python-deploy-framework Documentation

Release 0.1.12

Jonathan Slenders

Mar 26, 2017

Contents

Framework for remote execution on Posix systems.

- genindex
- modindex
- search

Important key features are:

- Powerful interactive command line with autocompletion;
- Interactive and fast parallel execution;
- Reusability of your code (through inheritance);
- Normally using SSH for remote execution, but pluggable for other execution methods;
- Pluggable logging framework;
- All scripts can be used as a library, easy to call from another Python application. (No global state.)

It's more powerful than Fabric, but different from Saltstack. It's not meant to replace anything, it's another tool for your toolbox.

Questions? Just create a ticket in Github for now:

- Read the tutorials: *Hello world* and *Deploying an application*
- Find the source code at github.

CHAPTER 1

Table of contents

Tutorial: Hello world

In this short tutorial, we'll demonstrate how to create a simple interactive shell around one simple deployment command that just prints 'Hello World'. We suppose you have already an understanding of the Python language and Python packages.

Install requirements

Install the following package.

```
pip install deployer
```

This will probably also install dependencies like `paramiko`, `twisted` and `pexpect`, but don't worry too much about that.

Creating nodes

Now we will create a `deployer.node.Node` to contains the 'Hello world' action. Such a `Node` class is the start for any deployment component. Paste the following in an empty Python file:

```python
from deployer.node import Node

class SayHello(Node):
    def hello(self):
        self.hosts.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 to this Node.

Linking the node to actual hosts

Now we need to define on which hosts this node should run. Let's use Python class inheritance for this. Append the following to your Python file:

```
from deployer.host import LocalHost

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

Starting an interactive shell

One way of execting this code, is by wrapping it in an *interactive shell*. This is the last thing to do: add the following to the bottom of your Python file, and save it as `my_deployment.py`.

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

Call it like below, and you'll get a nice interactive shell with tab-completion from where you can run the `hello` command.

```
python deployment.py run
```

Remote SSH Hosts

So, in the example we have shown how to run 'Hello world' on your local machine. That's fine, but probably we want to execute this on a remote machine that's connected through SSH. That's possible by creating an `SSHHost` class instead of using `LocalHost`. Make sure to change the credentials to your own.

```
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
```

Complete example

As a final example, we show how we created two instances of `SayHello`. One mapped to your local machine, and one mapped to a remote SSH Host. These two nodes are now wrapped in a parent node, that groups both.

```
#!/usr/bin/env python

# Imports
from deployer.client import start
from deployer.host import SSHHost, LocalHost
```

```python
from deployer.node import Node

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

# The deployment nodes

class SayHello(Node):
    def hello(self):
        self.hosts.run('echo hello world')

class RootNode(Node):
    class local_hello(SayHello):
        class Hosts:
            host = LocalHost

    class remote_hello(SayHello):
        class Hosts:
            host = MyRemoteHost

if __name__ == '__main__':
    start(RootNode)
```

Where to go now?

What you learned here is a basic example of how to use the deployment framework. However, there are much more advanced concepts possible. A quick listing of items to learn are the following. (In logical order of learning.)

- *Read the Django tutorial*
- *Architecture of role and nodes*
- *Inheritance (and double underscore expansion)*
- *Query expressions*
- *Introspection*

Tutorial: Deploying a (Django) application

This is a short tutorial that walks you through the steps required to create a script that automatically installs a Django application on a server. We use the Django application only as an example, the tutorial is meant to cover enough that you can apply it yourself for deployments or management of any kind of remote applications.

You learn how to write a deploy or remote execution script that can be (re)used for installation of a new servers, for incremental upgrades or for manually debugging the server.

Some assumptions:

- You should have SSH credentials of the server on which you're going to deploy and you know how to use SSH.
- You should know how to work with a bash shell.

Not required, but useful:

- You have knowledge of Git, and your code is in a Git-repository. (Then we can use `git clone` to get our code on the servers.)
- You have some knowledge of gunicorn, nginx and other tools for running wsgi applications.

Note: It's important that you understand the tools you're going to deploy, and how to cofigure them by hand. In this case, we are configuring gunicorn and Django as an example, so we would have to know how these things work. (You can't write a script to repeat some work for you, if you have no idea how to do it yourself.) The deployer framework has no idea what Django or nginx is, it just executes code on servers.

This tutorial is only an example of how you could automatically deploy a Django application. You can but probably won't do it exactly as described here. The purpose of the tutorial is in the first place to explain some relevant steps, so you have an idea how you could create a repeatable script of the steps that you would otherwise do by hand.

So we are going to write a script that:

- gets your code from the repository to the server (git clone);
- Installs the requirements in a virtualenv;
- sets up a `local_settings.py` configuration file on the server;
- installs and configures Gunicorn.

Using python-deployer

On your local system, you need to install the `deployer` Python library with `pip` or `easy_install`. (If you are not using a virtualenv, you have to use `sudo` to install it system-wide.)

```
pip install deployer
```

Now, you can create a new Python file, save it as `deploy.py` and paste the following template in there.

```
#!/usr/bin/env python
from deployer.client import start
from deployer.node import Node

class DjangoDeployment(Node):
    pass

if __name__ == '__main':
    start(DjangoDeployment)
```

Make it executable:

```
chmod +x deploy.py
```

This does nothing yet. In the following sections, we are going to add more code to the `DjangoDeployment` *Node*. If you run the script, you will already get an *interactive shell*, but there's also nothing much to see yet. Try to run the script as follows:

```
./deploy.py
```

You can quit the shell by typing `exit`.

Writing the deployment script

Git checkout

Lets start by adding code for cloning and checking out a certain revision of the repository. You can add the `install_git`, `git_clone` and `git_checkout` methods in the snippet below to the `DjangoDeployment` node.

```python
from deployer.utils import esc1

class DjangoDeployment(Node):
    project_directory = '~/git/django-project'
    repository = 'git@github.com:example/example.git'

    def install_git(self):
        """ Installs the ``git`` package. """
        self.host.sudo('apt-get install git')

    def git_clone(self):
        """ Clone repository."""
        with self.host.cd(self.project_directory, expand=True):
            self.host.run("git clone '%s'" % esc1(self.repository))

    def git_checkout(self, commit):
        """ Checkout specific commit (after cloning)."""
        with self.host.cd(self.project_directory, expand=True):
            self.host.run("git checkout '%s'" % esc1(commit))
```

Probably obvious, we have a clone and checkout function that are meant to go to a certain directory on the server and run a shell command in there through *run()*. Some points worth noting:

- `expand=True`: this means that we should do tilde-expension. You want the tilde to be replaced with the home directory. If you have an absolute path, this isn't necessary.
- *esc1()*: This is important to avoid shell injection. We receive the commit variable from a parameter, and we don't know what it will look like. The *esc1()* escape function is designed to escape a string for use inside single quotes in a shell script: note the surrounding quotes in `'%s'`.
- We need to use *sudo()* for the installation of Git, because `apt-get` needs to have root rights.

Defining the SSH host

Now we are going to define the SSH host. It is recommended to authenticate through a private key. If you have a `~/.ssh/config` setup in a way that allows you to connect directly through the `ssh` command by only passing the address, then you also can drop all the other settings (except the address) from the *SSHHost* below.

```python
from deployer.host import SSHHost

class remote_host(SSHHost):
    address = '192.168.1.1' # Replace by your IP address
    username = 'user'       # Replace by your own username.
    password = 'password'   # Optional, but required for sudo operations
    key_filename = None     # Optional, specify the location of the RSA
                            #   private key
```

That defines how to access the remote host. If you ever have to define another host, feel free to use Python inheritance if they share some settings.

Now we have to tell `DjangoDeployment` node to use this host. The following syntax may look slightly overkill at first, but this is how we link the `remote_host` to the `DjangoDeployment`.[1] Instead of putting the `Hosts` class inside the original `DjangoDeployment`, you can off course again –like always in Python– inherit the original class and extend that one by nesting `Hosts` in there.

```
class DjangoDeployment(Node):
    class Hosts:
        host = remote_host

    ...
```

Put together, we currently have the following in our script:

```
#!/usr/bin/env python
from deployer.utils import esc1
from deployer.host import SSHHost

class remote_host(SSHHost):
    address = '192.168.1.1' # Replace by your IP address
    username = 'user'       # Replace by your own username.
    password = 'password'   # Optional, but required for sudo operations
    key_filename = None     # Optional, specify the location of the RSA
                            #   private key

class DjangoDeployment(Node):
    class Hosts:
        host = remote_host

    project_directory = '~/git/django-project'
    repository = 'git@github.com:example/example.git'

    def install_git(self):
        """ Installs the ``git`` package. """
        self.host.sudo('apt-get install git')

    def git_clone(self):
        """ Clone repository."""
        with self.host.cd(self.project_directory, expand=True):
            self.host.run("git clone '%s'" % esc1(self.repository))

    def git_checkout(self, commit):
        """ Checkout specific commit (after cloning)."""
        with self.host.cd(self.project_directory, expand=True):
            self.host.run("git checkout '%s'" % esc1(commit))

if __name__ == '__main__':
    start(DjangoDeployment)
```

If you run this executable, you can already execute the methods if this class from the interactive shell.

Configuration management

For most Django projects you also want to have a settings file for the server configuration. Django projects define a Python module through the environment variable DJANGO_SETTINGS_MODULE. Usually, these settings are not entirely the same on a local development machine and the server, you might have another database or caching

[1] The reason is that you can add multiple hosts to a node, and even multiple hosts to multiple 'roles' in a node. This allows for some more complex setups and parallel deployments.

server. Often, you have a `settings.py` in your repository, while each server still gets a `local_settings.py` to override the server specific configurations. (12factor.net has some good guidelines about config management.)

Anyway, suppose that you have a configuration that you want to upload to `~/git/django-project/local_settings.py`. Let's create a method for that:

```
django_settings = \
"""
DATABASES['default'] = ...
SESSION_ENGINE = ...
DEFAULT_FILE_STORAGE = ...
"""

class DjangoDeployment(Node):
    ...
    def upload_django_settings(self):
        """ Upload the content of the variable 'local_settings' in the
        local_settings.py file. """
        with self.host.open('~/git/django-project/local_settings.py') as f:
            f.write(django_settings)
```

So, by calling `open()`, we can write to a remote file on the host, as if it were a local file.

Managing the virtualenv

Virtualenvs can sometimes be very tricky to manage on the server and to use them in automated scripts. You are working inside a virtualenv if your `$PATH` environment is set up to prefer binaries installed at the path of the virtual env rather than use the system default. If you are working inside a interactive shell, you may use a tool like `workon` or something similar to activate the virtualenv. We don't want to rely on the availability of these tools and inclusion of such scripts from a `~/.bashrc`. Instead, we can call the `bin/activate` by hand to set up a correct `$PATH` variable. It is important to prefix all commands that apply to the virtualenv by this activation command.

In this tutorial we will suppose that you already have a virtualenv created by hand, called `'project-env'`. Lets now create a few reusable functions for installing stuff inside the virtualenv.

```
class DjangoDeployment(Node):
    ...
    # Command to execute to work on the virtualenv
    activate_cmd = '. ~/.virtualenvs/project-env/bin/activate'

    def install_requirements(self):
        """
        Script to install the requirements of our Django application.
        (We have a requirements.txt file in our repository.)
        """
        with self.host.prefix(self.activate_cmd):
            self.host.run("pip install -r ~/git/django-project/requirements.txt')

    def install_package(self, name):
        """
        Utility for installing packages through ``pip install`` inside
        the env.
        """
        with self.host.prefix(self.activate_cmd):
            self.host.run("pip install '%s'" % name)
```

Notice the `prefix()` context manager that makes sure that all `run()` commands are executed inside the virtualenv.

Running Django management commands

It’s good and useful have to have a helper function somewhere that can execute Django management commands from the deployment script. You’re going to use it all the time.

Lets add a `run_management_command` which accepts a `command` parameter to be passed as an argument to `./manage.py`. As an example we also add a `django_shell` method which starts in interactive django shell on the server.

```
class DjangoDeployment(Node):
    ...
    def run_management_command(self, command):
        """ Run Django management command in virtualenv. """
        # Activate the virtualenv.
        with self.host.prefix(self.activate_cmd):
            # Go to the directory where we have our 'manage.py' file.
            with self.host.cd('~/git/django-project/'):
                self.host.run('./manage.py %s' % command)

    def django_shell(self):
        """ Open interactive Django shell. """
        self.run_management_command('shell')
```

Running gunicorn through supervisord

You don’t want to use Django’s `runserver` on production, so we’re going to install and configure gunicorn. We are going to use supervisord to mangage the gunicorn process, but depending on your system you meight prefer systemd or upstart instead. We need to install both gunicorn and supervisord in the environment and create configuration files file both.

Let’s first add a few methods for installing the required packages inside the virtualenv.

```
class DjangoDeployment(Node):
    ...

    def install_gunicorn(self):
        """ Install gunicorn inside the virtualenv. """
        self.install_package('gunicorn')

    def install_supervisord(self):
        """ Install supervisord inside the virtualenv. """
        self.install_package('supervisor')
```

For testing purposes, we add a command to run the gunicorn server from the shell.[2]

```
class DjangoDeployment(Node):
    ...

    def run_gunicorn(self):
        """ Run the gunicorn server """
        self.run_management_command('run_gunicorn')
```

Obviously, you don’t want to keep your shell open all the time. So, let’s configure supervisord. The following code will upload the supervisord configuration to `/etc/supervisor/conf.d/django-project.conf`. This is similar to uploading the Django configuration earlier.

[2] See: http://docs.gunicorn.org/en/latest/run.html#django-manage-py

```
supervisor_config = \
"""
[program:djangoproject]
command = /home/username/.virtualenvs/project-env/bin/gunicorn_start  ; Command to␣
↪start app
user = username                                                       ; User to run as
stdout_logfile = /home/username/logs/gunicorn_supervisor.log          ; Where to␣
↪write log messages
redirect_stderr = true                                                ; Save stderr␣
↪in the same log
"""

class DjangoDeployment(Node):
    ...

    def upload_supervisor_config(self):
        """ Upload the content of the variable 'supervisor_config' in the
        supervisord configuration file. """
        with self.host.open('/etc/supervisor/conf.d/django-project.conf') as f:
            f.write(supervisor_config)
```

Gathering again everything we have:

```
#!/usr/bin/env python
from deployer.utils import esc1
from deployer.host import SSHHost

supervisor_config = \
"""
[program:djangoproject]
command = /home/username/.virtualenvs/project-env/bin/gunicorn_start  ; Command to␣
↪start app
user = username                                                       ; User to run as
stdout_logfile = /home/username/logs/gunicorn_supervisor.log          ; Where to␣
↪write log messages
redirect_stderr = true                                                ; Save stderr␣
↪in the same log
"""

django_settings = \
"""
DATABASES['default'] = ...
SESSION_ENGINE = ...
DEFAULT_FILE_STORAGE = ...
"""

class remote_host(SSHHost):
    address = '192.168.1.1' # Replace by your IP address
    username = 'user'       # Replace by your own username.
    password = 'password'   # Optional, but required for sudo operations
    key_filename = None     # Optional, specify the location of the RSA
                            #   private key
class DjangoDeployment(Node):
    class Hosts:
        host = remote_host

    project_directory = '~/git/django-project'
    repository = 'git@github.com:example/example.git'
    def install_git(self):
        """ Installs the ``git`` package. """
        self.host.sudo('apt-get install git')

    def git_clone(self):
        """ Clone repository."""
        with self.host.cd(self.project_directory, expand=True):
            self.host.run("git clone '%s'" % esc1(self.repository))

    def git_checkout(self, commit):
        """ Checkout specific commit (after cloning)."""
        with self.host.cd('~/git/django-project', expand=True):
            self.host.run("git checkout '%s'" % esc1(commit))

    # Command to execute to work on the virtualenv
    activate_cmd = '. ~/.virtualenvs/project-env/bin/activate'

    def install_requirements(self):
        """
        Script to install the requirements of our Django application.
        (We have a requirements.txt file in our repository.)
        """
        with self.host.prefix(self.activate_cmd):
            self.host.run("pip install -r ~/git/django-project/requirements.txt')

    def install_package(self, name):
        """
        Utility for installing packages through ``pip install`` inside
        the env.
        """
        with self.host.prefix(self.activate_cmd):
            self.host.run("pip install '%s'" % name)

    def upload_django_settings(self):
        """ Upload the content of the variable 'local_settings' in the
        local_settings.py file. """
        with self.host.open('~/git/django-project/local_settings.py') as f:
            f.write(django_settings)

    def run_management_command(self, command):
        """ Run Django management command in virtualenv. """
        # Activate the virtualenv.
        with self.host.prefix(self.activate_cmd):
            # Cd to the place where we have our 'manage.py' file.
            with self.host.cd('~/git/django-project/'):
                self.host.run('./manage.py %s' % command)

    def django_shell(self):
        """ Open interactive Django shell. """
        self.run_management_command('shell')

    def install_gunicorn(self):
        """ Install gunicorn inside the virtualenv. """
        self.install_package('gunicorn')

    def install_supervisord(self):
        """ Install supervisord inside the virtualenv. """
        self.install_package('supervisor')

    def run_gunicorn(self):
        """ Run the gunicorn server """
        self.run_management_command('run_gunicorn')

    def upload_supervisor_config(self):
        """ Upload the content of the variable 'supervisor_config' in the
        supervisord configuration file. """
        with self.host.open('/etc/supervisor/conf.d/django-project.conf') as f:
            f.write(supervisor_config)

if __name__ == '__main':
    start(DjangoDeployment)
```

Making stuff reusable

The above deployment script works. But it's not really reusable. You don't want to write a gunicorn configuration for every Django project you're going to set up. And you also don't want to do the same again for a staging environment if you have the scripts for the production, even when there are minor differences. So we are going to move hard coded parts out of our code and make our `DjangoDeployment` reusable.

A reusable virtualenv class.

Let's start by putting all the virtualenv related functions in one class. Most of the script will be the same among projects, except for a few variables:

- The location of the virtualenv
- The packages to be installed there
- The location of a `requirements.txt` file

A reusable `VirtualEnv` class could look like this:

```
class VirtualEnv(Node):
    location = required_property()
    requirements_files = []
    packages = []

    # Command to execute to work on the virtualenv
    @property
    def activate_cmd(self):
        return  '. %s/bin/activate' % self.location

    def install_requirements(self):
        """
        Script to install the requirements of our Django application.
        (We have a requirements.txt file in our repository.)
        """
        with self.host.prefix(self.activate_cmd):
            for f in self.requirements_files:
                self.host.run("pip install -r '%s' " % esc1(f))

    def install_package(self, name):
        """
        Utility for installing packages through ``pip install`` inside
        the env.
        """
        with self.host.prefix(self.activate_cmd):
            self.host.run("pip install '%s'" % name)

    def setup_env(self):
        """ Install everything inside the virtualenv """
        # From `self.packages`
        for p in self.packages:
            self.install_package(p)

        # From requirements.txt files
        self.install_requirements()
```

So we have created another `Node` class and moved some of the code we already had in there. The `setup_env` method is added to group the installation in one command. One other thing worth noting is the `location` class variable, to which `required_property()` was assigned. Actually, that is a property that raises an exception when it's accessed. The idea there is that we inherit from the `VirtualEnv` class and override this variable by an actual value.

Now, to use this in the `DjangoDeployment` node is now possible by nesting these classes. As said, we inherit from `VirtualEnv` and replace the variables by whatever we need. We also add a `setup` method in `DjangoDeployment` which will eventually do all the setup, so that we only have to call one method for the first initial setup of our deployment.

```
class DjangoDeployment(Node):
    ...

    class virtual_env(VirtualEnv):
        location = '~/.virtualenvs/project-env/'
        requirements_files = [ '~/git/django-project/requirements.txt' ]
        packages = [ 'gunicorn', 'supervisor' ]

    def setup(self):
        # Install virtual packages
        self.virtual_env.setup_env()

    ...
```

Did you see what we did? This `setup`-method does some magic. Take a look at how we access `virtual_env`. Normal Python code would return a `VirtualEnv` class at that point, so `self.virtual_env.setup_env` would be a classmethod and you would get a `TypeError: unbound method must be called with ...` exception. But in a `Node` class, Python acts differently, if we access one node class which is nested inside another, we'll automatically get a `Node` instance of the inner class.[3]

The reason will probably become clearer if you take a look The `self.host` variable. Calling run on `self.host` will execute commands on that host. Remember that we defined the host by nesting the `Hosts` class inside the `DjangoDeployment` node? We didn't have to do that for `virtual_env`, but `VirtualEnv` also expects `self.host.run` to work. The magic is what we call mapping of roles/hosts. If not explicitely defined, an instance of the nested class knows on which hosts to execute by looking at the parent instance, and they're linked because the framework instantiates the nested class at the point that we access from the parent.

You should not worry too much about what happens under the hood, it's a well tested and well thought through, but it can be hard to grasp at first.

[3] Internally, this works thanks to Python descriptors.

Reusable `git` class

Let's do something similar for the `git` class.

```
class Git(Node):
    project_directory = required_property()
    repository = required_property()

    def install(self):
        """ Installs the ``git`` package. """
        self.host.sudo('apt-get install git')

    def clone(self):
        """ Clone repository."""
        with self.host.cd(self.project_directory, expand=True):
            self.host.run("git clone '%s'" % esc1(self.repository))

    def checkout(self, commit):
        """ Checkout specific commit (after cloning)."""
        with self.host.cd('~/git/django-project', expand=True):
            self.host.run("git checkout '%s'" % esc1(commit))
```

And in `DjangoDeployment`:

```
class DjangoDeployment(Node):
    ...

    class git(Git):
        project_directory = '~/git/django-project'
        repository = 'git@github.com:example/example.git'

    def setup(self):
        # Clone repository
        self.git.clone()

        # Install virtual packages
        self.virtual_env.setup_env()
```

Our reusable `DjangoDeployment`

If we do the same exercise for the other parts of our script we get the following. The `Hosts` class is removed by purpose, the reason will become clear in the following section.

Let's save the following in a file called `django_deployment.py`:

```
from deployer.utils import esc1
from deployer.host import SSHHost

supervisor_config = \
"""
[program:djangoproject]
command = /home/username/.virtualenvs/project-env/bin/gunicorn_start  ; Command to␣
↪start app
user = username                                                        ; User to run as
stdout_logfile = /home/username/logs/gunicorn_supervisor.log          ; Where to␣
↪write log messages
redirect_stderr = true                                                 ; Save stderr␣
↪in the same log
```

```
"""

django_settings = \
"""
DATABASES['default'] = ...
SESSION_ENGINE = ...
DEFAULT_FILE_STORAGE = ...
"""

class VirtualEnv(Node):
    location = required_property()
    requirements_files = []
    packages = []

    # Command to execute to work on the virtualenv
    @property
    def activate_cmd(self):
        return  '. %s/bin/activate' % self.location

    def install_requirements(self):
        """
        Script to install the requirements of our Django application.
        (We have a requirements.txt file in our repository.)
        """
        with self.host.prefix(self.activate_cmd):
            for f in self.requirements_files:
                self.host.run("pip install -r '%s' " % esc1(f))

    def install_package(self, name):
        """
        Utility for installing packages through ``pip install`` inside
        the env.
        """
        with self.host.prefix(self.activate_cmd):
            self.host.run("pip install '%s'" % name)

    def setup_env(self):
        """ Install everything inside the virtualenv """
        # From `self.packages`
        for p in self.packages:
            self.install_package(p)

        # From requirements.txt files
        self.install_requirements()

class Git(Node):
    project_directory = required_property()
    repository = required_property()

    def install(self):
        """ Installs the ``git`` package. """
        self.host.sudo('apt-get install git')

    def clone(self):
        """ Clone repository."""
        with self.host.cd(self.project_directory, expand=True):
            self.host.run("git clone '%s'" % esc1(self.repository))
    def checkout(self, commit):
        """ Checkout specific commit (after cloning)."""
        with self.host.cd('~/git/django-project', expand=True):
            self.host.run("git checkout '%s'" % esc1(commit))

class DjangoDeployment(Node):
    class virtual_env(VirtualEnv):
        location = '~/.virtualenvs/project-env/'
        packages = [ 'gunicorn', 'supervisor' ]
        requirements_files = ['~/git/django-project/requirements.txt' ]

    class git(Git):
        project_directory = '~/git/django-project'
        repository = 'git@github.com:example/example.git'

    def setup(self):
        # Clone repository
        self.git.clone()

        # Install virtual packages
        self.virtual_env.setup_env()

    def upload_django_settings(self):
        """ Upload the content of the variable 'local_settings' in the
        local_settings.py file. """
        with self.host.open('~/git/django-project/local_settings.py') as f:
            f.write(django_settings)

    def run_management_command(self, command):
        """ Run Django management command in virtualenv. """
        # Activate the virtualenv.
        with self.host.prefix(self.activate_cmd):
            # Cd to the place where we have our 'manage.py' file.
            with self.host.cd('~/git/django-project/'):
                self.host.run('./manage.py %s' % command)

    def django_shell(self):
        """ Open interactive Django shell. """
        self.run_management_command('shell')

    def run_gunicorn(self):
        """ Run the gunicorn server """
        self.run_management_command('run_gunicorn')

    def upload_supervisor_config(self):
        """ Upload the content of the variable 'supervisor_config' in the
        supervisord configuration file. """
        with self.host.open('/etc/supervisor/conf.d/django-project.conf') as f:
            f.write(supervisor_config)
```

Adding hosts

The file that we saved to `django_deployment.py` in the previous section did not contain any hosts. So, it's rathar a template of a deployment script that we are going to apply here on a host. We inherit from `DjangoDeployment` and add the hosts.

```
#!/usr/bin/env python

class remote_host(SSHHost):
    address = '192.168.1.1' # Replace by your IP address
    username = 'user'       # Replace by your own username.
    password = 'password'   # Optional, but required for sudo operations
    key_filename = None     # Optional, specify the location of the RSA
                            #   private key
class DjangoDeploymentOnHost(DjangoDeployment):
    class Hosts:
        host = remote_host

    # Override a few properties of the parent.
    virtual_env__location = '~/.virtualenvs/project-env-2/'
    git__project_directory = '~/git/django-project-2'

if __name__ == '__main':
    start(DjangoDeploymentOnHost)
```

Class inheritance is powerful in Python. But did you notice the that we never had a `git__project_directory` or `virtual_env__location` variable before? This is again some magic. It's a pattern that very offen occurs in this framework. Python has no easy way to write that you want to override a property of the nested class. We introduced *double underscore expansion* which tells Python that in our case that if a member of a node class has double underscores in its name, it means that we are overriding a property of a nested node. In this case we override the `location` property of the `virtual_env` class of the parent and the value of `project_directory` of the nested `git` class.

That's it. This script is executable and if you start it, you have a nice interactive shell from which you can run all the commands.

And now?

The script can still even more be improved. For instance, in `deployer.contrib.nodes.config` is a nice `Config` class that we could use for managing the Django and supervisord settings. It contains a few handy functions for comparing the content of the remote file with that of what we would overwrite it with.

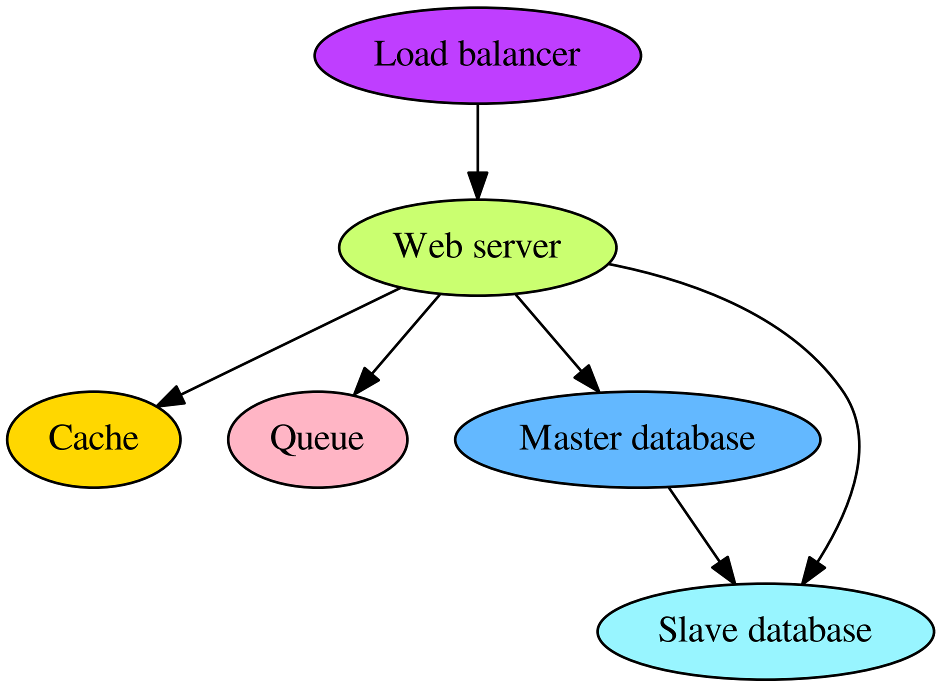
Also, learn about *query expressions* and the `parent` variable which are very powerful.

Architecture of roles and nodes

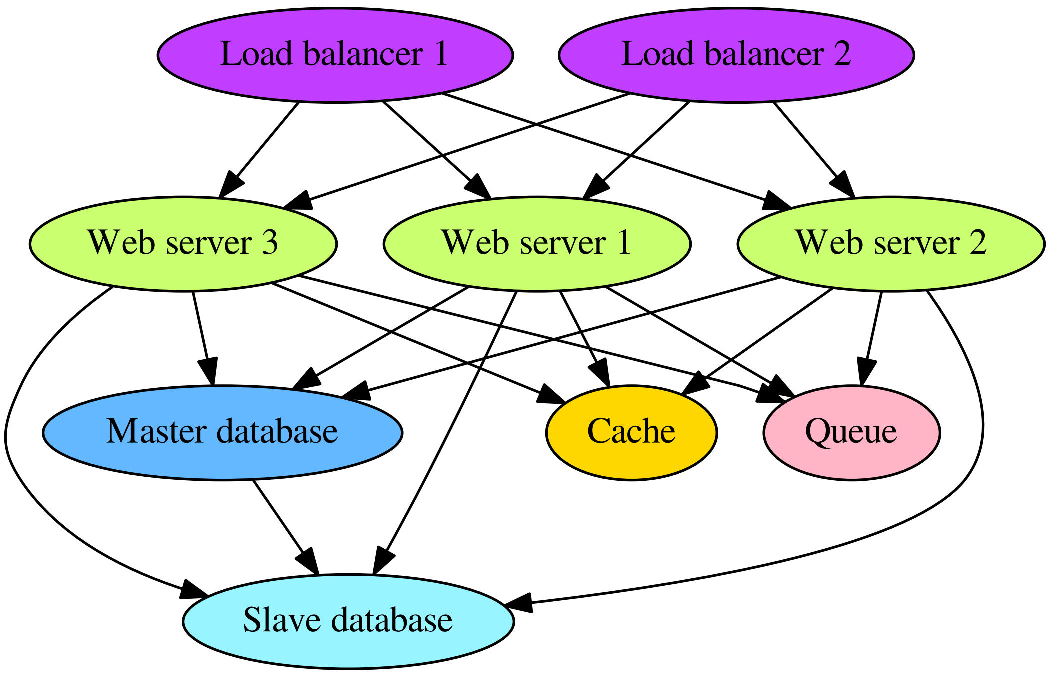
In this chapter, we go a little more in depth on what a `Node` really is.

Use cases

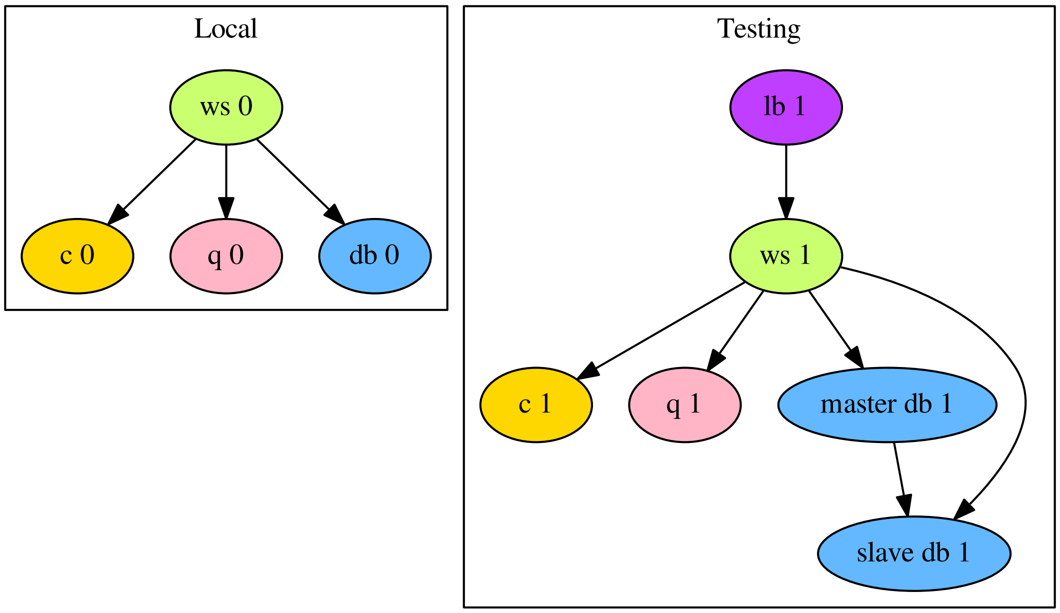
Before we go in depth, let's first look at a typical set-up of a web server. The following picture displays serveral connected components. It contains a web server connected to some database back-ends, and a load balancer in front of it. Every component appears exactly once.

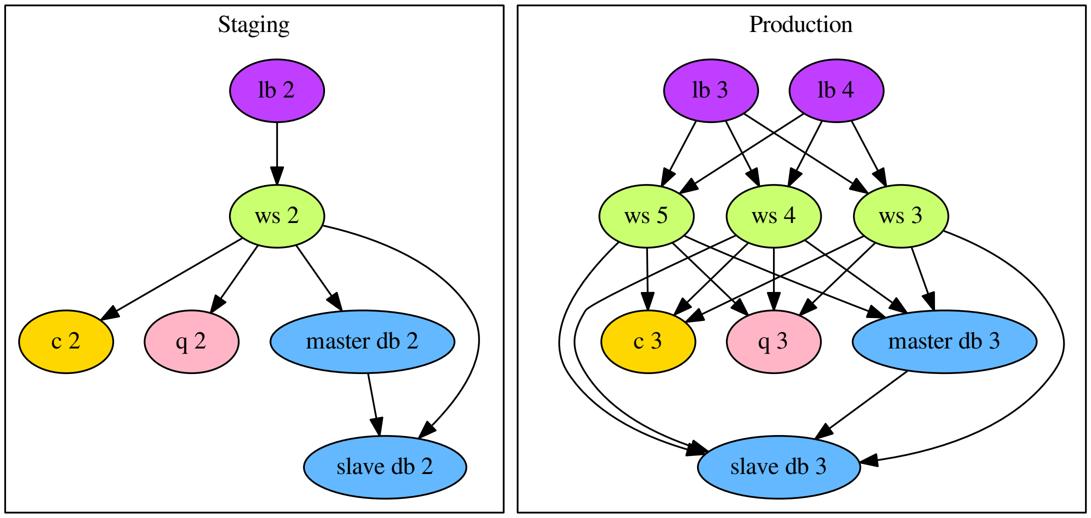


Now we are going to scale. If we triple the amount of web servers, and put an extra load balancer in front of our system. We end up with many more arrows.



It’s even possible that we have several instaces of all this. A local development set-up, a test server, a staging server, and a production server. Let’s see:





Obviously we don't want to write 4 different deployment scripts. The components are exacty the same every time, the only difference is that the amount of how many times a certain component appears is not always the same.

In this example, we can identify 4 roles:

- Load balancer
- Cache server

- Queue server
- Master database
- slave database

Creating nodes

Now we are going to create `Node` classes. A Node is probably the most important class in this framework, because basically all deployment code is structured in node. Every circle in the above diagrams can be considered a node.

So we are going to write a script that contains all these connected parts or nodes. Basically, it's one container node, and childnodes for all the components that we have. As an example, we also add the `Git` component where we'll put in the commands for checking the web server code out from our version control system.

```python
from deployer.node import Node

class WebSystem(Node):
    class Cache(Node):
        pass

    class Queue(Node):
        pass

    class LoadBalancer(Node):
        pass

    class Database(Node):
        pass

    class Git(Node):
        pass
```

The idea is that if we create multiple instances of `WebSystem` here, we only have to tell the root node which roles map to which hosts. We can use inheritance to override the `WebSystem` node and add `Hosts` to the derived classes. Wrapping it in `RootNode` is not really necessary, but cool to group these if we'd put an interactive shell around it.

```python
class RootNode(Node):
    class StagingSystem(WebSystem):
        class Hosts:
            load_balancer= { StagingHost0 }
            web = { StagingHost0  }
            master_db = { StagingHost0 }
            slave_db = set() # If empty, this line can be left away.
            queue = { StagingHost0 }
            cache = { StagingHost0 }

    class ProductionSystem(WebSystem):
        class Hosts:
            load_balancer = { LB0, LB1 }
            web = { WebServer1, WebServer2, WebServer3 }
            master_db = { MasterDB }
            slave_db = { SlaveDB }
            queue = { QueueHost }
            cache = { CacheHost }
```

Note that on the staging system, the same physical host is assigned to all the roles. That's fine: the web server can also act as load balancer, as well as a cache or queue server. On the production side, we separate them on different machines.

Now it's up to the framework to the figure out which hosts belong to which childnodes. With a little help of the map_roles decorator, that's possible. We adjust the original WebSystem node as follows:

```
from deployer.node import Node, map_roles

class WebSystem(Node):
    """
    roles: cache, queue, master_db, slave_db, web.
    """
    @map_roles(host='cache')
    class Cache(Node):
        pass

    @map_roles(host='queue')
    class Queue(Node):
        pass

    @map_roles(host='queue')
    class LoadBalancer(Node):
        pass

    @map_roles(master='master_db', slave='slave_db')
    class Database(Node):
        pass

    @map_roles(host=['www', 'load_balancer', 'queue'])
    class Git(Node):
        def checkout(self, commit):
            self.hosts.run('git checkout %s' % commit)
```

@map_roles needs a list of keyword arguments. The value can be either a string or list and decribes the roles of the parent node, and the key tells the new role in the child node to which these hosts are assigned.

If we now type self.hosts.run('shell command') in for instance the Database child node, it will only run in the hosts assigned there. In the case of our ProductionSystem above, that's on MasterDB and SlaveDB. In the case of Git.checkout above, the run-command will execute on all hosts that were mapped to the role host.

More complete example

Below, we present a more complete example with real actions like start and stop. The queue, the cache and the database, they have some methods in common, – in fact they are all upstart services –, so therefor we created a base class UpstartService that handles the common parts.

```
#!/usr/bin/env python
from deployer.node import Node, map_roles, required_property
from deployer.utils import esc1

from our_nodes import StagingHost0, LB0, LB1, WebServer1, WebServer2, \
        WebServer3, MasterDB, SlaveDB, QueueHost, CacheHost

class UpstartService(Node):
    """
    Abstraction for any upstart service with start/stop/status methods.
    """
    name = required_property()
    def start(self):
        self.hosts.sudo('service %s start' % esc1(self.name))

    def stop(self):
        self.hosts.sudo('service %s stop' % esc1(self.name))

    def status(self):
        self.hosts.sudo('service %s status' % esc1(self.name))

class WebSystem(Node):
    """
    The base definition of our web system.

    roles: cache, queue, master_db, slave_db, web.
    """
    @map_roles(host='cache')
    class Cache(UpstartService):
        name = 'redis'

    @map_roles(host='queue')
    class Queue(UpstartService):
        name = 'rabbitmq'

    @map_roles(host='queue')
    class LoadBalancer(Node):
        # ...
        pass

    @map_roles(master='master_db', slave='slave_db')
    class Database(UpstartService):
        name = 'postgresql'

    @map_roles(host=['www', 'load_balancer', 'queue'])
    class Git(Node):
        def checkout(self, commit):
            self.hosts.run('git checkout %s' % esc1(commit))

        def show(self):
            self.hosts.run('git show')

class RootNode(Node):
    """
    The root node of our configuration, containing two 'instances' of
    `WebSystem`,
    """
    class StagingSystem(WebSystem):
        class Hosts:
            load_balancer = { StagingHost0 }
            web = { StagingHost0  }
            master_db = { StagingHost0 }
            slave_db = set() # If empty, this line can be left away.
            queue = { StagingHost0 }
            cache = { StagingHost0 }

    class ProductionSystem(WebSystem):
        class Hosts:
            load_balancer = { LB0, LB1 }
            web = { WebServer1, WebServer2, WebServer3 }
            master_db = { MasterDB }
            slave_db = { SlaveDB }
            queue = { QueueHost }
            cache = { CacheHost }

if __name__ == '__main__':
    start(RootNode)
```

So, in this example, if `Staginghost0`, `LB0` and the others were real `deployer.host.Host` definitions, we could start *an interactive shell*. Then we could for instance navigate to the database of the production system, by typing “`cd ProductionSystem Database`” and then “`start`” to execute the command.

The interactive shell

It’s very easy to create an interactive command line shell from a node tree. Suppose that you have a *`Node`* called `MyRootNode`, then you can create a shell by making an executable file like this:

```
#!/usr/bin/env python
from deployer.client import start
from deployer.node import Node

class MyRootNode(Node):
    ...

if __name__ == '__main__':
    start(MyRootNode)
```

If you save this as `client.py` and call it by typing `python ./client.py --help`, the following help text will be shown:

```
Usage:
  ./client.py run [-s | --single-threaded | --socket SOCKET] [--path PATH]
                  [--non-interactive] [--log LOGFILE] [--scp]
                  [--] [ACTION PARAMETER...]
  ./client.py listen [--log LOGFILE] [--non-interactive] [--socket SOCKET]
  ./client.py connect (--socket SOCKET) [--path PATH] [--scp]
                  [--] [ACTION PARAMETER...]
  ./client.py telnet-server [--port PORT] [--log LOGFILE] [--non-interactive]
  ./client.py list-sessions
  ./client.py scp
  ./client.py -h | --help
  ./client.py --version

Options:
  -h, --help              : Display this help text.
  -s, --single-threaded   : Single threaded mode.
  --path PATH             : Start the shell at the node with this location.
  --scp                   : Open a secure copy shell.
  --non-interactive       : If possible, run script with as few interactions as
                            possible.  This will always choose the default
                            options when asked for questions.
  --log LOGFILE           : Write logging info to this file. (For debugging.)
  --socket SOCKET         : The path of the unix socket.
  --version               : Show version information.
```

There are several options to start such a shell. It can be multi or single threaded, or you can run it as a telnet-server. Assuming you made the file also executable using `chmod +x client.py`, you just type the following to get the interactive prompt:

```
./client.py run
```

Navigation

Navigation is very similar to navigating in a Bash shell.

Command	Meaning
`cd`	`cd node_name` will move to a certain node. `cd -` will move back to the previous node. `cd ..` will move to the and `cd /` will move to the root node. It's the same as a Bash shell, except that spaces are used instead of slashes when several nodes are chained, e.g. `cd node childnode`.
`ls`	Move to a certain node
`pwd`	Print current node (directory)
`find`	Recursively list all the childnode. Press `q` to quit the pager.
`exit`	Leave the deployment shell.
`clear`	Clear the screen.

Running node actions

In order to execute an action of the current node, just type the name of the action and press enter. Follow the action name by a space and a value if you want to pass that value as parameter.

Sandboxed execution is possible by preceding the action name by the word `sandbox`. e.g. type: `sandbox do_something param`. This will run the action, like usual, but it won't execute the actual commands on the hosts, instead it will execute a syntax-checking command instead.

Special commands

Some special commands, starting with double dash:

Command	Meaning
`--inspect`	Show information about the current node. This displays the file where the node has been defined, the hosts that are bound to this node and the list of actions child nodes that it contains.
`--source-code`	Display the source code of the current node.
`--connect`	Open an interactive (bash) shell on a host of this node. It will ask which host to connect if there are several hosts in this node.
`--version`	Show version information.
`--scp`	Open an SCP shell.
`--run`	Run a shell command on all hosts in the current node.
`--run-with-sudo`	Identical to `--run`, but using `sudo`

For `--inspect`, `--source-code` and `--connect`, it's possible to pass the name or path of another node as parameter. E.g.: `--connect node child_node`.

The SCP (secure copy) shell

Typing `--scp` in the main shell will open a subshell in which you can run SCP commands. This is useful for manually downloading and uploading files to servers.

Where	Command	Meaning
Remote	`cd <directory>`	Go to another directory at the server.
	`pwd`	Print working directory at the server.
	`stat <file>`	Print information about file or directory on the server.
	`edit <file>`	Open this file in an editor (vim) on the server.
	`connect`	Open interactive (bash) shell at the at the server.
Local	`lcd <directory>`	Go locally to another directory.
	`lpwd`	Print local working directory.
	`lstat <file>`	Print information about a local file or directory.
	`ledit <file>`	Open this local file in an editor
	`lconnect`	Open local interactive (bash) shell at this directory.
File operations	`put <file>`	Upload this local file to the server.
	`get <file>`	Download remote file from the server.
Other	`exit`	Return to the main shell.
	`clear`	Clear screen.

The node object

The *Node* class is probably the most important class of this framework. See *architecture of roles and nodes* for a high level overview of what a Node exactly is.

A simple example of a node:

```
from deployer.node import ParallelNode

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

Note: It is interesting to know that `self` is actually not a *Node* instance like you would expect, 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

In order to run methods of a node, it has to be wrapped in an *Env* object. This will manage execution, optional sandboxing, logging and much more. It will also make sure that *self.hosts* actually becomes a *HostsContainer*, a proxy through which you can run methods on a series of hosts.

The easiest way to wrap a node inside an *Env* is by using the *default_from_node()* helper. This will make sure that you can see the output and you can interact.

```
from deployer.node import Env
```

```
env = Env.default_from_node(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

The double underscore expansion is a kind of syntactic sugar to make overriding more readable.

Suppose we already had a node like this:

```
class WebApp(Node):
    class Nginx(Node):
        class Server(Node):
            domain = 'www.example.com'
```

Now, we'd like to inherit from `WebApp`, but change the `Nginx.Server.domain` property there to 'mydomain.com'. Normally, in Python, you do this:

```
class MyWebApp(WebApp):
    class Nginx(WebApp.Nginx):
        class Server(WebApp.Nginx.Server):
            domain = 'mydomain.com'
```

This is not too bad, but if you have a lot of nested classes, it can become pretty ugly. Therefor, the *`Node`* class has some magic which allows us to do this instead:

```
class MyWebApp(WebApp):
    Nginx__Server__domain = 'mydomain.com'
```

If you'd like, you can also use the same syntax to add function to the inner classes:

```
class MyWebApp(WebApp):
    def Nginx__Server__get_full_domain(self):
        # Note that 'self' points to the 'Server' class at this point,
        # not to 'Webapp'!
        return 'http://%s' % self.domain
```

The importance of `ParallelNode`

There are several kind of setups. You can have many hosts which are all doing exactly the same, or many hosts that do something different. Simply said, *`ParallelNode`* should be used when you have many hosts in your node that all do exactly the same. Actions on such a *`ParallelNode`* can be executed in parallel. The hosts are equal but also independend and don't need to know about each other. An example is an array of stateless web servers.

A typical setup consists of a root node which is just a normal *`Node`*, with several arrays of *`ParallelNode`* nested inside.

Isolation of hosts in `ParallelNode`.

Take the following example:

```
class WebSystem(ParallelNode):
    class Hosts:
        host = { Host1, Host2, Host3, Host4 }

    def checkout_git(self, commit):
        self.host.run("git checkout '%s'" % esc1(commit))

    def restart(self):
        self.host.run("nginx restart")

    def deploy(self, commit):
        self.checkout_git(commit)
        self.restart()
```

We see a *`ParallelNode`* class with three actions and four Hosts mapped to the role `host` of this node. Because of the isolation that *`ParallelNode`* provides, it is possible to call any of the four actions independently on any of the four hosts. Look how our `WebSystem` acts like an array:

```
websystem = Env.default_from_node(WebSystem())
websystem[Host1].deploy('abcde6565eee...')
websystem[Host2].restart()
```

We can also call an action directly without specifying a host. This will allow parallel execution. It says: call this action on every cell of the array. They are independent and unordered in this case, so we don't have to run the deploy sequentially.

```
websystem = Env.default_from_node(WebSystem())
websystem.deploy('abcde6565eee...') # Parallel execution.
```

Note: One thing worth noting is that there is a variable *`host`* in the class. This is because the isolation always happens by convention on the role named `host`. Both sides of the following equation will represent a *`HostContainer`* containing exactly one host: the host of the current isolation.

```
self.host == self.hosts.filter('host')
```

If there happen to be hosts mapped to other roles, they will simply become available for every instance in the role named `host`. If you'd call `self.hosts.filter('other_role')`, that would still work.

.Array and .JustOne

`.Array` and `.JustOne` are required for nesting a *`ParallelNode`* inside a normal *`Node`*. The idea is that when host roles are mapped from the parent *`Node`*, to the child – which is a *`ParallelNode`* –, that this childnode behaves as an array. Each 'cell' in the array is isolated, so it's possible to execute a command on just one 'cell' (or host) of the array or all 'cells' (or hosts.) You can use it as follows:

```
class NormalNode(Node):
    class OurParallelNode(ParallelNode.Array):
        class PNode(ParallelNode):
            pass
```

Basically, you can nest 'normal' nodes inside each other, and `ParallelNode` classes inside each other. However, when nesting such a `ParallelNode` inside a normal node, the `.Array` suffix is required to indicate the creation of an array. `.JustOne` can always be used instead of an array, if you assert that only one host will be in there.

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

See *Node reference*.

Node reference

Note: Maybe it's useful to read the read about *the node object* first.

class `deployer.node.base.`**`Action`**(*attr_name*, *node_instance*, *func*, *is_property=False*, *is_query=False*, *query=None*)

Node actions, which are defined as just functions, will be wrapped into this Action class. When one such action is called, this class will make sure that a correct `env` object is passed into the function as its first argument. :param node_instance: The Node Env to which this Action is bound. :type node_instance: None or `deployer.node.Env`

class `deployer.node.base.`**`Env`**(*node*, *pty=None*, *logger=None*, *is_sandbox=False*)

Wraps a `deployer.node.Node` into an executable context.

```
n = Node()
e = Env(n)
e.do_action()
```

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

Note: Node actions can never be executed directly on the node instance, without wrapping it in an Env object first. But if you use the *interactive shell*, the shell will do this for you.

Parameters

- **`node`** (`deployer.node.Node`) – The node that this `Env` should wrap.

- **pty** (*deployer.pseudo_terminal.Pty*) – The terminal object that wraps the input and output streams.
- **logger** (deployer.logger.LoggerInterface) – (optional) The logger interface.
- **is_sandbox** (*bool*) – Run all commands in here in sandbox mode.

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

classmethod default_from_node (*node*)
Create a default environment for this node to run.

It will be attached to stdin/stdout and commands will be logged to stdout. The is the most obvious default to create an Env instance.

Parameters node – *Node* 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.base.**EnvAction** (*env*, *action*)
Action wrapped by an Env object. Calling this will execute the action in the environment.

class deployer.node.base.**IsolationIdentifierType**
Manners of identifing a node in an array of nodes.

HOSTS_SLUG = 'HOSTS_SLUG'
Use a tuple of Host slugs

HOST_TUPLES = 'HOST_TUPLES'
Use a tuple of Host classes

INT_TUPLES = 'INT_TUPLES'
Use a tuple of integers

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

For the attributes, also have a look at the proxy class deployer.node.Env. The parent parameter is used internally to pass the parent Node instance into here.

Hosts = None
Hosts can be None or a definition of the hosts that should be used for this node. e.g.:

```
class MyNode(Node):
    class Hosts:
        role1 = [ LocalHost ]
        role2 = [ SSHHost1, SSHHost2]
```

parent
Reference to the parent *Node*. (This is always assigned in the constructor. You should never override it.)

class deployer.node.base.**NodeBase**
Metaclass for Node. This takes mostly care of wrapping Node members into the correct descriptor, but it does some metaclass magic.

class deployer.node.base.**ParallelActionResult**(*isolations_and_results*)
When an action of a ParallelNode was called from outside the parallel node itself, a *ParallelActionResult* instance is returned. This contains the result for each isolation.

(Unconventional, but) Iterating through the *ParallelActionResult* class will yield the values (the results) instead of the keys, because of backwards compatibility and this is typically what people are interested in if they run: `for result in node.action(...)`.

The *keys*, *items* and *values* functions work as usual.

class deployer.node.base.**ParallelNode**(*parent=None*)
A `ParallelNode` 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'.

If you call a method on a `ParallelNode`, it will be called one for every host, which can be accessed through the `host` property.

Note This was called *SimpleNode* before.

host
This is the proxy to the active host.

Returns *HostContainer* instance.

deployer.node.base.**SimpleNode**
Deprecated alias for ParallelNode

alias of *ParallelNode*

deployer.node.base.**SimpleNodeBase**
Deprecated alias for ParallelNodeBase

alias of `ParallelNodeBase`

deployer.node.base.**iter_isolations**(*node*, *identifier_type='INT_TUPLES'*)
Yield (index, Node) for each isolation of this node.

class deployer.node.base.**required_property**(*description=''*)
Placeholder for properties which are required when a service is inherit.

```
class MyNode(Node):
    name = required_property()

    def method(self):
        # This will raise an exception, unless this class was
        # inherited, and `name` was filled in.
        print (self.name)
```

Decorators

`deployer.node.decorators.`**`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.decorators.`**`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.decorators.`**`isolate_one_only`**(*func*)

When using role isolation, and several hosts are available, run on only one role. Useful for instance, for a database client. it does not make sense to run the interactive client on every host which has database access.

`deployer.node.decorators.`**`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.

Role mapping

`deployer.node.role_mapping.`**`ALL_HOSTS`** = **ALL_HOSTS**

Constant to indicate in a role mapping that all hosts of the parent should be mapped to this role.

`deployer.node.role_mapping.`**`map_roles`**

alias of `RoleMapping`

class `deployer.node.role_mapping.`**`DefaultRoleMapping`**(**host_mapping*, ***mappings*)

Default mapping: take the host container from the parent.

Host

This module contains the immediate wrappers around the remote hosts and their terminals. It's possible to run commands on a host directly by using these classes. As an end-user of this library however, you will call the methods of *`SSHHost`* and *`LocalHost`* through *`HostsContainer`*, the host proxy of a *`Node`*.

Base classes

class `deployer.host.base.`**`Host`**(*pty=None*, *logger=None*)

Abstract base class for SSHHost and LocalHost.

Parameters

- **`pty`** (*`deployer.pseudo_terminal.Pty`*) – The pseudo terminal wrapper which handles the stdin/stdout.
- **`logger`** (*`LoggerInterface`*) – The logger interface.

```
class MyHost(SSHHost):
    ...
my_host = MyHost()
my_host.run('pwd', interactive=False)
```

copy(*pty=None*)
Create a deep copy of this Host class. (the pty-parameter allows to bind it to anothor pty)

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

get_file(*remote_path*, *local_path*, *use_sudo=False*, *sandbox=False*)
Download this remote_file.

get_ip_address(*interface='eth0'*)
Return internal IP address of this interface.

get_start_path()
The path in which commands at the server will be executed. by default. (if no cd-statements are used.) Usually, this is the home directory. It should always return an absolute path, starting with '/'

getcwd()
Return current working directory as absolute path.

ifconfig()
Return the network information for this host.

Returns An *`IfConfig`* instance.

listdir_stat(*path='.'*)
Return a list of *`Stat`* instances for each file in this directory.

open(*remote_path*, *mode='rb'*, *use_sudo=False*, *sandbox=False*)
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')
```

password = ''
Password for connecting to the host. (for sudo)

put_file(*local_path*, *remote_path*, *use_sudo=False*, *sandbox=False*)
Upload this local_file to the remote location.

run(*command*, *use_sudo=False*, *sandbox=False*, *interactive=True*, *user=None*, *ignore_exit_status=False*, *initial_input=None*, *silent=False*)
Execute this shell command on the host.

Parameters

- **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.

- **initial_input** – When `interactive`, send this input first to the host.

slug = ‘’
The slug should be a unique identifier for the host.

start_interactive_shell (*command=None*, *initial_input=None*)
Start an interactive bash shell.

sudo (*command*, *use_sudo=False*, *sandbox=False*, *interactive=True*, *user=None*, *ignore_exit_status=False*, *initial_input=None*, *silent=False*)
Wrapper around *run()* which uses `sudo`.

username = ‘’
Username for connecting to the Host

class `deployer.host.base.`**HostContext**
A push/pop stack which keeps track of the context on which commands at a host are executed.

(This is mainly used internally by the library.)

cd (*path*, *expand=False*)
Execute commands in this directory. Nesting of cd-statements is allowed.

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

Parameters **expand** (*bool*) – Expand tildes.

copy ()
Create a deep copy.

env (*variable*, *value*, *escape=True*)
Set this environment variable

```
with host.cd('VAR', 'my-value'):
    host.run('echo $VAR')
```

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

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

class `deployer.host.base.`**Stat** (*stat_result*, *filename*)
Base *Stat* class

is_dir
True when this is a directory.

is_file
True when this is a regular file.

st_gid
Group ID

st_size
File size in bytes.

st_uid
User ID

Localhost

class `deployer.host.local.`**`LocalHost`**(*pty=None*, *logger=None*)

`LocalHost` can be used instead of `SSHHost` for local execution. It uses `pexpect` underneat.

`listdir_stat`(*path='.'*)

Return a list of `Stat` instances for each file in this directory.

`start_interactive_shell`(*command=None*, *initial_input=None*)

Start an interactive bash shell.

SSH Host

class `deployer.host.ssh.`**`SSHHost`**(**a*, ***kw*)

SSH Host.

For the authentication, it's required to provide either a `password`, a `key_filename` or `rsa_key`. e.g.

```
class WebServer(SSHHost):
    slug = 'webserver'
    password = '...'
    address = 'example.com'
    username = 'jonathan'
```

`address` = **'example.com'**

SSH Address

`config_filename` = **'~/.ssh/config'**

SSH config file (optional)

`get_start_path`()

The path in which commands at the server will be executed. by default. (if no cd-statements are used.) Usually, this is the home directory. It should always return an absolute path, starting with '/'

`keepalive_interval` = **30**

SSH keep alive in seconds

`key_filename` = **None**

RSA key filename (optional)

`listdir_stat`(*path='.'*)

Return a list of `Stat` instances for each file in this directory.

`port` = **22**

SSH Port

`rsa_key` = **None**

RSA key. (optional)

`rsa_key_password` = **None**

RSA key password. (optional)

`start_interactive_shell`(*command=None*, *initial_input=None*, *sandbox=False*)

Start /bin/bash and redirect all SSH I/O from stdin and to stdout.

`timeout` = **10**

Connection timeout in seconds.

`username` = **''**

SSH Username

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 `Env` wrapper around 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.`**`HostsContainer`**(*hosts*, *pty=None*, *logger=None*, *is_sandbox=False*)

Proxy to a group of `Host` instances.

For instance, if you have a role, name 'www' inside the container, you could do:

```
host_container.run(...)
host_container[0].run(...)
host_container.filter('www')[0].run(...)
```

Typically, you get a `HostsContainer` class by accessing the `hosts` property of an `Env` (`Node` wrapper.)

`cd`(*path*, *expand=False*)

Execute commands in this directory. Nesting of cd-statements is allowed.

Call `cd()` on the `HostContext` of every host.

```
with host_container.cd('directory'):
    host_container.run('ls')
```

`env`(*variable*, *value*, *escape=True*)

Sets an environment variable.

This calls `env()` on the `HostContext` of every host.

```
with host_container.cd('VAR', 'my-value'):
    host_container.run('echo $VAR')
```

`exists`(*filename*, *use_sudo=True*)

Returns an array of boolean values that represent whether this a file with this name exist for each host.

`filter`(**roles*)

Returns a new HostsContainer instance, containing only the hosts matching this filter. The hosts are passed by reference, so if you'd call *cd()* on the returned container, it will also effect the hosts in this object.

Examples:

```
hosts.filter('role1', 'role2')
```

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

get_hosts()
Return a set of `deployer.host.Host` classes that appear in this container. Each `deployer.host.Host` class will abviously appear only once in the set, even when it appears in several roles.

get_hosts_as_dict()
Return a dictionary which maps all the roles to the set of `deployer.host.Host` classes for each role.

getcwd()
Calls `getcwd()` for every host and return the result as an array.

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

prefix(*command*)
Call `prefix()` on the `HostContext` of every host.

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

run(*command*, *sandbox=False*, *interactive=True*, *user=None*, *ignore_exit_status=False*, *initial_input=None*)
Call `run()` with this parameters on every `Host` in this container. It can be executed in parallel when we have multiple hosts.

Returns An array of all the results.

sudo(*command*, *sandbox=False*, *interactive=True*, *user=None*, *ignore_exit_status=False*, *initial_input=None*)
Call `sudo()` with this parameters on every `Host` in this container. It can be executed in parallel when we have multiple hosts.

Returns An array of all the results.

class `deployer.host_container.`**HostContainer**(*hosts*, *pty=None*, *logger=None*, *is_sandbox=False*)
Similar to `HostsContainer`, but wraps only around exactly one `Host`.

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

get_file(**args*, ***kwargs*)
Download this remote_file.

getcwd()
Calls `getcwd()` for every host and return the result as an array.

has_command(*command*, *use_sudo=False*)
Test whether this command can be found in the bash shell, by executing a `which` Returns `True` when the command exists.

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_file(**args*, ***kwargs*)
Upload this local_file to the remote location.

run(*command*, *sandbox=False*, *interactive=True*, *user=None*, *ignore_exit_status=False*, *initial_input=None*)
Call `run()` with this parameters on every `Host` in this container. It can be executed in parallel when we have multiple hosts.

Returns An array of all the results.

sudo(*command*, *sandbox=False*, *interactive=True*, *user=None*, *ignore_exit_status=False*, *initial_input=None*)
Call `sudo()` with this parameters on every `Host` in this container. It can be executed in parallel when we have multiple hosts.

Returns An array of all the results.

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.

color = None
Colour for this service/action in the shell. Right now, only the colours from the `termcolor` library are supported:

grey, red, green, yellow, blue, magenta, cyan, white

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

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

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`.

Parameters **`pty`** – *`deployer.pseudo_terminal.Pty`* instance.

`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.

`is_interactive`

When `False` don't ask for input and choose the default options when possible.

`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)

`progress_bar`(*message*, *expected=None*, *clear_on_finish=False*, *format_str=None*)

Display a progress bar. This returns a Python context manager. Call the next() method to increase the counter.

```
with console.progress_bar('Looking for nodes') as p:
    for i in range(0, 1000):
        p.next()
        ...
```

Returns `ProgressBar` instance.

Parameters **`message`** – Text label of the progress bar.

`progress_bar_with_steps`(*message*, *steps*, *format_str=None*)
Display a progress bar with steps.

```
steps = ProgressBarSteps({
    1: "Resolving address",
    2: "Create transport",
    3: "Get remote key",
    4: "Authenticating" })

with console.progress_bar_with_steps('Connecting to SSH server', steps=steps)␣
↪as p:
    ...
    p.set_progress(1)
    ...
    p.set_progress(2)
    ...
```

Parameters

- **`steps`** – `ProgressBarSteps` instance.
- **`message`** – Text label of the progress bar.

`pty`
The *`deployer.pseudo_terminal.Pty`* of this console.

`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.

`warning`(*text*)
Print a warning.

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()
```

Some usecases:

- Suppose that you have a huge deployment tree, covering dozens of projects, each having both a staging and production set-up, and all of them are doing a git checkout. Now you want to list all the current checkouts of all the repositories on all your machines. This is easy by traversing the nodes, filtering on the type *gitnode* and calling *git show* in there.
- Suppose you have an *nginx* node, which generates the configuration according to the childnodes in there. One childnode could for instance define a back-end, another one could define the location of static files, etc... By using this inspection module, you cat find the childnodes that contain a configuration section and combine these.
- Internally, the whole interactive shell is also using quite a lot of reflection.

Inspector

Reflexion/introspection on a *deployer.node.Node*

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.

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.

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.

Query expressions

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. The equivalent of `Q.parent.do_something` is:

```
@property
def do_something(self):
    return self.parent.do_something
```

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.

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 *`IfConfig`* objects.

Example usage:

```
ifconfig = parse_ifconfig_output(host.run('ifconfig'))
interface = ifconfig.get_interface('eth0')
print interface.ip
```

class `deployer.utils.network.`**`IfConfig`**

Container for the network settings, found by *ifconfig*. This contains a list of *`NetworkInterface`*.

`get_address`(*ip*)

Return the *`NetworkInterface`* object, given an IP addres (e.g. "127.0.0.1") or raise *AttributeError*.

`get_interface`(*name*)

Return the *`NetworkInterface`* object, given an interface name (e.g. "eth0") or raise *AttributeError*.

`interfaces`

List of all *`NetworkInterface`* objects.

class `deployer.utils.network.`**`NetworkInterface`**(*name='eth0'*)

Information about a single network interface.

`ip`

IP address of the network interface. e.g. "127.0.0.1"

`name`

Name of the network interface. e.g. "eth0".

Exceptions

exception `deployer.exceptions.`**`ActionException`**(*inner_exception*, *traceback*)

When an action fails.

exception `deployer.exceptions.`**`ConnectionFailedException`**
When connecting to an SSH host 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.

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 isn'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*, *term_var=''*)
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.)

stdin
Return the input file object.

stdout
Return the output file object.

`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.

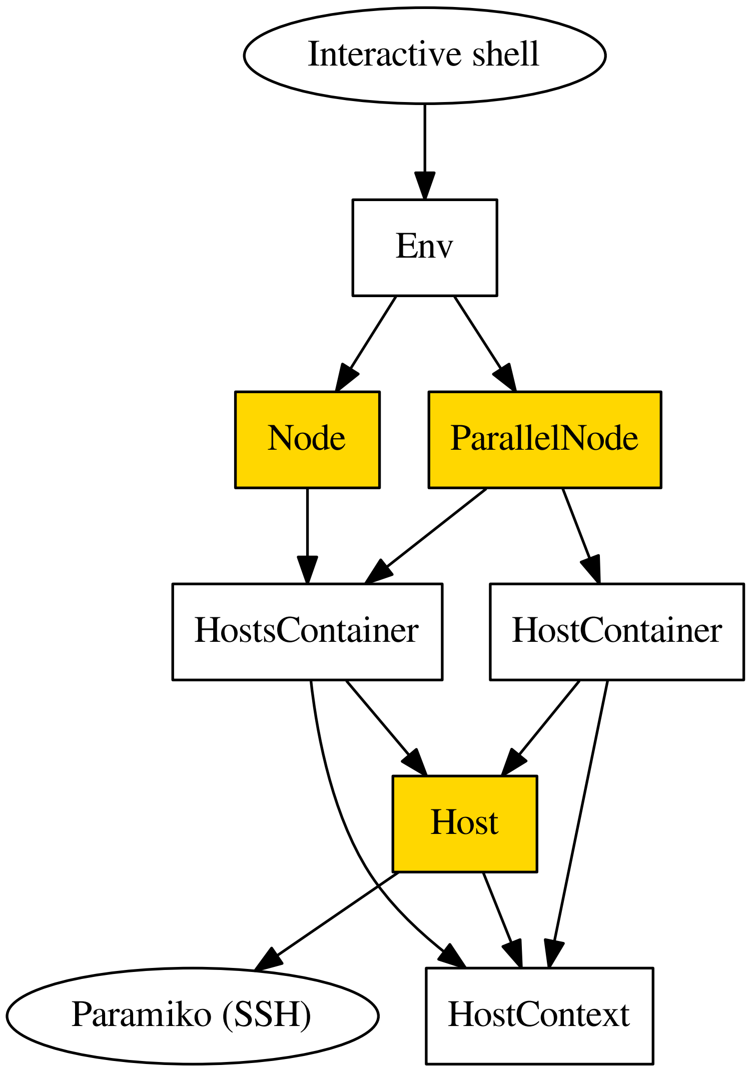
Internals

This page will try to give a high level overview of how the framework is working. While the end-user of the framework won't usually touch much more than the `Node` and `Host` classes, there's a lot more going on underneat.

There's a lot of meta-programming, some domain specific languages, and a mix of event-driven and blocking code.

Data flow

Roughly, this is the current flow from the interactive shell untill the actual SSH client.



The yellow classes – *Node*, *ParallelNode* and `Host` – are the ones which an average end-user of this framework will use. He will inherit from there to define his deployment script.

HostContainer (singular and plural) and *Env* are proxy classes. They are created by the framework, but passed to the user's code at some points.

`Paramiko`, at the lowest level, is responsible for the SSH connection. The `Host` class takes care of calling Paramiko, the end-user should not directly depend on Paramiko. In the future, we may replace it with for instance twisted.conch.

At the top level, we usually have the interactive shell. But if a deployment script is called as a library, it can have any other front-end. The built-in interactive shell also has a telnet server (remote shell) and a shell which has some multithreaded execution model (parallel deployment). These are realized through Twisted Matrix, and there's some event-driven code touching the iterative blocking code.

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. Also thanks for all the useful input from all the people I met.

Authors

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

Python Module Index

Index