Skybeard Documentation

Release 1

Lance Maverick

January 17, 2017

Contents:

CHAPTER 1

Introduction

Skybeard is a plug-in based bot for telegram, and uses the telepot library.

CHAPTER 2

Quickstart guide

2.1 Installation

It is recommended to use a `virtualenv` for Skybeard. Create and activate the virtual environment with

```
virtualenv venv
source venv/bin/activate
```

then install the base requirements with

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

You will then need to make a `config.py`. An example `config.py` is provided so you can simply:

```
cp config.py.example config.py
```

2.2 Running Skybeard

To run skybeard define your key in the environment variable `$TG_BOT_TOKEN` or as an argument with `-k` and run `main.py`. this can be done easily e.g.:

```
./main.py -k 99121185:RUE-UAa7dsEaagAKkysPDjqa2X7KxX48e
```

2.3 Skybeard's many beards

Skybeard source documentation: http://skybeard-2.readthedocs.io/en/latest/ Skybeard wears many beards. The bot will automatically load any “beard” (a plug-in) that is placed in the beards folder. Beards are typically structured like so:

```
beards
|
|___myBeard
    |    __init__.py
    |    config.py
    |    requirements.txt
    |    ...
    |
    |___docs
```

```
        |     README
        |     ...
```

In this example the `myBeard` folder containts a `requirements.txt` for any additonal dependencies so they can be pipped, a `config.py` file for configuration of the beard and settings and the `__init__.py` which contains the class that that is the interface between the plug-in and skybeard. This interface class inherits from skybeard.beards.BeardChatHandler`which handles the mounting of the plug-in, registering of commands etc, and also the`telepot.aio.helper.ChatHandler‘.

The folder can also contain any other python modules and files that are needed for the plugin.

2.4 Growing a new beard

Creating a new beard requires knowledge of the **telepot** telegram API, see: http://telepot.readthedocs.io/en/latest/

An example async plug-in that would echo the user’s message would look like this:

```python
import telepot
import telepot.aio
from skybeard.beards import BeardChatHandler


class EchoPlugin(BeardChatHandler):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        #register command "/hello" to dispatch to self.say_hello()
        self.register_command("hello", self.say_hello)

    #is called when "/hello" is sent
    async def say_hello(self, msg):
        name = msg['from']['first_name']
        await self.sender.sendMessage('Hello {}!'.format(name))

    #is called every time a message is sent
    async def on_chat_message(self, msg):
        text = msg['text']
        await self.sender.sendMessage(text)
        await super().on_chat_message(msg)
```

This plug-in will greet the user when they send “/hello” to Skybeard by using the `register_command()` method of the `BeardChatHandler` and will also echo back any text the user sends by overwriting the `on_chat_message()` method (and calling the base method with `super()` afterwards).

See the examples folder for examples of callback functionality, timers, and regex predication.

CHAPTER 3

Project Modules

3.1 beards Module

3.1.1 `beards` Module

Handles the loading and running of skybeard plugins. architecture inspired by: http://martyalchin.com/2008/jan/10/simple-plugin-framework/ and http://stackoverflow.com/a/17401329

class `beards.`**`Beard`**`(`*name*, *bases*, *attrs*`)`

Bases: `type`

Metaclass for creating beards.

`beards` = []

`register``(`*beard*`)`

Add beard to internal list of beards.

class `beards.`**`BeardChatHandler`**`(`**args*, ***kwargs*`)`

Bases: `telepot.aio.helper.ChatHandler`

Chat handler for beards.

This is the primary interface between skybeard and any plug-in. The plug-in must define a class that inherets from BeardChatHandler.

This class should overwrite __commands__ with a list of tuples that route messages containing commands, or if they pass certain “Filters” (see skybeard.beards.Filters). E.g:

‘‘‘Python __commands__ = [

(‘mycommand’, ‘my_func’, ‘this is a help message’), (Filters.location, ‘my_other_func’, ‘another help message’)]

‘‘‘

In this case, when the bot receives the command “/mycommand”, it will call self.my_func(msg) where msg is a dict containing all the message information. The filter (from skybeard.beards) will call self.my_other_func(msg) whenever “msg” contains a location. The help messages are collected by the help functions and automatically formatted and sent when a user sends /help to the bot.

Instances of the plug-in classes are created when required (such as when a filter is passed, a command or a regex pattern for the bot is matched etc.) and they are destructed after a set timeout. The default is 10 seconds, but this can be overwritten with, for example

_timeout = 90

The class should also define a __userhelp__ string which will be used in the auto help message generation.

deserialize(*data*)
Deserializes the callback data

classmethod get_name()
Get the name of the beard (e.g. cls.__name__).

get_username()
Returns the username of the bot

on_chat_message(*msg*)
Default on_chat_message for beards.

Can be overwritten in order to define the behaviour of the plug-in whenever any message is received.

NOTE: super().on_chat_message(msg) must be called in the overwrite to preserve default behaviour. This is usually done after custom behaviour, e.g.

‘‘‘Python async def on_chat_message(self, msg):

await self.sender.sendMessage(“I got your message!”)

super().on_chat_message(msg)

‘‘‘

on_close(*e*)
Removes per beard logger handler and calls telepot default on_close.

register_command(*pred_or_cmd*, *coro*, *hlp=None*)
Registers an instance level command.

This can be used to create instance specific commands e.g. if a user needs to type /cmdSOMEAPIKEY:

` self.register_commmand('cmd{}'.format(SOMEAPIKEY), 'name_of_coro') `

serialize(*data*)
Serialises data to be specific for each beard instance.

Serialize callback data (such as with inline keyboard buttons). The id of the plug-in is encoded into the callback data so ownership of callbacks can be easily checked when it is deserialized. Also avoids the same plug-in receiving callback data from another chat

classmethod setup_beards(*key*)
Perform setup necessary for all beards.

class beards.**Command**(*pred*, *coro*, *hlp=None*)
Bases: object

Holds information to determine whether a function should be triggered.

class beards.**Filters**
Bases: object

Filters used to call plugin methods when particular types of messages are received.

For usage, see description of the BeardChatHandler.__commands__ variable.

classmethod document(*chat_handler*, *msg*)
Filters for sent documents

classmethod location(*chat_handler*, *msg*)
Filters for sent locations

classmethod **`text`**(*chat_handler*, *msg*)
Filters for text messages

class `beards.`**`SlashCommand`**(*cmd*, *coro*, *hlp=None*)
Bases: `object`

Holds information to determine whether a telegram command was sent.

class `beards.`**`TelegramHandler`**(*bot*, *parse_mode=None*)
Bases: `logging.Handler`

A logging handler that posts directly to telegram

`emit`(*record*)

exception `beards.`**`ThatsNotMineException`**
Bases: `Exception`

Raised if data does not match beard.

Used to check if serialized callback data belongs to the plugin. See BeardChatHandler.serialize()

`beards.`**`command_predicate`**(*cmd*)
Returns a predicate coroutine which returns True if command is sent.

`beards.`**`create_command`**(*cmd_or_pred*, *coro*, *hlp=None*)
Creates a Command or SlashCommand object as appropriate.

Used to make __commands__ tuples into Command objects.

`beards.`**`regex_predicate`**(*pattern*)
Returns a predicate function which returns True if pattern is matched.

3.2 decorators Module

3.2.1 `decorators` Module

`decorators.`**`debugonly`**(*f_or_text=None*, ***kwargs*)
A decorator to prevent commands being run outside of debug mode.

If the function is awaited when skybeard is not in debug mode, it sends a message to the user. If skybeard is run in debug mode, then it executes the body of the function.

If passed a string as the first argument, it sends that message instead of the default message when not in debug mode.

e.g.

```
@debugonly("Skybeard is not in debug mode.")
async def foo(self, msg):
    # This message will only be sent if skybeard is run in debug mode
    await self.sender.sendMessage("You are in debug mode!")
```

`decorators.`**`onerror`**(*f_or_text=None*, ***kwargs*)
A decorator for sending a message to the user on an exception.

If no arguments are used (i.e. the function is passed directly to the decorator), beard.__onerror__(exception) is called if the decorated function excepts.

If a string is passed as the first argument, then the decorated function sends this message instead of calling the beard.__onerror__ function. kwargs are passed to beard.sender.sendMessage and beard.__onerror__(exception) is called.

If only kwargs are passed, then the decorated function attempts beard.sender.sendMessage(**kwargs) and then calls beard.__onerror__(exception).

3.3 utils Module

3.3.1 `utils` Module

`utils.`**`all_possible_beards`**(*paths*)
List generator of all plug-ins that Skybeard has found and can be loaded

`utils.`**`embolden`**(*string*)
wraps a string in bold tags

`utils.`**`get_args`**(*msg_or_text*, *return_string=False*, ***kwargs*)
Helper function when the command used in the telegram chat may have arguments, e.g /command arg1 arg2. Returns a list of any arguments found after the command

`utils.`**`get_literal_beard_paths`**(*beard_paths*)
Returns list of literal beard paths.

`utils.`**`get_literal_path`**(*path_or_autoloader*)
Gets literal path from AutoLoader or returns input.

`utils.`**`is_module`**(*path*)
Checks if path is a module.

`utils.`**`italisize`**(*string*)
wraps a string in italic tags

`utils.`**`partition_text`**(*text*)
Generator for splitting long texts into ones below the character limit. Messages are split at the nearest line break and each successive chunk is yielded. Relatively untested

3.4 help Package

3.4.1 `help` Package

3.5 autoloaders Package

3.5.1 `autoloaders` Package

class `autoloaders.__init__.`**`AutoLoader`**
Bases: `object`

Base class for automatic loaders (e.g. Git)

class `autoloaders.__init__.`**`Git`**(*url*, *import_as=None*, *branch=None*)
Bases: `autoloaders.__init__.AutoLoader`

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index