python-deploy-framework Documentation

Release 0.1.12

Jonathan Slenders

Mar 26, 2017

Contents

A Python framework for automatic deployment and remote execution on Posix systems.

- genindex
- modindex
- search

Important key features are:

- Interactive execution of remote commands.
- Fast parallel execution.
- Reusability of all deployment code.

CHAPTER 1

Table of content

Getting started

Install the framework as follows:

```
pip install deployer
```

Hello world

Creating nodes

As a quick example, we create a simple node, which does nothing, except printing 'hello world', by executing an *echo* command.

```
from deployer.node import SimpleNode

class SayHello(SimpleNode):
    def hello(self):
        self.host.run('echo hello world')
```

When *SayHello.hello* is called in the example above, it will run the echo command on all the hosts that are known in this Node.

Now we need to define on which hosts this node should run. Let's use Python class inheritance for this.

```
from deployer.host import LocalHost

class SayHelloOnLocalHost(SayHello):
    class Hosts:
        host = LocalHost
```

Starting an interactive shell

Add the following to your Python file, and save it as `deployment.py`.

```
if __name__ == '__main__':
    from deployer.client import start
    start(SayHelloOnLocalHost)
```

If you call it like below, you get a nice interactive shell with tab-completion from where you can run the `hello` command.

```
python deployment.py run
```

Remote SSH Hosts

Instead of using `LocalHost`, you can also run the code on an SSH host.

```
from deployer.host import SSHHost

class MyRemoteHost(SSHHost):
    slug = 'my-host'
    address = '192.168.0.200'
    username = 'john'
    password = '...'

class RemoteHello(SayHello):
    class Hosts:
        host = MyRemoteHost
```

If is even possible to put several instances of the `SayHello` node in your deployment tree, for instance, where one instance is local and the other is remote.

The Console object

The `console` object is an interface for user interaction from within a `Node`. Among the input methods are choice lists, plain text input and password input.

It has output methods that take the terminal size into account, like pagination and multi-column display. It takes care of the pseudo terminal underneat.

Example:

```
class MyNode(Node):
    def do_something(self):
        if self.console.confirm('Should we really do this?', default=True):
            # Do it...
            pass
```

Note: When the script runs in a shell that was started with the `--non-interactive` option, the default options will always be chosen automatically.

---

class `deployer.console.`**`Console`**(*pty*)

Interface for user interaction from within a `Node`.

choice(*question*, *options*, *allow_random=False*, *default=None*)

Parameters

- **options** (*list*) – List of (name, value) tuples.
- **allow_random** (*bool*) – If `True`, the default option becomes 'choose random'.

confirm(*question*, *default=None*)
Print this yes/no question, and return `True` when the user answers 'Yes'.

in_columns(*item_iterator*, *margin_left=0*)

Parameters **item_iterator** – An iterable, which yields either `basestring` instances, or (colored_item, length) tuples.

input(*label*, *is_password=False*, *answers=None*, *default=None*)
Ask for plain text input. (Similar to raw_input.)

Parameters

- **is_password** (*bool*) – Show stars instead of the actual user input.
- **answers** – A list of the accepted answers or None.
- **default** – Default answer.

lesspipe(*line_iterator*)
Paginator for output. This will print one page at a time. When the user presses a key, the next page is printed. `Ctrl-c` or `q` will quit the paginator.

Parameters **line_iterator** – A generator function that yields lines (without trailing newline)

select_node(*root_node*, *prompt='Select a node'*, *filter=None*)
Show autocompletion for node selection.

select_node_isolation(*node*)
Ask for a host, from a list of hosts.

deployer.console.**warning**(*text*)
Print a warning.

Exceptions

exception deployer.exceptions.**ActionException**(*inner_exception*, *traceback*)
When an action fails.

exception deployer.exceptions.**DeployerException**
Base exception class.

exception deployer.exceptions.**ExecCommandFailed**(*command*, *host*, *use_sudo*, *status_code*, *result=None*)
Execution of a run() or sudo() call on a host failed.

exception deployer.exceptions.**QueryException**(*node*, *attr_name*, *query*, *inner_exception*)
Resolving of a Q object in a deployer Node failed.

The interactive shell

Creating an interactive shell from a node tree.

```
if __name__ == '__main__':
    from deployer.client import start
    start(MyRootNode)
```

Navigation

Navigation: TODO

Special commands

Special commands: –inspect, –query, ...

Implementing a custom shell

TODO

Groups

A `Group` can be attached to every Node, in order to put them in categories.

Typically, you have group names like `alpha`, `beta` and `production`. The interactive shell will show the nodes in other colours, depending on the group they're in.

For instance.

```
from deployer.groups import production, staging

class N(Node):
    @production
    class Child(Node):
        pass
```

class `deployer.groups.`**Group**
Group to which a node belongs.

`deployer.groups.`**set_group**(*group*)
Set the group for this node.

```
@set_group(Staging)
class MyNode(Node):
    pass
```

host_container

Access to hosts from within a `Node` class happens through a `HostsContainer` proxy. This container object has also methods for reducing the amount of hosts on which commands are executed, by filtering according to conditions.

The `hosts` property of a node instance returns such a `HostsContainer` object.

```
class MyNode(Node):
    class Hosts:
        web_servers = [Host1, Host2]
        caching_servers = Host3

    def do_something(self):
        # self.hosts here, is a HostsContainer instance.
        self.hosts.filter('caching_servers').run('echo hello')
```

Reference

class deployer.host_container.**HostContainer**(*hosts*, *pty=None*, *logger=None*, *is_sandbox=False*, *host_contexts=None*)

Similar to hostsContainer, but wraps only around exactly one host.

exists(**a*, ***kw*)

Returns `True` when this file exists on the hosts.

get(**args*, ***kwargs*)

Download this remote_file.

has_command(**a*, ***kw*)

Test whether this command can be found in the bash shell, by executing a 'which'

open(**args*, ***kwargs*)

Open file handler to remote file. Can be used both as:

```
with host.open('/path/to/somefile', wb') as f:
    f.write('some content')
```

or:

```
host.open('/path/to/somefile', wb').write('some content')
```

put(**args*, ***kwargs*)

Upload this local_file to the remote location.

run(**a*, ***kw*)

Execute this shell command on the host.

Parameters

- **pty** (*deployer.pseudo_terminal.Pty*) – The pseudo terminal wrapper which handles the stdin/stdout.
- **command** (*basestring*) – The shell command.
- **use_sudo** (*bool*) – Run as superuser.
- **sandbox** (*bool*) – Validate syntax instead of really executing. (Wrap the command in `bash -n`.)
- **interactive** (*bool*) – Start an interactive event loop which allows interaction with the remote command. Otherwise, just return the output.
- **logger** (*LoggerInterface*) – The logger interface.
- **initial_input** (*basestring*) – When `interactive`, send this input first to the host.

- **context** (:class:‘deployer.host.HostContext;) –

sudo(**a*, ***kw*)
Run this command using sudo.

class deployer.host_container.**HostsContainer**(*hosts*, *pty=None*, *logger=None*, *is_sandbox=False*, *host_contexts=None*)
Facade to the host instances. if you have a role, name ‘www’ inside the service webserver, you can do:

•webserver.hosts.run(...)

•webserver.hosts.www.run(...)

•webserver.hosts[0].run(...)

•webserver.hosts.www[0].run(...)

•webserver.hosts.filter(‘www’)[0].run(...)

The host container also keeps track of HostStatus. So, if we fork a new thread, and the HostStatus object gets modified in either thread. Clone this HostsContainer first.

cd(**a*, ***kw*)
Execute commands in this directory. Nesting of cd-statements is allowed.

```
with host.cd('~/directory'):
    host.run('ls')
```

env(**a*, ***kw*)
Set this environment variable

exists(*filename*, *use_sudo=True*)
Returns `True` when this file exists on the hosts.

filter(**roles*)
Examples:

```
hosts.filter('role1', 'role2')
hosts.filter('*') # Returns everything
hosts.filter( ['role1', 'role2' ]) # TODO: deprecate
host.filter('role1', MyHostClass) # This means: take 'role1' from this␣
↪container, but add an instance of this class
```

classmethod from_definition(*hosts_class*, ***kw*)
Create a `HostContainer` from a Hosts class.

get(**roles*)
Similar to `filter()`, but returns exactly one host instead of a list.

has_command(*command*, *use_sudo=False*)
Test whether this command can be found in the bash shell, by executing a ‘which’

prefix(**a*, ***kw*)
Prefix all commands with given command plus `&&`.

```
with host.prefix('workon environment'):
    host.run('./manage.py migrate')
```

run(**a*, ***kw*)
Execute this shell command on the host.

Parameters

- **pty** (*deployer.pseudo_terminal.Pty*) – The pseudo terminal wrapper which handles the stdin/stdout.
- **command** (*basestring*) – The shell command.
- **use_sudo** (*bool*) – Run as superuser.
- **sandbox** (*bool*) – Validate syntax instead of really executing. (Wrap the command in `bash -n`.)
- **interactive** (*bool*) – Start an interactive event loop which allows interaction with the remote command. Otherwise, just return the output.
- **logger** (*LoggerInterface*) – The logger interface.
- **initial_input** (*basestring*) – When `interactive`, send this input first to the host.
- **context** (:class:‘deployer.host.HostContext;) –

sudo (**args*, ***kwargs*)
Run this command using sudo.

Inspection

The inspection module contains a set of utilities for introspection of the deployment tree. This can be either from inside an action, or externally to reflect on a given tree.

Suppose that we already have the following node instantiated:

```
from deployer.node import Node

class Setup(Node):
    def say_hello(self):
        self.hosts.run('echo "Hello world"')

setup = Setup()
```

Now we can ask for the list of actions that this node has:

```
from deployer.inspection import Inspector

insp = Inspector(setup)
print insp.get_actions()
print insp.get_childnodes()
```

class deployer.inspection.inspector.**Inspector**(*node*)
Introspection of a `Node` instance.

get_action(*name*)
Return the `Action` with this name or raise `AttributeError`.

get_actions(*include_private=True*)
Return a list of `Action` instances for the actions in this node.

Parameters **include_private** (*bool*) – Include actions starting with an underscore.

get_childnode(*name*)
Return the childnode with this name or raise `AttributeError`.

get_childnodes(*include_private=True*, *verify_parent=True*)
Return a list of childnodes.

Parameters

- **include_private** (*bool*) – ignore names starting with underscore.
- **verify_parent** (*bool*) – check that the parent matches the current node.

get_group()
Return the *deployer.groups.Group* to which this node belongs.

get_name()
Return the name of this node.

Note: when a node is nested in a parent node, the name becomes the attribute name of this node in the parent.

get_parent()
Return the parent `Node` or raise `AttributeError`.

get_path(*path_type='NAME_ONLY'*)
Return a (name1, name2, ...) tuple, defining the path from the root until here.

Parameters **path_type** (*PathType*) – Path formatting.

get_properties(*include_private=True*)
Return the attributes that are properties.

This are the members of this node that were wrapped in `@property` :returns: A list of `Action` instances.

get_property(*name*)
Returns the property with this name or raise AttributeError. :returns: `Action` instance.

get_queries(*include_private=True*)
Return the attributes that are `deployer.query.Query` instances.

get_query(*name*)
Returns the Action object that wraps the Query with this name or raise AttributeError.

Returns An `Action` instance.

get_root()
Return the root `Node` of the tree.

has_action(*name*)
Returns `True` when this node has an action called `name`.

has_childnode(*name*)
Returns `True` when this node has a childnode called `name`.

has_property(*name*)
Returns `True` when the attribute `name` is a @property.

has_query(*name*)
Returns `True` when the attribute `name` of this node is a Query.

is_callable()
Return `True` when this node implements `__call__`.

suppress_result_for_action(*name*)
`True` when *deployer.node.suppress_action_result()* has been applied to this action.

walk(*filter=None*)
Recursively walk (topdown) through the nodes and yield them.

It does not split `SimpleNodes` nodes in several isolations.

Parameters **filter** – A *filters.Filter* instance.

Returns A *NodeIterator* instance.

class deployer.inspection.inspector.**NodeIterator**(*node_iterator_func*)
Generator object which yields the nodes in a collection.

call_action(*name*, **a*, ***kw*)
Call a certain action on all the nodes.

filter(*filter*)
Apply filter on this node iterator, and return a new iterator instead. *filter* should be a Filter instance.

prefer_isolation(*index*)
For nodes that are not yet isoleted. (SimpleNodes, or normal Nodes nested in there.) yield the isolations with this index. Otherwise, nodes are yielded unmodified.

class deployer.inspection.inspector.**PathType**
Types for displaying the `Node` address in a tree. It's an options for Inspector.get_path()

NAME_ONLY = 'NAME_ONLY'
A list of names.

NODE_AND_NAME = 'NODE_AND_NAME'
A list of `(Node, name)` tuples.

NODE_ONLY = 'NODE_ONLY'
A list of nodes.

Filters for NodeIterator

`NodeIterator` is the iterator that `Inspector.walk()` returns. It supports filtering to limit the yielded nodes according to certain conditions.

A filter is a `Filter` instance or an AND or OR operation of several filters. For instance:

```
from deployer.inspection.filters import HasAction, PublicOnly
Inspector(node).walk(HasAction('my_action') & PublicOnly & ~ InGroup(Staging))
```

class deployer.inspection.filters.**Filter**
Base class for `Inspector.walk` filters.

deployer.inspection.filters.**PublicOnly = PublicOnly**
Filter on public nodes.

deployer.inspection.filters.**PrivateOnly = PrivateOnly**
Filter on private nodes.

class deployer.inspection.filters.**IsInstance**(*node_class*)
Filter on the nodes which are an instance of this `Node` class.

Parameters **node_class** – A *deployer.node.Node* subclass.

class deployer.inspection.filters.**HasAction**(*action_name*)
Filter on the nodes which implement this action.

class deployer.inspection.filters.**InGroup**(*group*)
Filter nodes that are in this group.

Parameters **group** – A *deployer.groups.Group* subclass.

The node object

TODO: examples and documentation.

```
from deployer.node import SimpleNode

class SayHello(SimpleNode):
    def hello(self):
        self.host.run('echo hello world')
```

Note: It is interesting to know that `self` is actually not a `Node` instance, but an `Env` object which will proxy this actual Node class. This is because there is some metaclass magic going on, which takes care of sandboxing, logging and some other nice stuff, that you get for free.

Except that a few other variables like `self.console` are available, you normally won't notice anything.

Running the code

```
from deployer.node import Env

env = Env(MyNode())
env.hello()
```

Inheritance

A node is meant to be reusable. It is encouraged to inherit from such a node class and overwrite properties or class members.

Expansion of double underscores

TODO: ...

The difference between Node and SimpleNode

TODO: ...

.Array and .JustOne

TODO: ...

Using contrib.nodes

The deployer framework is delivered with a *contrib.nodes* directory which contains nodes that should be generic enough to be usable by a lot of people. Even if you can't use them in your case, they may be good examples of how to do certain things. So don't be afraid to look at the source code, you can learn some good practices there. Take these and inherit as you want to, or start from scratch if you prefer that way.

Some recommended contrib nodes:

- *deployer.contrib.nodes.config.Config*

  This a the base class that we are using for every configuration file. It is very useful for when you are automatically generating server configurations according to specific deployment configurations. Without any efford, this class will allow you to do diff's between your new, generated config, and the config that's currently on the server side.

Reference

class deployer.node.**Env**(*node*, *pty=None*, *logger=None*, *is_sandbox=False*)
Wraps a Node into a context where actions can be executed.

Instead of `self`, the first parameter of a `Node`-action will be this instance. It acts like a proxy to the `Node`, but in the meantime it takes care of logging, sandboxing, the terminal and context.

console
Interface for user input. Returns a *deployer.console.Console* instance.

hosts
deployer.host_container.HostsContainer instance. This is the proxy to the actual hosts.

initialize_node(*node_class*)
Dynamically initialize a node from within another node. This will make sure that the node class is initialized with the correct logger, sandbox and pty settings. e.g:

Parameters **node_class** – A `Node` subclass.

```
class SomeNode(Node):
    def action(self):
        pass

class RootNode(Node):
    def action(self):
        # Wrap SomeNode into an Env object
        node = self.initialize_node(SomeNode)

        # Use the node.
        node.action2()
```

class deployer.node.**Node**(*parent=None*)
This is the base class for any deployment node.

class deployer.node.**SimpleNode**(*parent=None*)
A `SimpleNode` is a `Node` which has only one role, named `host`. Multiple hosts can be given for this role, but all of them will be isolated, during execution. This allows parallel executing of functions on each 'cell'.

deployer.node.**suppress_action_result**(*action*)
When using a deployment shell, don't print the returned result to stdout. For example, when the result is superfluous to be printed, because the action itself contains already print statements, while the result can be useful for the caller.

deployer.node.**dont_isolate_yet**(*func*)
If the node has not yet been separated in serveral parallel, isolated nodes per host. Don't do it yet for this function. When anothor action of the same host without this decorator is called, the node will be split.

It's for instance useful for reading input, which is similar for all isolated executions, (like asking which Git Checkout has to be taken), before forking all the threads.

Note that this will not guarantee that a node will not be split into its isolations, it does only say, that it does not have to. It is was already been split before, and this is called from a certain isolation, we'll keep it like that.

deployer.node.**alias**(*name*)
Give this node action an alias. It will also be accessable using that name in the deployment shell. This is useful, when you want to have special characters which are not allowed in Python function names, like dots, in the name of an action.

pseudo_terminal

Note: This module is mainly for internal use.

Pty implements a terminal abstraction. This can be around the default stdin/out pair, but also around a pseudo terminal that was created through the `openpty` system call.

class deployer.pseudo_terminal.**DummyPty**(*input_data=''*)
Pty compatible object which insn't attached to an interactive terminal, but to dummy StringIO instead.

This is mainly for unit testing, normally you want to see the execution in your terminal.

class deployer.pseudo_terminal.**Pty**(*stdin=None*, *stdout=None*, *interactive=True*)
Terminal abstraction around a stdin/stdout pair.

Contains helper function, for opening an additional Pty, if parallel deployments are supported.

Stdin The input stream. (`sys.__stdin__` by default)

Stdout The output stream. (`sys.__stdout__` by default)

Interactive When `False`, we should never ask for input during the deployment. Choose default options when possible.

get_height()
Return the height.

get_size()
Get the size of this pseudo terminal.

Returns A (rows, cols) tuple.

get_width()
Return the width.

run_in_auxiliary_ptys(*callbacks*)
For each callback, open an additional terminal, and call it with the new 'pty' as parameter. The callback can potentially run in another thread.

The default behaviour is not in parallel, but sequential. Socket_server however, inherits this pty, and overrides this function for parrallel execution.

Parameters **callbacks** – A list of callables.

set_size(*rows*, *cols*)
Set terminal size.

(This is also mainly for internal use. Setting the terminal size automatically happens when the window resizes. However, sometimes the process that created a pseudo terminal, and the process that's attached to the output window are not the same, e.g. in case of a telnet connection, or unix domain socket, and then we have to sync the sizes by hand.)

deployer.pseudo_terminal.**select**(**args*, ***kwargs*)
Wrapper around select.select.

When the SIGWINCH signal is handled, other system calls, like select are aborted in Python. This wrapper will retry the system call.

The query object

Queries provide syntactic sugar for expressions inside nodes. For instance:

```
from deployer.query import Q

class MyNode(Node):
    do_something = True

    class MyChildNode(Node):
        do_something = Q.parent.do_something

        def setup(self):
            if self.do_something:
                ...
                pass
```

Technically, such a Query object uses the descriptor protocol. This way, it acts like any python `property`, and is completely transparent.

More examples

A query can address the attribute of an inner node. When the property `attribute_of_a` in the example below is retrieved, the query executes and accesses the inner node `A` in the background.

```
class Root(Node):
    class A(Node):
        attribute = 'value'

    attribute_of_a = Q.A.attribute

    def action(self):
        if self.attribute_of_a == 'value':
            do_something(...)
```

A query can also call a function. The method `get_url` is called in the background.

```
class Root(Node):
    class A(Node):
        def get_url(self, domain):
            return 'http://%s' % domain

    url_of_a = Q.A.get_url('example.com')

    def print_url(self):
        print self.url_of_a
```

Note: Please make sure that a query can execute without side effects. This means, that a query should never execute a command that changes something on a host. Consider it read-only, like the getter of a property.

(This is mainly a convension, but could result in unexpected results otherwise.)

A query can even do complex calculations:

```
from deployer.query import Q

class Root(Node):
    class A(Node):
        length = 4
        width = 5

    # Multiply
    size = Q.A.length * Q.A.width

    # Operator priority
    size_2 = (Q.A.length + 1) * Q.A.width

    # String interpolation
    size_str = Q('The size is %s x %s') % (Q.A.height, Q.A.width)
```

Note: Python does not support overloading the `and`, `or` and `not` operators. You should use the bitwise equivalents `&`, `|` and ~ instead.

Reference

TODO: implemented operators

TODO: implemented special methods: __getitem__,

Utils

String utilities

`deployer.utils.string_utils.`**`esc1`**(*string*)
Escape single quotes, mainly for use in shell commands. Single quotes are usually preferred above double quotes, because they never do shell expension inside. e.g.

```
class HelloWorld(Node):
    def run(self):
        self.hosts.run("echo '%s'" % esc1("Here's some text"))
```

`deployer.utils.string_utils.`**`esc2`**(*string*)
Escape double quotes

`deployer.utils.string_utils.`**`indent`**(*string*, *prefix=' '*)
Indent every line of this string.

Other

`deployer.utils.network.`**`parse_ifconfig_output`**(*output*, *only_active_interfaces=True*)
Parse the output of an *ifconfig* command.

Returns A list of `NetworkInterface` objects.

About

Special thanks to

This framework depends on two major libraries: Paramiko and Twisted Matrix. A small amount of code was also inspired by Fabric.

Authors

- Jonathan Slenders (VikingCo, Mobile Vikings)
- Jan Fabry (VikingCo, Mobile Vikings)

Python Module Index

d

Index