fuselage documentation

Release 0.0.1

John Carr

July 06, 2014

Contents

The `fuselage` module provides a Python API for declaring and applying configuration to a compute resource such as a Linux server.

Fuselage is not an imperative system for server management. Instead it is a framework for building configuration bundles that can be applied to any system with a python interpreter.

It does not mandate a transport/execution mechanism. A compiled bundle could be transferred to the target system by ssh, https or even flash drive.

Contents:

CHAPTER 1

Defining configuration

1.1 Resource bundles

A fuselage bundle is a list of resources to manage and the configuration to apply to them. You use the `add` method to build a bundle:

```
from fuselage.bundle import ResourceBundle
from fuselage.resources import *

bundle = ResourceBundle()
bundle.add(Package(name="apache2"))
```

You can assemble a bundle into a compressed archive using a builder object:

```
from fuselage.builder import Builder

builder = Builder.write_to_path('/tmp/example_payload')
builder.embed_fuselage_runtime()
builder.embed_resource_bundle(bundle)
```

The output is a zipfile that can be executed by python. On linux and OSX it can even be executed directly:

```
/tmp/example_payload
```

1.2 Dependencies between resources

Resources are always applied in the order they are listed in the resource bundle. But if you want to express relationships between steps (for example, you want to run a command after updating a checkout) then you can use the `watches` argument.

For example:

```
bundle.add(Checkout(
    name="/usr/local/src/mycheckout",
    repository="git://github.com/example/example_project",
))

bundle.add(Execute(
    name="install-requirements",
    command="/var/sites/myapp/bin/pip install -r /usr/local/src/mycheckout/requirements.txt",
```

```
    watches=['/usr/local/src/mycheckout'],
))
```

When the `Checkout` step pulls in a change from a repository, the `Execute` resource will be applied.

You can do the same for monitoring file changes too:

```
bundle.add(File(
    name="/etc/apache2/security.conf",
    source="apache2/security.conf",
))

bundle.add(Execute(
    name="restart-apache",
    command="apache2ctl graceful",
    watches=['/etc/apache2/security.conf'],
))
```

Sometimes you can't use `File` (perhaps `buildout` or `maven` or similar generates a config file for you), but you still want to trigger a command when a file changes during deployment:

```
bundle.add(Execute(
    name="buildout",
    command="buildout -c production.cfg",
    changes=['/var/sites/mybuildout/parts/apache.cfg'],
))

bundle.add(Execute(
    name="restart-apache",
    command="apache2ctl graceful",
    watches=['/var/sites/mybuildout/parts/apache.cfg'],
))
```

This declares that the `buildout` step might change `/var/sites/mybuildout/parts/apache.cfg`). Subsequent steps can then subscribe to this file as though it was an ordinary `File` resource.

All of these examples use a trigger system. When a trigger has been set fuselage will remember it between invocations. Consider the following example:

```
bundle.add(File(
    name="/etc/apache2/sites-enabled/mydemosite",
))

bundle.add(File(
    name="/var/local/tmp/this/paths/parent/dont/exist",
))

bundle.add(Execute(
    name="restart-apache2",
    command="/etc/init.d/apache2 restart",
    watches=['/etc/apache2/sites-enabled/mydemosite'],
))
```

By inspection we can tell the 2nd step will fail, so will fuselage restart apache when we've fixed the bug? Yes:

- On the first run, fuselage will create the file and because a change occurred it will set a trigger for any resources that have a `watches` against it. This will be persisted in `/var/lib/yaybu/event.json` immediately.
- It will fail to create a Directory and stop processing changes.
- A human will correct the configuration and re-run it

- On the 2nd run it will check the file exists with the correct permissions and make no changes
- It will create the directory.
- Before running the restart it will check `event.json` to see if it is triggered or not.
- It will run the restart
- The trigger will be immediately removed from `event.json`.

This means that the restart step will always execute when the file changes, even if an intermediate step fails and the process has to be repeated. If the restart fails then fuselage will try again the next time it is run.

CHAPTER 2

Resources

This section describes the core resources you can use to describe your server configuration.

2.1 Line

Ensure that a line is present or missing from a file.

For example, this will ensure that selinux is disabled:

```
Line(
    name="/etc/selinux/config",
    match=r"^SELINUX",
    replace="SELINUX=disabled",
)
```

The available parameters are:

name The full path to the file to perform an operation on.

match The python regular expression to match the line to be updated.

replace The text to insert at the point the expression matches (otherwise at the end of the file).

2.2 File

A provider for this resource will create or amend an existing file to the provided specification.

For example, the following will create the /etc/hosts file based on a static local file:

```
File(
    name="/etc/hosts",
    owner="root",
    group="root",
    mode=0o644,
    source="my_hosts_file",
)
```

The available parameters are:

name The full path to the file this resource represents.

owner A unix username or UID who will own created objects. An owner that begins with a digit will be interpreted as a UID, otherwise it will be looked up using the python 'pwd' module.

group A unix group or GID who will own created objects. A group that begins with a digit will be interpreted as a GID, otherwise it will be looked up using the python 'grp' module.

mode A mode representation as an octal. This can begin with leading zeros if you like, but this is not required. DO NOT use yaml Octal representation (0o666), this will NOT work.

source An optional file that is rendered into `name` on the target. Fuselage searches the searchpath to find it.

contents The arguments passed to the renderer.

2.3 Directory

A directory on disk. Directories have limited metadata, so this resource is quite limited.

For example:

```
Directory(
    name="/var/local/data",
    owner="root",
    group="root",
    mode=0o755,
)
```

The available parameters are:

name The full path to the directory on disk

owner The unix username who should own this directory, by default this is 'root'

group The unix group who should own this directory, by default this is 'root'

mode The octal mode that represents this directory's permissions, by default this is '755'.

parents Create parent directories as needed, using the same ownership and permissions, this is False by default.

2.4 Link

A resource representing a symbolic link. The link will be from *name* to *to*. If you specify owner, group and/or mode then these settings will be applied to the link itself, not to the object linked to.

For example:

```
Link(
    name="/etc/init.d/exampled",
    to="/usr/local/example/sbin/exampled",
    owner="root",
    group="root",
)
```

The available parameters are:

name The name of the file this resource represents.

owner A unix username or UID who will own created objects. An owner that begins with a digit will be interpreted as a UID, otherwise it will be looked up using the python 'pwd' module.

group A unix group or GID who will own created objects. A group that begins with a digit will be interpreted as a GID, otherwise it will be looked up using the python 'grp' module.

to The pathname to which to link the symlink. Dangling symlinks ARE considered errors in Fuselage.

2.5 Execute

Execute a command. This command *is* executed in a shell subprocess.

For example:

```
Execute(
    name="add-apt-key",
    command="apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 0x000000"
)
```

The available parameters are:

name The name of this resource. This should be unique and descriptive, and is used so that resources can reference each other.

command If you wish to run a single command, then this is the command.

commands If you wish to run multiple commands, provide a list

cwd The current working directory in which to execute the command.

environment The environment to provide to the command, for example:

```
Execute(
    name="example",
    command="echo $FOO",
    environment={"FOO": "bar"},
)
```

returncode The expected return code from the command, defaulting to 0. If the command does not return this return code then the resource is considered to be in error.

user The user to execute the command as.

group The group to execute the command as.

umask The umask to use when executing this command

unless A command to run to determine is this execute should be actioned

creates The full path to a file that execution of this command creates. This is used like a "touch test" in a Makefile. If this file exists then the execute command will NOT be executed.

touch The full path to a file that fuselage will touch once this command has completed successfully. This is used like a "touch test" in a Makefile. If this file exists then the execute command will NOT be executed.

2.6 Checkout

This represents a "working copy" from a Source Code Management system. This could be provided by, for example, Subversion or Git remote repositories.

Note that this is '*a* checkout', not 'to checkout'. This represents the resource itself on disk. If you change the details of the working copy (for example changing the branch) the provider will execute appropriate commands (such as `svn switch`) to take the resource to the desired state.

For example:

```
Checkout(
    name="/usr/src/myapp",
    repository="https://github.com/myusername/myapp",
    scm="git",
)
```

The available parameters are:

name The full path to the working copy on disk.

repository The identifier for the repository - this could be an http url for subversion or a git url for git, for example.

branch The name of a branch to check out, if required.

tag The name of a tag to check out, if required.

revision The revision to check out or move to.

scm The source control management system to use, e.g. subversion, git.

scm_username The username for the remote repository

scm_password The password for the remote repository.

user The user to perform actions as, and who will own the resulting files. The default is root.

group The group to perform actions as. The default is to use the primary group of `user`.

mode A mode representation as an octal. This can begin with leading zeros if you like, but this is not required. DO NOT use yaml Octal representation (0o666), this will NOT work.

2.7 Package

Represents an operating system package, installed and managed via the OS package management system. For example, to ensure these three packages are installed:

```
Package(
    name="apache2",
)
```

The available parameters are:

name The name of the package. This can be a single package or a list can be supplied.

version The version of the package, if only a single package is specified and the appropriate provider supports it (the Apt provider does not support it).

purge When removing a package, whether to purge it or not.

When installing a package `apt-get` may give a `404` error if your local apt cache is stale. If Fuselage thinks this might be the cause it will `apt-get update` and retry before giving up.

2.8 User

A resource representing a UNIX user in the password database. The underlying implementation currently uses the "useradd" and "usermod" commands to implement this resource.

This resource can be used to create, change or delete UNIX users.

For example:

```
User(
    name="django",
    fullname="Django Software Owner",
    home="/var/local/django",
    system=True,
    disabled_password=True,
)
```

The available parameters are:

name The username this resource represents.

password The encrypted password, as returned by crypt(3). You should make sure this password respects the system's password policy.

fullname The comment field for the password file - generally used for the user's full name.

home The full path to the user's home directory.

uid The user identifier for the user. This must be a non-negative integer.

gid The group identifier for the user. This must be a non-negative integer.

group The primary group for the user, if you wish to specify it by name.

groups A list of supplementary groups that the user should be a member of.

append A boolean that sets how to apply the groups a user is in. If true then fuselage will add the user to groups as needed but will not remove a user from a group. If false then fuselage will replace all groups the user is a member of. Thus if a process outside of fuselage adds you to a group, the next deployment would remove you again.

system A boolean representing whether this user is a system user or not. This only takes effect on creation - a user cannot be changed into a system user once created without deleting and recreating the user.

shell The full path to the shell to use.

disabled_password A boolean for whether the password is locked for this account.

disabled_login A boolean for whether this entire account is locked or not.

2.9 Group

A resource representing a unix group stored in the /etc/group file. `groupadd` and `groupmod` are used to actually make modifications.

For example:

```
Group(
    name="zope",
    system=True,
)
```

The available parameters are:

name The name of the unix group.

gid The group ID associated with the group. If this is not specified one will be chosen.

system Whether or not this is a system group - i.e. the new group id will be taken from the system group id list.

password The password for the group, if required

2.10 Service

This represents service startup and shutdown via an init daemon.

The available parameters are:

name A unique name representing an initd service. This would normally match the name as it appears in /etc/init.d.

priority Priority of the service within the boot order. This attribute will have no effect when using a dependency or event based init.d subsystem like upstart or systemd.

start A command that when executed will start the service. If not provided, the provider will use the default service start invocation for the init.d system in use.

stop A command that when executed will start the service. If not provided, the provider will use the default service stop invocation for the init.d system in use.

restart A command that when executed will restart the service. If not provided, the provider will use the default service restart invocation for the init.d system in use. If it is not possible to automatically determine if the restart script is avilable the service will be stopped and started instead.

reconfig A command that when executed will make the service reload its configuration file.

running A comamnd to execute to determine if a service is running. Should have an exit code of 0 for success.

pidfile Where the service creates its pid file. This can be provided instead of `running` as an alternative way of checking if a service is running or not.

CHAPTER 3

Contrib

The `fabric.contrib` module contains examples showing how to integrate fuselage with other tools and libraries such as fabric and libcloud.

3.1 Fabric

The `fuselage` module comes with fabric integration. This is simple a decorator that allows you to simple `yield` resources and apply them to any hosts fabric can connect to.

For example in your `fabfile.py` you might write:

```
from fuselage.contrib.fabric import blueprint
from fuselage.resources import *

@blueprint
def deploy(bundle):
    """ Deploy configuration to app server cluster """
    yield File(
        name='/tmp/some-thing'
    )
    yield Package(name="apache2")
```

This task will show up in your fabric task list like any other:

```
# fab -l
Available commands:

    deploy  Deploy configuration to app server cluster
```

You can run this against multiple computers:

```
# fab -H server1,server2 app_server
```

3.2 Libcloud

The libcloud compute API exposes simple deployment functionality via the `deploy_node` method. This method starts a compute node and runs one or more `Deployment` objects against it. The fuselage contrib module provides `FuselageDeployment`.

To create a node at Brightbox and deploy Apache on it you could write something like this:

```
from fuselage.contrib.libcloud import FuselageDeployment
from fuselage.resources import *

from libcloud.compute.types import Provider
from libcloud.compute.providers import get_driver

Driver = get_driver(Provider.BRIGHTBOX)
driver = Driver('my-client-id', 'my-client-secret')

images = conn.list_images()
sizes = conn.list_sizes()

step = FuselageDeployment(resources=[
    Package(name='apache2'),
    File(
        name='/etc/apache2/sites-enabled/default',
        contents='<VirtualHost>...snip...',
    ),
    Execute(
        name='restart-apache',
        command='apache2ctl graceful',
        watches=['/etc/apache2/sites-enabled/default']
    ),
])

node = conn.deploy_node(
    name='test',
    image=images[0],
    size=sizes[0],
    deploy=step
)
```

3.3 Vagrant

The easiest way to use fuselage with vagrant is via the *vagrant-carpet <https://github.com/mover-io/vagrant-carpet>_*.

You will need to install Virtualbox and Vagrant. Then you can install the `vagrant-carpet` plugin:

```
$ vagrant-plugin install vagrant-carpet
Installing the 'vagrant-carpet' plugin. This can take a few minutes...
Installed the plugin 'vagrant-carpet (0.3.0)'!
```

You can set up a `Vagrantfile` in your project that looks like this:

```
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
    config.vm.box = "hashicorp/precise64"
    config.vm.provision :fabric do |fabric|
        fabric.fabfile_path = "./fabfile.py"
        fabric.tasks = ["deploy", ]
    end
end
```

You can then run `vagrant up` to spin up a new Ubuntu Precise VM and run your fuselage enabled `deploy` task.

CHAPTER 4

Contributing

To contribute to Fuselage send us pull requests on GitHub!

If you have any questions we are in `#yaybu` on `irc.oftc.net`.