vexbot Documentation

Release 0.4.0

Ben Hoff

Dec 30, 2017

Contents:

CHAPTER 1

Extension Development

1.1 Foreword

Currently extensions are tied very closely to python's packaging ecosystem. This is unlikely to change, honestly.

1.2 Setup a Packging Environment

```
$ # Note that the directory these commands are run from is `~/my_vexbot_extensions`!

$ mkdir my_vexbot_extenstions
$ cd my_vexbot_extensions
```

Create a *setup.py* file.

```
# filepath: my_vexbot_extensions/setup.py
from setuptools import setup

setup(name='my_vexbot_extensions')
```

Now take a breather. Python packaging is hard, and we're almost done!

1.3 Extensions Development

Done with your breather? Good. Now let's say we've got a *hello_world* function that we want to use such as the below.

```
# filepath: my_vexbot_extensions/hello_world.py

def hello(*args, **kwargs):
    print('Hello World!')
```

Dope. Let's add this as an extension in our *setup.py* file.

```
# filepath: my_vexbot_extensions/setup.py
from setuptools import setup

setup(name='my_vexbot_extensions',
      entry_points={'vexbot_extensions':['hello=hello_world:hello']})
```

That's it. Let's make sure the package is installed on our path. Make sure you're in the directory *my_vexbot_extensions* (or whatever you named your folder).

```
$ # Note that the directory these commands are run from is `~/my_vexbot_extensions`!

$ ls
setup.py    hello_world.py

$ python setup.py develop
```

You might say, wait, that's not a normal python packaging structure! And you'd be right, so let's look at how it'll change for that. We'll make a directory called *my_src* and create a file in there known as *goodbye_world.py*

```
# filepath: my_vexbot_extensions/my_src/goodbye_world.py

def angsty(*args, **kwargs):
    print('Goodbye, cruel world!')
```

Note how I'm taking an arbitrary amount of arguments and key word arguments using *args and **kwargs? You should do that for every extension, or your program will error out at somepoint when it gets called with metadata that it's not ready for.

```
# filepath: my_vexbot_extensions/my_src/goodbye_world.py

def angsty(*args, **kwargs):
    print('Goodbye, cruel world!')

# Note: Do NOT make extensions without using `*args, **kwargs`
def function_that_will_fail_on_metadata():
    print('Please notice the lack of flexibility in this function for taking arguments
↪')
    print('This type of extension will inevitabley throw `TypeError` exceptions if␣
↪put in a codebase')
```

But back to registering our extension in our *setup.py* file. Remember that the filepath for this is *my_vexbot_extensions/my_src/goodbye_world.py*.

```
# filepath: my_vexbot_extensions/setup.py
from setuptools import setup

setup(name='my_vexbot_extensions',
      entry_points={'vexbot_extensions':['hello=hello_world:hello',
                                         'angsty=my_src.goodbye_world:angsty']})
```

Notice how we use the . operator to represent the folder directory, and the *:* to specify the method name? That's important.

We can have multiple methods in a file that way.

```
# filepath: my_vexbot_extensions/my_src/goodbye_world.py

def angsty(*args, **kwargs):
    print('Goodbye, cruel world!')

def crocodile(*args, **kwargs):
    print('Goodbye, crocodile!')
```

```
# filepath: my_vexbot_extensions/setup.py
from setuptools import setup

setup(name='my_vexbot_extensions',
      entry_points={'vexbot_extensions':['hello=hello_world:hello',
                                         'angsty=my_src.goodbye_world:angsty',
                                         'crocodile=my_src.goodbye_world:crocodile'])
```

If you have a deeply python nested file, such as one in *my_vexbot_extensions/my_src/how/deep/we/go.py*. . . .. code-block:: python

filepath: my_vexbot_extensions/setup.py from setuptools import setup

setup(name='my_vexbot_extensions',

entry_points={'vexbot_extensions':['hello=hello_world:hello',
'angsty=my_src.goodbye_world:angsty', 'crocodile=my_src.goodbye_world:crocodile', 'nested_func=my_src.how.deep.we.go:waay_to_much'])

Note that each folder is separated by the . operator, and the function name in the above example is *waay_to_much*, which is how deeply I feel that function is nested for a simple example such as this.

Remember, remember the 5th of November. And also to re-run *python setup.py develop* once you've added an entry point/extension to your *setup.py* file.

```
$ # Note that the directory these commands are run from is `~/my_vexbot_extensions`!

$ ls
setup.py    hello_world.py    my_src

$ python setup.py develop
```

The string used before the path decleration, I.e the *nested_func* in the string *nested_func=my_src.how.deep.we.go:waay_to_much* is the name you will use in vexbot itself or the *hello* in *hello=hellow_world:hello*.

Let's add our hello world greeting to our command line interface.

```
$ vexbot

vexbot: !add_extensions hello
vexbot: !hello
Hello World!
```

You can also add the hello world to the robot instance as well.

```
$ vexbot

vexbot: !add_extensions hello --remote
```

Last, but not least, you can specify command name alias's, specify if the command should be hidden, and give a short description for what the command does by using the *command* decorator.

```
# filepath: my_vexbot_extensions/hello_world.py
from vexbot.commands import command


@command(alias=['hello_world', 'world_hello'],
         hidden=True, # default is `False`
         short='Prints `Hello World`!')
def hello(*args, **kwargs):
    print('Hello World!')
```

CHAPTER 2

Extension Management

For the most part it's assumed that you are running in the command line program for this documentation.

```
$ vexbot

vexbot:
```

2.1 See Current Commands

```
vexbot: !commands
```

See bot commands

```
vexbot: !commands --remote
```

2.2 See Extensions In Use

```
vexbot: !extensions
```

From bot:

```
vexbot: !extensions --remote
```

2.3 Remove Extensions In Use

Let's say you've the *get_cache* (shows you your configuration cache) and the *cpu_count* extensions in use and you'd like to remove them.

```
vexbot: !remove_extension get_cache cpu_count
```

Alternatively just removing one:

```
vexbot: !remove_extension get_cache
```

2.4 See Installed Extensions

Installed extensions are on your path, but not neccesairly attached to anything running currently

```
vexbot: !get_installed_extensions
```

This command displays a list of every extension installed with a short doc string on what it does.

For example, the command *power_off* will be displayed as

'power_off: Power off a digital ocean droplet'

You can also use a module name to see all the extensions that are provided by that module. *vexbot.extensions.subprocess* is an installed module for vexbot. To see all the extensions that are provided by that module:

```
vexbot: !get_installed_modules vexbot.extensions.subprocess
```

2.5 See Installed Modules

There are a lot of installed extensions and it's hard to figure out what each one does. You can break them up into modules

```
vexbot: !get_installed_modules
```

This is helpful because the installed modules can be used with the *get_installed_extensions* to narrow down what is shown. For example, the every extensions in the module *vexbot.extensions.digitalocean* can be shown by using the following command:

```
vexbot: !get_installed_modules vexbot.extensions.digitalocean
```

2.6 Add Extensions

```
vexbot: !add_extensions get_code delete_cache
```

To add commands to the robot instance:

```
vexbot: !add_extensions get_code delete_cache --remote
```

CHAPTER 3

Extension Discovery

3.1 See Installed Extensions

Installed extensions are on your path, but not neccesairly attached to anything running currently

```
vexbot: !get_installed_extensions
```

This command displays a list of every extension installed with a short doc string on what it does.

For example, the command *power_off* will be displayed as

'power_off: Power off a digital ocean droplet'

You can also use a module name to see all the extensions that are provided by that module. *vexbot.extensions.subprocess* is an installed module for vexbot. To see all the extensions that are provided by that module:

```
vexbot: !get_installed_modules vexbot.extensions.subprocess
```

3.2 See Installed Modules

There are a lot of installed extensions and it's hard to figure out what each one does. You can break them up into modules

```
vexbot: !get_installed_modules
```

This is helpful because the installed modules can be used with the *get_installed_extensions* to narrow down what is shown. For example, the every extensions in the module *vexbot.extensions.digitalocean* can be shown by using the following command:

```
vexbot: !get_installed_modules vexbot.extensions.digitalocean
```

CHAPTER 4

Adapter Development

4.1 Create A New Adapter

Create a messaging instance and pass in a service name that will uniquely identify it.

```
from vexbot.adapters.messaging import Messaging

messaging = Messaging('unique_service_name', run_control_loop=True)
messaging.run(blocking=False)
# Your messaging code here

# Some sort of loop, using a `while` loop for illustration purposes
while True:
    author = ''
    message = ''
    # optional
    # channel = ''

    messaging.send_chatter(author=author,
                           message=message)

    # NOTE: Alternate implementation
    """
    messaging.send_chatter(author=author,
                           message=message,
                           channel=channel)
    """
```

Dope. But what about something that sends commands to the robot?

```
from vexbot.adapters.messaging import Messaging

messaging = Messaging('unique_service_name', run_control_loop=True)
messaging.run(blocking=False)
```

```
# Your code here. You would probably want this in a loop as well.
command = ''
args = []
kwargs = {}

messaging.send_command(command, *args, **kwargs)
```

You probably want a response back out of that command, huh?

```
from vexbot.observer import Observer
from vexbot.adapters.messaging import Messaging

class MyObserver(Observer):
    def on_next(self, request):
        result = request.kwargs.get('result')
        # your code here

    def on_error(self, *args, **kwargs):
        pass

    def on_completed(*args, **kwargs):
        pass

messaging = Messaging('unique_service_name', run_control_loop=True)
messaging.run(blocking=False)

my_observer = MyObserver()

messaging.command.subscribe(my_observer)
# You can also pass in methods to the `subscribe` method
messaging.command.subscribe(your_custom_method_here)
```

Actually you probably want the ability to dynamically load commands, persist your dynamic commands, and see all the installed commands available.

```
import shelve
from os import path
from vexbot.observer import Observer
from vexbot.extensions import extensions

from vexbot.util.get_cache_filepath import get_cache
from vexbot.util.get_cache_filepath import get_cache_filepath as get_cache_dir

class MyObserver(Observer):
    extensions = (extensions.add_extensions,
                  extensions.remove_extension,
                  # NOTE: you can pass in dict's here to change the behavior
                  {'method': your_method_here,
                   'hidden': True,
                   'name': 'some_alternate_method_name',
                   'alias': ['method_name2',
                             'method_name3']},

                  extensions.get_extensions,
                  extensions.get_installed_extensions)

    def __init__(self):
        super().__init__()
```

```
        self._commands = {}
        cache_dir = get_cache_dir()
        mkdir = not path.isdir(cache_dir)
        if mkdir:
            os.makedirs(cache_dir, exist_ok=True)

        filepath = get_cache(__name__ + '.pickle')
        init = not path.isfile(filepath)

        self._config = shelve.open(filepath, flag='c', writeback=True)

        if init:
            self._config['extensions'] = {}
            self._config['disabled'] = {}
            self._config['modules'] = {}

    # NOTE: Here's our command handeling
    def handle_command(self, command: str, *args, **kwargs):
        callback = self._commands.get(command)
        if callback is None:
            return

        # Wrap our callback to catch errors
        try:
             result = callback(*args, **kwargs)
        except Exception as e:
             self.on_error(command, e, args, kwargs)

        print(result)

    def on_next(self, request):
        # NOTE: Here's our responses back from the bot
        result = request.kwargs.get('result')
        # your code here

    def on_error(self, *args, **kwargs):
        pass

    def on_completed(*args, **kwargs):
        pass

>> observer = MyObserver()
>> observer.handle_command('get_extensions')
>> []
>> observer.handle_command('add_extensions', 'log_level')
>> observer.handle_command('get_extensions')
>> ['log_level']
```

CHAPTER 5

Adapter Configuration

There are two things that need to be configured for most adapters. The *.service* file which systemd uses to launch the service, and an option configuration file, which can be used to pass in configurations that need to be persisted.

See this link for example configurations for the packaged adapters. And the below for a primer on ZMQ addresses, if you desire to change the configuration of anything from running locally on the loopback address.

Please note that the *.service* files should be put in *~/.config/systemd/user/**. The *.ini* files may be placed almost anywhere, as long as they are referred to properly in the *.service* file, but recommend they be placed in *!/.config/vexbot/** for consistency.

5.1 Configuring ZMQ Addresses

Addresses can be configured for the adapters and the bot itself in the .ini files. This is a bit more advanced and probably not recommended.

The address expected is in the format of *tcp://[ADDRESS]:[PORT_NUMBER]*.

For example *tcp://127.0.0.1:5617* is a valid address. 127.0.0.1 is the ADDRESS and 5617 is the PORT_NUMBER.

127.0.0.1 was chosen specifially as an example because for IPV4 it is the "localhost". Localhost is the computer the program is being run on. So if you want the program to connect to a socket on your local computer (you probably do), use 127.0.0.1.

Port numbers range from 0-65536, and can be mostly aribratry chosen. For linux ports 0-1024 are reserved, so best to stay away from those. Port 5555 is usually used as an example port for coding examples, so probably best to stay away from that as well.

The value of the *publish_address* and *subscribe_address* at the top of the settings file are likely what you want to copy for the *publish_address* and *subscribe_address* under shell, irc, xmpp, youtube, and socket_io if you're running everything locally on one computer. But you don't have to. You could run all the services on one computer and the main robot on a different computer. You would just need to configure the address and ports correctly, as well as work through any networking/port issues going across the local area network (LAN).

CHAPTER 6

Packages

+====================+==========+ | required packages | License | +====================+==========+ | vexmessage | GPL3 | +————————————-+————+ | pyzmq | BSD | +————————————-+————+ | rx | Apache | +————————————-+————+ | tblib | BSD | +————————————-+————+ | tornado | Apache | +————————————-+————+

6.1 Optional Packages

+====================+==========+ | nlp | License | +====================+==========+ | spacy | | +————————————+————+ | sklearn | | +————————————+————+ | sklearn_crfsuite | | +————————————+————+ | wheel | | +————————————+————+

+====================+==========+ | socket_io | License | +====================+==========+ | requests | | +————————————+————+ | websocket-client | | +————————————+————+

+================+==========+ | summarization | License | +================+==========+ | gensim | | +————————————+————+ | newspaper3k | | +————————————+————+

+============================+==========+ | youtube | License | +============================+==========+ | google-api-python-client | | +————————————————+————+

+========+==========+ | dev | License | +========+==========+ | flake8 | | +————+————+ | twine | | +————+————+ | wheel | | +————+————+

+============+==========+ | xmpp | License | +============+==========+ | sleekxmpp | | +————————+————+ | dnspython | | +————————+————+

+==============+==========+ | process_name | License | +==============+==========+ | setproctitle | | +————————————+————+

+==============+==========+ | speechtotext | License | +==============+==========+ | speechtotext | | +————————————+————+

+==================+==========+ | process_manager | License | +==================+==========+ | pydus | | +————————————+————+

+=================+=========+ | gui | License | +=================+=========+ | chatimusmaximus | | +————————+———+

+======+=========+ | irc | License | +======+=========+ | irc3 | | +——+———+

+============+=========+ | microphone | License | +============+=========+ | microphone | | +————+———+

+==============+=========+ | speechtotext | License | +==============+=========+ | speechtotext | | +————————+———+

CHAPTER 7

Installation

You will need an active DBus user session bus. Depending on your distro, you might already have one (Arch linux, for example).

For Ubuntu:

```
$ apt-get install dbus-user-session python3-gi python3-dev python3-pip build-essential
```

For everyone:

```
$ python3 -m venv <DIR>
```

```
$ source <DIR>/bin/activate
```

```
$ ln -s /usr/lib/python3/dist-packages/gi <DIR>/lib/python3.5/site-packages/
```

```
$ pip install vexbot[process_manager]
```

Make sure your virtual environment is activated. Then run:

```
$ vexbot_generate_certificates
```

```
$ vexbot_generate_unit_file
```

```
$ systemctl --user daemon-reload
```

Your bot is ready to run!

CHAPTER 8

Running

```
$ systemctl --user start vexbot
```

Or

```
$ vexbot_robot
```

Please note that vexbot has a client/server architecture. The above commands will launch the server. To launch the command line client:

```
$ vexbot
```

Exit the command line client by typing *!exit* or using *ctl+D*.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search