pyHoloDeck Documentation

Release 0.0.1

Paul Brian

March 20, 2016

Contents

Holodeck is an attempt to write an immutable server build chain for Python packages.

At the moment I have a means to build .deb packages from a python package (say hosted on github, with a working *setup.py* file)

The build can occur on a spun up cloud server thus meaning the build willcorrectly target the final destination OS, no matter what your laptop runs.

Then we can spin up a destination cloud server and using saltstack / ansible (in transition) we can deploy the package, and configure it using holo-config)

I have introduced a Docker build after this, so the final artifact can be either .deb or a docker image.

This .deb file can then be taken to another server, built in the cloud using salt also, and installed. That way we can build our version of a package once, and move it from test to production, confident we are using the same code, same binary on live as we tested.

Holodeck is based on the rant of Python Core Committer Hynek Schlawack.

It also owes a lot to parcel - not necessarily that any of the codebase is the same but for the sheer get on and do it. Sadly, I could not muster the energy to cross the hg/git divide so instead of contributing patches I simply redid. They have better looking docs too.

HoloDeck is an attempt at a pun - the core idea is to wrap up an entire virtualenv and pass it from host to host. In other words we *enclose virtual environments.* I never said it was a good pun.

The idea is to build a new wrapped venv for every commit, and install it onto immutable servers as it progresses through testing.

I hope this will facilitate more Python (web) packages in the *micro-services* style.

CHAPTER 1

Contents

1.1 HoloMaker

Imagine we want to guarantee a consistent deployment of a single Python package. PyHolodeck is designed to make this simple, but no simpler.

1.1.1 API Docs

Create .deb files from python venvs as artifacts for deployment

app_path is where we create the virtual env and it is also the destination for the final target venv. We cannot avoid this - .deb creates it from dirs

todo: convert to run autmaotically within python (not pront cmds) todo: discvover postinst files and add as cmd switches to fpm todo: have some core service that postinst can call (fabric?) todo: chain to build servers

We need to hace fpm natively installed

class `pyholodeck.maker.`**`DeployConfig`**(*filepath*)

Accept json file, simple convervsion to hold it all lots of very big assumptions here !

class `pyholodeck.maker.`**`Deployment`**(*app_name*, *giturl*)

A big wrapper around different stages in making the python package into a .deb

We are building a simple solution 1. We build on local disk, in the expected locations, a venv

representing the state of the venv we want eventually to deploy

2. We wrap that venv, with the python interpreter etc, into a *.deb* file (tarball basically).
3. We define a *saltstack* file that will deploy the .deb file artifact to our infrastructure. This file will define how to create the .ini / .conf files that will be put into well-known locations for the configuration of the package.
4. We define in the package the conf template for reference

Alternatively the artifact can be a Docker image that contains our .deb file

`BASE_PATH` = **'/mikado'**

the root where the final .deb installed code will get put it is also, for ease of building .debs, where we put the code so the .deb making stage can find it

`python_exe` = **None**

the interpreter in this venv

src_path = None
where we will extract the git source to before runing setup

class pyholodeck.maker.**Docker_Salt**

class pyholodeck.maker.**SubCmd**(*cmdlist*, *pythonstmt=None*, *args=None*)
Smoothly act as store of a subprocess cmd

we want to have same command as a list for non-shell and in friendly form.

nb Its a lot easier to .join a list than parse a string

pyholodeck.maker.**gitfetch**(*url*, *parentfolder*)
Given a git url, retrieve to *parentfolder*

CHAPTER 2

Salt Notes

2.1 Install salt-master

I am focusing on rackspace for salt-cloud.

Initially I build a cloud server, and then convert it into a salt-master. You could use your laptop, but thats not a particularly long term solution.

Installing salt onto ubunutu 12.04:

```
sudo apt-get -y install python-software-properties
sudo add-apt-repository -y ppa:saltstack/salt
sudo apt-get update

sudo apt-get -y install salt-master
sudo apt-get -y install salt-minion
sudo apt-get -y install salt-cloud
## delete as applicable
```

We now have a salt-master on a host, lets put salt-cloud up

2.1.1 Basic Directory Layout

There are two directorys to worry about

- /etc/salt - basic config for both cloud, master, minion
- /srv/salt - location of all the files we are going to put on minion. (Its more complex than that but thats the simplest explantion)

2.1.2 configure the cloud

In */etc/salt* we want to create / adjust two files, */etc/salt/cloud.providers* holds credentials and identifiers for our cloud account. */etc/salt/cloud.profiles*

salt-cloud is going through a revamp of it's configuration, and the new stuff is not quite ready for prime time. This works to date.

/etc/salt/cloud.providers

```
my-rackspace-config:
  # Set the location of the salt-master
  #
  minion:
    master: saltmaster.example.com

  # Configure Rackspace using the OpenStack plugin
  #
  identity_url: 'https://identity.api.rackspacecloud.com/v2.0/tokens'
  compute_name: cloudServersOpenStack
  protocol: ipv4

  # Set the compute region:
  #
  compute_region: DFW

  # Configure Rackspace authentication credentials
  #
  user: myname
  tenant: 123456
  apikey: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  provider: openstack
```

/etc/salt/cloud.profiles:

```
mikado_512:
    provider: mikado-rackspace
    size: 512MB Standard Instance
    image: Ubuntu 12.04 LTS (Precise Pangolin)
```

I have linked this minimal profile called mikado_512, to the rackspace account mikado-rackspace, with the sizes and images configured from (tbd).

2.1.3 Bring up our first minion

```
sudo salt-cloud -p mikado_512 minone
```

We are telling salt-cloud to create a minion, using the mikado_512 profile defined above, and the provider details, and call that minion minone.

When it exists we can do lots of fun things with the minion, from salt-master.

```
171  salt '*' test.ping
172  sudo salt '*' test.ping
173  sudo salt 'myinstance' test.ping
174  sudo salt 'myinstance' sys.doc
175  sudo salt 'myinstance' timezone.get_zone
176  sudo salt 'myinstance' cmd.run 'ls -l /tmp'
177  sudo salt 'myinstance' pkg.install emacs
```

This is all very well, but still fairly manual and prescriptive. Lets move on.

2.2 Writing your first salt module

Salt modules are simply python files that are executed by the running minion on the minion-server, after being told to by the salt-master. Some environment and configuration can be passed in, but most of what we need is available by introspecting the minion-host.

2.2.1 Where to put a salt module?

First create *{FILES_ROOT}/_modules/*. *FILES_ROOT* is defined in /etc/salt/master, and defaults to */srv/salt*

Now create a python module in the _modules directory, such as *pbrian.py*

Simplest possible salt module:

```
import salt

def hello()
    return "hello world"
```

And that's it.

2.2.2 Synchronise from the salt-master to the minion(s)

```
salt '*' saltutils.sync_all
     ^
     selects which minions

 $ sudo salt 'myinstance' saltutil.sync_all
 myinstance:
     ----------
     grains:
     modules:
         - modules.pbrian
     outputters:
     renderers:
     returners:
     states:
```

This will synch the _modules directory (and lots else) from master to minion. So modules are either those you have written and deployed into _modules yourself, or are properly incorporated into the main salt repos

2.2.3 Now run your module on the minion

```
$ sudo salt 'myinstance' pbrian.hello
myinstance:
    hello world
```

Hooray!

So lets recap.

We can manually build a salt-master. We can then auto build any number of minions (up to our credit card limit !) Then we can write a python module to do *anything* on the minion, deploy it and get its output returned to us.

2.2.4 Next steps

- Better Python Integration.
- Actually building our build server.

2.3 Improved Salt Modules

We now want to get a little more useful.

2.3.1 Developing locally

Synching and running commands remotely is all very well, but sometimes we need to develop locally to the salt-master. That's fine:

```
We can fiddle with the local minion modules dir found here:

   /var/cache/salt/minion/extmods/modules/pbrian.py

Or we can alter local `\etc\salt\minion` file and add our chosen location to `modules_dir`
```

This is the simplest and fastest means to develop a module, at least until we delve deeper into saltstack. Do remember that a sync will overwrite your changes !!

2.3.2 Making a minion do something useful

Firstly we shall look at *grains*

```
Grains
  Static bits of information that a minion collects about the system when the minion first starts.
```

I can use them from the CLI:

```
sudo salt '*' grains.ls
```

However this is more fun:

```
import salt

def show_grains():
    return __grains__
```

Which gives us:

```
/snip
cpu_model:
    AMD Opteron(tm) Processor 4170 HE
cpuarch:
    x86_64
defaultencoding:
    UTF-8
defaultlanguage:
    en_US
/snip
```

or even:

```
import salt

def show_grains():
    return __grains__['pythonversion']
```

So, let's sync up the current local pbrian.py with our minion.

```
sudo salt 'myinstance' saltutil.sync_all
```

2.3.3 Running salt programmatically

Let's write a simple python script, in our home-dir.

```
import salt.client
client = salt.client.LocalClient()
ret = client.cmd('myinstance', 'show_grains',[])
print ret
```

gives us:

```
{'myinstance': [2, 7, 3, 'final', 0]}
```

A python dict, returned from a remote minion, ready for manipulation here.

2.4 Immutable salt

I intend to use salt as an automated build tool, for which I wish to use the concept of immutable servers.

Mostly its pretty simple - have one automated build system to build a (virtual) server from scratch, and make sure that server is *exactly the same* each time. Same OS, same package installs, same config files.

Change something? Thats a new version - a *different* immutable server.

Need to upgrade nginx, then your SaaS app needs to get pulled off v.1.2.3 servers and onto v.1.2.4

You do *not* upgrade nginx in-situ.

Thats it really.

2.5 Salt and State

2.5.1 Recap

So far we have covered the basics of how salt works (It puts a service called a *minion* onto a host, and that minion calls back to a 0MQ server for instructions from a salt-master)

We can build minions manually (boo-hiss) or we can use salt-cloud to build VMs in our provider of choice.

Then we can write simple modules that just do-stuff on the minions. There are a lot of these.

While most of the time this is used for server config, in fact this is an ad-hoc remote execution setup. And it knocks *fabric* into a cocked-hat.

2.5.2 One minor issue

OK, OK. Security. Its a biggie. The salt team has written it's own security setup. It has been reviewed. But salt is growing at such a pace, and the sheer difficulty of doing this right indicates that salt *could* face a big, stonking hole in the future.

Its worth bearing in mind, especially as *everything runs as root* (!).

However security is a trade off, and salt brings a lot to the party and looks to be making simple security choices. I am unable to compare it to chef or puppet, however my previous choice of fabric relied on ssh - which is a real battle-tested comms system.

In the end, if you have a bunch of servers automated, I suspect that rogue injection of commands into zeroMQ is less likely than attacking salt-master directly.

So I will stick with it.

2.5.3 Managing State

OK. I could issue one command after another, *expect*-style to create my remote servers.

But that would be the *old-way*.

So now we manage state with config files, and let the minion work out how to get there.

I would recommend reading this now, or very soon http://docs.saltstack.com/ref/states/

2.5.4 top.sls

The top file determines which state files are going to be synched with which minions.

/srv/salt/top.sls:

```
base:
  '*':
    - nginx
```

Now, that means every machine we have will get nginx installed on it (maybe not great) Next we need to define the nginx *state* that we want.

/srv/salt/nginx/init.sls file:

```
$ ls /srv/salt/nginx/
init.sls  nginx.conf
```

/srv/salt/nginx/init.sls:

```
nginx:
  pkg:
    - installed
  service:
    - running
    - enable: True
    - require:
      - pkg: nginx
    - watch:
      - file: /etc/nginx/nginx.conf

/etc/nginx/nginx.conf:
```

```
  file.managed:
    - source: salt://nginx/nginx.conf
```

/srv/salt/nginx/nginx.conf:

```
user www-data;
worker_processes 4;
pid /var/run/nginx.pid;

events {
       worker_connections
...
```

Now we install it:

```
<salt-master>$ sudo salt 'myinstance' state.highstate
```

And after a while we can visit the host in a browser:

p

B

D

G

P

S