if you are on ubuntu) and Redis (all three provided by your OS package manager). Once you have those you'll want to make and activate the virtualenv to keep your libraries you installed for ZenIRCBot (if you have virtualenvwrapper installed already feel free to use it of course):

```
$ virtualenv zib
$ source zib/bin/activate
```

Then use pip to install the dependencies:

```
$ pip install -r requirements.txt
```

Then all you need to do is run:

```
$ python bot.py
```

Voilà, your bot should connect, and go forth on its merry way. It wont do anything interesting until you start up services. You can find information on starting up *Services*.

1.4 Getting the clojure bot running

Warning: This version of the bot is less battle tested than the node version. This doesn't mean you shouldn't use it, just know that these instructions may change in the near future. Also it doesn't use all of the options in the `bot.json`

To start with you'll need to install Clojure, Leiningen and Redis. Once you have those installed you'll check out the clojure bot with:

```
$ git submodule init
$ git submodule update
```

Then you'll start the bot with:

```
$ cd clojurebot
$ lein trampoline run -m zenircbot-clojure.core
```

Voilà, your bot should connect, join the channels in the config and go forth on its merry way. It wont do anything interesting until you start up services. You can find information on starting up *Services*.

CHAPTER 2

Services

This is the documentation for the individual services that come with ZenIRCBot. For all of the Node.js based services you'll need to have the node redis library installed. For the Python based services you'll need to need to have the python redis library installed:

```
$ npm install redis # for Node
```

```
$ pip install redis # for Python
```

2.1 Admin

This is a service for doing basic things like starting and stopping other services or restarting the bot. You'll need to also have semantics running as it provides the `directed_privmsg` type that *admin.js* relies on. It is written in Node.js and also relies on forever:

```
$ npm install forever
```

2.1.1 Config

The config is very simple, a single option called `services` which is a list of services you want started when you start *admin.js*

2.1.2 Commands

start <service name> This will start the specified service.

restart <service name> This will restart the specific service.

stop <service name> This will restart the specific service.

pull This will pull down new code.

2.2 Github

This service relies on the weblistener service to pass along the post body with the service set in the JSON envelope. It is written in Node.js

2.3 Jira Feed

This service does a check on the specified JIRA issue RSS feed and posts to the channel whenever an issue is created, closed, or reopened. It is written in Python.

2.4 Jira Ticket

This service watches for something formated with 2 letters, a dash, then numbers. For example BH-1234, it takes that, and appends it to the specified JIRA URL and says it back to the channel so you get links to issues automatically. It is written in Node.js.

2.5 Release

This service is akin to the github service in that it relies on the weblistener service to send it data when something is posted to the port on the machine running the weblistener service. The post body should look like:

```
payload: "{
    branch: 'feature/cool_stuff',
    status: 'started',
    hostname: 'staging-04',
}"
```

Where payload is the post variable, and what it contains is a JSON string with the attributes branch, status, and hostname. And it will then emit something like:

```
release of feature/cool_stuff started on staging-04
```

To the channel specified in the config. It is written in Node.js.

2.6 Semantics

This service adds another message type that is sent out over *in* which is the *directed_privmsg* type. These are messages that have been determined to have been sent to the bot via the following two forms:

```
ZenIRCBot: commands
!commands
```

It will also grab messages that were sent directly to the bot via a PM. It sends the standard envelope like the *privmsg* type. Its *data* attribute is slightly different though:

```
"data": {
    "raw_message": "!commands",
    "message": "commands",
    "sender": "Wraithan",
    "channel": "#pdxbots"
}
```

It strips the the way it was determined to be addressed to the bot so you can listen specifically for commands to come through rather than having to check all possible methods for a person to send a direct message. It is written in Node.js.

2.7 Troll

This service is a basic trolling service. It is written in Node.js.

2.7.1 Commands

ls Responds with a funny picture.

irssi Responds with a suggestion to use weechat.

2.8 TWSRS

2.8.1 That's What She Really Said

This is a service inspired by talkbackbot and steals its quote DB which it got a lot of from this quote source.

2.8.2 Commands

That's what she said Responds with a quote from a famous woman when this is said in a channel.

twsrs This is an actual command and allows one to get quotes without having to say that's what she said.

2.9 Weblistener

This is a service that passes along post data to the *web_in* channel in redis in the format of:

```
body: {
    payload: "JSON String",
    app: 'whatever-path-on-the-url-posted-to',
}
```

Where payload is the POST body and app is http://example.com/whatever-path-on-the-url-posted-to for example. It is written in Node.js and also relies on having express installed:

```
$ npm install express
```

CHAPTER 3

Developing Services

3.1 Service Model

ZenIRCBot is a bit unique in how it works. A lot of IRC bots have plugins and you put the in the plugin directory, they get loaded when the bot starts. This model has worked pretty well in past for bots, the down side being you have to either have a really complex system to be able to dynamically load and unload plugins on the fly, or restart the bot in order to load new plugins. Also the plugins had to be written in the same language as the bot or one of the explicitly supported languages.

This bot on the other hand has no plugins, and adding or removing functionality has nothing to do with core bot itself. Not just that, but what you can do with the plugins is only limited by your imagination as they are their own processes.

When you start bot.js it connects to redis and subscribes to a pub/sub channel called 'out'. Any messages that come from IRC are published to 'in'. Then when you start a service, if it needs to know/process IRC messages it subscribes to the 'in' channel. If it wants to interact with the bot in some fashion, it publishes to the 'out' channel. Decoupling in this way allows for services to be written in any language, be running constantly or just as needed.

3.2 Writing Your Own

When writing your own service, the easiest way to do so is to place it in the `services` directory. This way you can use **admin.js_** to start, restart, and stop your service. Also you get access to `lib/api` which currently has nodejs and python versions. The things in `lib/api` are mostly convenience functions. The details of what is in there will be discussed later.

First lets look at the low level protocol. Messages all have the same envelope:

```
{
    "version": 1,
    "type": "",
    "data": {
        ...
    }
}
```

3.2.1 In messages

The possible `in` messages.

"privmsg"
Sent whenever a privmsg comes in:

```
{
    "version": 1,
    "type": "privmsg",
    "data": {
        "sender": "",
        "channel": "",
        "message": ""
    }
}
```

"privmsg_action"
Sent whenever an "ACTION" comes in:

```
{
    "version": 1,
    "type": "privmsg_action",
    "data": {
        "sender": "",
        "channel": "",
        "message": ""
    }
}
```

The `message` property contains the text portion of what was sent, the IRC-level "u0001ACTION" prefix is already removed.

"join"
Sent whenever someone joins a channel:

```
{
    "version": 1,
    "type": "join",
    "data": {
        "sender": "",
        "channel": ""
    }
}
```

"part"
Sent whenever someone leaves a channel:

```
{
    "version": 1,
    "type": "part",
    "data": {
        "sender": "",
        "channel": ""
    }
}
```

"quit"
Sent whenever someone quits:

```
{
    "version": 1,
    "type": "quit",
    "data": {
        "sender": ""
    }
}
```

"topic"

Sent whenever a channel's topic is changed:

```
{
    "version": 1,
    "type": "topic",
    "data": {
        "sender": "",
        "channel": "",
        "topic": ""
    }
}
```

"names"

Sent whenever a list of names is received for a channel.

If you had nicks: Wraithan, zenircbot, and @aaronpk in #pdxbots:

```
{
    "version": 1,
    "type": "names",
    "data": {
        "channel": "#pdxbots",
        "nicks": {
            "Wraithan": "",
            "zenircbot": "",
            "aaronpk": "@"
        }
    }
}
```

This can be triggered by sending a raw command of "NAMES #channel"

3.2.2 Out messages

The possible `out` messages.

"privmsg"

Used to have the bot say something:

```
{
    "version": 1,
    "type": "privmsg",
    "data": {
        "to": "",
        "message": ""
    }
}
```

"privmsg_action"

Used to have the bot send an action:

```
{
    "version": 1,
    "type": "privmsg_action",
    "data": {
        "to": "",
        "message": ""
    }
}
```

"raw"

Used to have the bot send a raw string to the IRC server:

```
{
    "version": 1,
    "type": "raw",
    "data": {
        "command": ""
    }
}
```

3.3 API Library

Warning: The following API is depreciated, use the new API libraries instead.

These are the functions that can be found in the python and node.js api library.

send_privmsg(*channel*, *message*)

Arguments

- **channel** (*string*) – The channel to send the message to.
- **message** (*string*) – The message to send.

This is a helper so you don't have to handle the JSON or the envelope yourself.

send_admin_message(*message*)

Arguments

- **message** (*string*) – The message to send.

This is a helper function that sends the message to all of the channels defined in `admin_spew_channels`.

register_commands(*script*, *commands*)

Arguments

- **script** (*string*) – The script with extension that you are registering.
- **commands** (*list*) – A list of objects with name and description attributes used to reply to a commands query.

This will notify all `admin_spew_channels` of the script coming online when the script registers itself. It will also setup a subscription to the 'out' channel that listens for 'commands' to be sent to the bot and responds with the list of script, command name, and command description for all registered scripts.

load_config(*name*)

Arguments

- **name** (*string*) – The JSON file to load.

Returns An native object with the contents of the JSON file.

This is a helper so you don't have to do the file IO and JSON parsing yourself.

Note: If you port `zenircbot-api` to another language, please send a pull request with it, I'll gladly add it and maintain it to stay up to date with any protocol changes.

CHAPTER 4

Frequently Asked Questions

4.1 Can I run 2 bots in one Redis instance?

No, even if you specify different database numbers. The issue is that the pub/sub parts of Redis are shared between all databases in an instance. Instead you'll have to run a Redis instance and set of services per bot.

CHAPTER 5

Future of ZenIRCBot

5.1 Deprecation

Deprecation protocol is as such: A feature is marked as deprecated. It then has warnings attached. In the next version the feature has louder warnings about using it. The version after that it is removed and no longer available.

5.1.1 2.2

- Use of services/lib/api.* is now deprecated Please instead install one of the libraries available as part of zenircbot-api

CHAPTER 6

Contributing

6.1 License

ZenIRCBot is covered by an MIT license, any code contributed will be required to be covered by this license as well.

6.2 Issues

The GitHub issue tracker is the prefered method of reporting an issue. Please do not apply any tags or milestones, I will do that myself when I triage the issue.

6.3 Code

If you write code and would like it integrated with the code base please follow the following steps:

1. Fork the repo
2. Create a branch in your repo
3. Do you bug fix/feature add/hackery
4. Send a pull request from that branch
5. Do not delete your fork until the pull request has been accepted or declined in such a way that you do not want to continue development on it.

6.4 Tests

There are some integration tests in the tests folder. You will need to have circus, and ngIRCd installed in order to use them. You run them like so:

```
$ cd tests/
$ ./test.sh
```

This will bootstrap an environment that includes an IRC daemon and redis running on non standard ports. Then it fires up the bot then runs the tests. So far this works on my systems, please open an issue if you find that it doesn't work for you.

Do not feel obligated to add tests for services, only if you are expanding the protocol that the bot itself is away of.

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*