

---

# **Ansible 2.2 Documentation**

**2.6**

**Ansible, Inc**

**10, 2018**



<b>1</b>	<b>About Ansible</b>	<b>1</b>
1.1	Installation Guide . . . . .	1
1.2	Configuring Ansible . . . . .	8
1.3	Ansible Porting Guides . . . . .	9
1.4	User Guide . . . . .	42
1.5	Ansible Community Guide . . . . .	270
1.6	Developer Guide . . . . .	285
1.7	Cisco ACI Guide . . . . .	354
1.8	Amazon Web Services Guide . . . . .	363
1.9	Microsoft Azure Guide . . . . .	367
1.10	CloudStack Cloud Guide . . . . .	375
1.11	Getting Started with Docker . . . . .	382
1.12	Google Cloud Platform Guide . . . . .	386
1.13	Using Ansible with the Packet host . . . . .	390
1.14	Rackspace Cloud Guide . . . . .	396
1.15	Continuous Delivery and Rolling Upgrades . . . . .	410
1.16	Using Vagrant and Ansible . . . . .	416
1.17	Getting Started with VMware . . . . .	418
1.18	Ansible for VMware . . . . .	419
1.19	Ansible for Network Automation . . . . .	428
1.20	Ansible Galaxy . . . . .	473
1.21	YAML Syntax . . . . .	482
1.22	Python 3 Support . . . . .	486
1.23	Testing Strategies . . . . .	488
1.24	Frequently Asked Questions . . . . .	492
1.25	Glossary . . . . .	501
1.26	Ansible Tower . . . . .	506
1.27	Ansible Roadmap . . . . .	506



# CHAPTER 1

---

## About Ansible

---

Ansible is an IT automation tool. It can configure systems, deploy software, and orchestrate more advanced IT tasks such as continuous deployments or zero downtime rolling updates.

Ansible's main goals are simplicity and ease-of-use. It also has a strong focus on security and reliability, featuring a minimum of moving parts, usage of OpenSSH for transport (with other transports and pull modes as alternatives), and a language that is designed around auditability by humans—even those not familiar with the program.

We believe simplicity is relevant to all sizes of environments, so we design for busy users of all types: developers, sysadmins, release engineers, IT managers, and everyone in between. Ansible is appropriate for managing all environments, from small setups with a handful of instances to enterprise environments with many thousands of instances.

Ansible manages machines in an agent-less manner. There is never a question of how to upgrade remote daemons or the problem of not being able to manage systems because daemons are uninstalled. Because OpenSSH is one of the most peer-reviewed open source components, security exposure is greatly reduced. Ansible is decentralized—it relies on your existing OS credentials to control access to remote machines. If needed, Ansible can easily connect with Kerberos, LDAP, and other centralized authentication management systems.

This documentation covers the current released version of Ansible (2.5) and also some development version features. For recent features, we note in each section the version of Ansible where the feature was added.

Ansible releases a new major release of Ansible approximately every two months. The core application evolves somewhat conservatively, valuing simplicity in language design and setup. However, the community around new modules and plugins being developed and contributed moves very quickly, adding many new modules in each release.

Welcome to the Ansible Installation Guide!

Introductory install guide text here.

## 1.1 Installation Guide

<b>Topics</b>
---------------

- *Installation Guide*
  - *Basics / What Will Be Installed*
  - *What Version To Pick?*
  - *Control Machine Requirements*
  - *Managed Node Requirements*
  - *Installing the Control Machine*
    - \* *Latest Release via DNF or Yum*
    - \* *Latest Releases Via Apt (Ubuntu)*
    - \* *Latest Releases Via Apt (Debian)*
    - \* *Latest Releases Via Portage (Gentoo)*
    - \* *Latest Releases Via pkg (FreeBSD)*
    - \* *Latest Releases on Mac OSX*
    - \* *Latest Releases Via OpenCSW (Solaris)*
    - \* *Latest Releases Via Pacman (Arch Linux)*
    - \* *Latest Releases Via Pip*
    - \* *Tarballs of Tagged Releases*
    - \* *Running From Source*
  - *Ansible on GitHub*

Welcome to the Ansible Installation Guide!

## **1.1.1 Basics / What Will Be Installed**

Ansible by default manages machines over the SSH protocol.

Once Ansible is installed, it will not add a database, and there will be no daemons to start or keep running. You only need to install it on one machine (which could easily be a laptop) and it can manage an entire fleet of remote machines from that central point. When Ansible manages remote machines, it does not leave software installed or running on them, so there's no real question about how to upgrade Ansible when moving to a new version.

## **1.1.2 What Version To Pick?**

Because it runs so easily from source and does not require any installation of software on remote machines, many users will actually track the development version.

Ansible's release cycles are usually about four months long. Due to this short release cycle, minor bugs will generally be fixed in the next release versus maintaining backports on the stable branch. Major bugs will still have maintenance releases when needed, though these are infrequent.

If you are wishing to run the latest released version of Ansible and you are running Red Hat Enterprise Linux (TM), CentOS, Fedora, Debian, or Ubuntu, we recommend using the OS package manager.

For other installation options, we recommend installing via "pip", which is the Python package manager, though other options are also available.

If you wish to track the development release to use and test the latest features, we will share information about running from source. It's not necessary to install the program to run from source.

### 1.1.3 Control Machine Requirements

Currently Ansible can be run from any machine with Python 2 (versions 2.6 or 2.7) or Python 3 (versions 3.5 and higher) installed (Windows isn't supported for the control machine).

This includes Red Hat, Debian, CentOS, macOS, any of the BSDs, and so on.

---

: macOS by default is configured for a small number of file handles, so if you want to use 15 or more forks you'll need to raise the ulimit with `sudo launchctl limit maxfiles unlimited`. This command can also fix any "Too many open files" error.

---

: Please note that some modules and plugins have additional requirements. For modules these need to be satisfied on the 'target' machine and should be listed in the module specific docs.

### 1.1.4 Managed Node Requirements

On the managed nodes, you need a way to communicate, which is normally ssh. By default this uses sftp. If that's not available, you can switch to scp in `ansible.cfg`. You also need Python 2 (version 2.6 or later) or Python 3 (version 3.5 or later).

---

:

- If you have SELinux enabled on remote nodes, you will also want to install `libselinux-python` on them before using any copy/file/template related functions in Ansible. You can use the `yum` module or `dnf` module in Ansible to install this package on remote systems that do not have it.
- By default, Ansible uses the python interpreter located at `/usr/bin/python` to run its modules. However, some Linux distributions may only have a Python 3 interpreter installed to `/usr/bin/python3` by default. On those systems, you may see an error like:

```
"module_stdout": "/bin/sh: /usr/bin/python: No such file or directory\r\n"
```

you can either set the `ansible_python_interpreter` inventory variable (see *Working with Inventory*) to point at your interpreter or you can install a Python 2 interpreter for modules to use. You will still need to set `ansible_python_interpreter` if the Python 2 interpreter is not installed to `/usr/bin/python`.

- Ansible's "raw" module (for executing commands in a quick and dirty way) and the script module don't even need Python installed. So technically, you can use Ansible to install a compatible version of Python using the raw module, which then allows you to use everything else. For example, if you need to bootstrap Python 2 onto a RHEL-based system, you can install it via

```
$ ansible myhost --sudo -m raw -a "yum install -y python2"
```

## 1.1.5 Installing the Control Machine

### Latest Release via DNF or Yum

On Fedora:

```
$ sudo dnf install ansible
```

On RHEL and CentOS:

```
$ sudo yum install ansible
```

RPMs for RHEL 7 are available from the [Ansible Engine repository](#).

To enable the Ansible Engine repository, run the following command:

```
$ sudo subscription-manager repos --enable rhel-7-server-ansible-2.6-rpms
```

RPMs for currently supported versions of RHEL, CentOS, and Fedora are available from [EPEL](#) as well as [releases.ansible.com](#).

Ansible version 2.4 and later can manage earlier operating systems that contain Python 2.6 or higher.

You can also build an RPM yourself. From the root of a checkout or tarball, use the `make rpm` command to build an RPM you can distribute and install.

```
$ git clone https://github.com/ansible/ansible.git
$ cd ./ansible
$ make rpm
$ sudo rpm -Uvh ./rpm-build/ansible-*.noarch.rpm
```

### Latest Releases Via Apt (Ubuntu)

Ubuntu builds are available [in a PPA here](#).

To configure the PPA on your machine and install ansible run these commands:

```
$ sudo apt-get update
$ sudo apt-get install software-properties-common
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install ansible
```

---

: On older Ubuntu distributions, "software-properties-common" is called "python-software-properties".

---

Debian/Ubuntu packages can also be built from the source checkout, run:

```
$ make deb
```

You may also wish to run from source to get the latest, which is covered below.

### Latest Releases Via Apt (Debian)

Debian users may leverage the same source as the Ubuntu PPA.



Add the following line to `/etc/apt/sources.list`:

```
deb http://ppa.launchpad.net/ansible/ansible/ubuntu trusty main
```

Then run these commands:

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 93C4A3FD7BB9C367
$ sudo apt-get update
$ sudo apt-get install ansible
```

: This method has been verified with the Trusty sources in Debian Jessie and Stretch but may not be supported in earlier versions.

### Latest Releases Via Portage (Gentoo)

```
$ emerge -av app-admin/ansible
```

To install the newest version, you may need to unmask the ansible package prior to emerging:

```
$ echo 'app-admin/ansible' >> /etc/portage/package.accept_keywords
```

: The current default Python slot on Gentoo is version 3.4. Ansible needs Python-3.5 or higher so you will need to *ref: 'bootstrap <managed\_node\_requirements>* a compatible version onto the machines.

### Latest Releases Via pkg (FreeBSD)

```
$ sudo pkg install ansible
```

You may also wish to install from ports, run:

```
$ sudo make -C /usr/ports/sysutils/ansible install
```

### Latest Releases on Mac OSX

The preferred way to install Ansible on a Mac is via pip.

The instructions can be found in [Latest Releases Via Pip](#) section. If you are running macOS version 10.12 or older, then you ought to upgrade to the latest pip (9.0.3 or newer) to connect to the Python Package Index securely.

### Latest Releases Via OpenCSW (Solaris)

Ansible is available for Solaris as SysV package from OpenCSW.

```
# pkgadd -d http://get.opencsw.org/now
# /opt/csw/bin/pkgutil -i ansible
```

### Latest Releases Via Pacman (Arch Linux)

Ansible is available in the Community repository:

```
$ pacman -S ansible
```

The AUR has a PKGBUILD for pulling directly from Github called [ansible-git](#).

Also see the [Ansible](#) page on the ArchWiki.

### Latest Releases Via Pip

Ansible can be installed via "pip", the Python package manager. If 'pip' isn't already available in your version of Python, you can get pip by:

```
$ sudo easy_install pip
```

Then install Ansible with<sup>1</sup>:

```
$ sudo pip install ansible
```

Or if you are looking for the latest development version:

```
$ pip install git+https://github.com/ansible/ansible.git@devel
```

If you are installing on macOS Mavericks, you may encounter some noise from your compiler. A workaround is to do the following:

```
$ sudo CFLAGS=-Qunused-arguments CPPFLAGS=-Qunused-arguments pip install ansible
```

Readers that use virtualenv can also install Ansible under virtualenv, though we'd recommend to not worry about it and just install Ansible globally. Do not use easy\_install to install Ansible directly.

---

: Older versions of pip defaults to <http://pypi.python.org/simple>, which no longer works. Please make sure you have an updated pip (version 10 or greater) installed before installing Ansible. Refer [here](#) about installing latest pip.

---

### Tarballs of Tagged Releases

Packaging Ansible or wanting to build a local package yourself, but don't want to do a git checkout? Tarballs of releases are available on the [Ansible downloads](#) page.

These releases are also tagged in the [git repository](#) with the release version.

### Running From Source

Ansible is easy to run from a checkout - root permissions are not required to use it and there is no software to actually install. No daemons or database setup are required. Because of this, many users in our community use the development version of Ansible all of the time so they can take advantage of new features when they are implemented and easily contribute to the project. Because there is nothing to install, following the development version is significantly easier than most open source projects.

---

<sup>1</sup> If you have issues with the "pycrypto" package install on Mac OSX, then you may need to try `CC=clang sudo -E pip install pycrypto`.

---

: If you are intending to use Tower as the Control Machine, do not use a source install. Please use OS package manager (like `apt/yum`) or `pip` to install a stable version.

---

To install from source, clone the Ansible git repository:

```
$ git clone https://github.com/ansible/ansible.git --recursive
$ cd ./ansible
```

Once git has cloned the Ansible repository, setup the Ansible environment:

Using Bash:

```
$ source ./hacking/env-setup
```

Using Fish:

```
$ source ./hacking/env-setup.fish
```

If you want to suppress spurious warnings/errors, use:

```
$ source ./hacking/env-setup -q
```

If you don't have pip installed in your version of Python, install pip:

```
$ sudo easy_install pip
```

Ansible also uses the following Python modules that need to be installed<sup>1</sup>:

```
$ sudo pip install -r ./requirements.txt
```

To update ansible checkouts, use pull-with-rebase so any local changes are replayed.

```
$ git pull --rebase
```

Note: when updating Ansible checkouts that are v2.2 and older, be sure to not only update the source tree, but also the "submodules" in git which point at Ansible's own modules.

```
$ git pull --rebase #same as above
$ git submodule update --init --recursive
```

Once running the `env-setup` script you'll be running from checkout and the default inventory file will be `/etc/ansible/hosts`. You can optionally specify an inventory file (see [Working with Inventory](#)) other than `/etc/ansible/hosts`:

```
$ echo "127.0.0.1" > ~/ansible_hosts
$ export ANSIBLE_INVENTORY=~/ansible_hosts
```

---

: `ANSIBLE_INVENTORY` is available starting at 1.9 and substitutes the deprecated `ANSIBLE_HOSTS`

---

You can read more about the inventory file in later parts of the manual.

Now let's test things with a ping command:

```
$ ansible all -m ping --ask-pass
```

You can also use "sudo make install".

## 1.1.6 Ansible on GitHub

You may also wish to follow the [GitHub project](#) if you have a GitHub account. This is also where we keep the issue tracker for sharing bugs and feature ideas.

:

*Introduction To Ad-Hoc Commands* Examples of basic commands

*Working With Playbooks* Learning ansible's configuration management language

*How do I handle the package dependencies required by Ansible package dependencies during Ansible installation ?*  
Ansible Installation related to FAQs

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**irc.freenode.net** #ansible IRC chat channel

## 1.2 Configuring Ansible

### Topics

- *Configuring Ansible*
  - *Configuration file*
    - \* *Getting the latest configuration*
  - *Environmental configuration*
  - *Command line options*

This topic describes how to control Ansible settings.

### 1.2.1 Configuration file

Certain settings in Ansible are adjustable via a configuration file (`ansible.cfg`). The stock configuration should be sufficient for most users, but there may be reasons you would want to change them. Paths where configuration file is searched are listed in reference documentation.

#### Getting the latest configuration

If installing Ansible from a package manager, the latest `ansible.cfg` file should be present in `/etc/ansible`, possibly as a `".rpmnew"` file (or other) as appropriate in the case of updates.

If you installed Ansible from pip or from source, you may want to create this file in order to override default settings in Ansible.

An [example file](#) is available on Github.

For more details and a full listing of available configurations go to `configuration_settings`. Starting with Ansible version 2.4, you can use the `ansible-config` command line utility to list your available options and inspect the current values.

For in-depth details, see `ansible_configuration_settings`.

## 1.2.2 Environmental configuration

Ansible also allows configuration of settings using environment variables. If these environment variables are set, they will override any setting loaded from the configuration file.

You can get a full listing of available environment variables from `ansible_configuration_settings`.

## 1.2.3 Command line options

Not all configuration options are present in the command line, just the ones deemed most useful or common. Settings in the command line will override those passed through the configuration file and the environment.

The full list of options available is in `ansible-playbook` and `ansible`.

## 1.3 Ansible Porting Guides

This section lists porting guides that can help you in updating playbooks, plugins and other parts of your Ansible infrastructure from one version of Ansible to the next.

Please note that this is not a complete list. If you believe any extra information would be useful in these pages, you can edit by clicking *Edit on GitHub* on the top right, or raising an issue.

### 1.3.1 Ansible 2.0 Porting Guide

This section discusses the behavioral changes between Ansible 1.x and Ansible 2.0.

It is intended to assist in updating your playbooks, plugins and other parts of your Ansible infrastructure so they will work with this version of Ansible.

We suggest you read this page along with [Ansible Changelog for 2.0](#) to understand what updates you may need to make.

This document is part of a collection on porting. The complete list of porting guides can be found at [porting guides](#).

#### Topics

- *Ansible 2.0 Porting Guide*
  - *Playbook*
    - \* *Deprecated*
    - \* *Other caveats*
  - *Porting plugins*
    - \* *Lookup plugins*
    - \* *Connection plugins*
    - \* *Action plugins*
    - \* *Callback plugins*
    - \* *Connection plugins*
  - *Hybrid plugins*

- \* *Lookup plugins*
- \* *Connection plugins*
- \* *Action plugins*
- \* *Callback plugins*
- \* *Connection plugins*
- *Porting custom scripts*

## Playbook

This section discusses any changes you may need to make to your playbooks.

```
# Syntax in 1.9.x
- debug:
    msg: "{{ 'test1_junk 1\\\\\\3' | regex_replace('(.*?)_junk (.*?)', '\\\\1 \\\2') }}"
# Syntax in 2.0.x
- debug:
    msg: "{{ 'test1_junk 1\\\\3' | regex_replace('(.*?)_junk (.*?)', '\\1 \2') }}"

# Output:
"msg": "test1 1\\\\3"
```

To make an escaped string that will work on all versions you have two options:

```
- debug: msg="{{ 'test1_junk 1\\\\3' | regex_replace('(.*?)_junk (.*?)', '\\1 \2') }}"
```

uses key=value escaping which has not changed. The other option is to check for the ansible version:

```
"{{ (ansible_version|version_compare('2.0', 'ge'))|ternary( 'test1_junk 1\\\\3' | regex_
↪replace('(.*?)_junk (.*?)', '\\1 \2') , 'test1_junk 1\\\\\\\\3' | regex_replace('(.*?)_
↪junk (.*?)', '\\\\\\1 \\\2') ) }}"
```

- **trailing newline** When a string with a trailing newline was specified in the playbook via yaml dict format, the trailing newline was stripped. When specified in key=value format, the trailing newlines were kept. In v2, both methods of specifying the string will keep the trailing newlines. If you relied on the trailing newline being stripped, you can change your playbook using the following as an example:

```
# Syntax in 1.9.x
vars:
  message: >
    Testing
    some things
tasks:
- debug:
    msg: "{{ message }}"

# Syntax in 2.0.x
vars:
  old_message: >
    Testing
    some things
  message: "{{ old_message[:-1] }}"
- debug:
```

(continues on next page)

()

```
msg: "{{ message }}"
# Output
"msg": "Testing some things"
```

- Behavior of templating DOS-type text files changes with Ansible v2.

A bug in Ansible v1 causes DOS-type text files (using a carriage return and newline) to be templated to Unix-type text files (using only a newline). In Ansible v2 this long-standing bug was finally fixed and DOS-type text files are preserved correctly. This may be confusing when you expect your playbook to not show any differences when migrating to Ansible v2, while in fact you will see every DOS-type file being completely replaced (with what appears to be the exact same content).

- When specifying complex args as a variable, the variable must use the full jinja2 variable syntax (`{{var_name}}`) - bare variable names there are no longer accepted. In fact, even specifying args with variables has been deprecated, and will not be allowed in future versions:

```
---
- hosts: localhost
  connection: local
  gather_facts: false
  vars:
    my_dirs:
      - { path: /tmp/3a, state: directory, mode: 0755 }
      - { path: /tmp/3b, state: directory, mode: 0700 }
  tasks:
    - file:
      args: "{{item}}" # <- args here uses the full variable syntax
      with_items: "{{my_dirs}}"
```

- porting task includes
- More dynamic. Corner-case formats that were not supposed to work now do not, as expected.
- variables defined in the yaml dict format <https://github.com/ansible/ansible/issues/13324>
- templating (variables in playbooks and template lookups) has improved with regard to keeping the original instead of turning everything into a string. If you need the old behavior, quote the value to pass it around as a string.
- Empty variables and variables set to null in yaml are no longer converted to empty strings. They will retain the value of *None*. You can override the *null\_representation* setting to an empty string in your config file by setting the `ANSIBLE_NULL_REPRESENTATION` environment variable.
- Extras callbacks must be whitelisted in `ansible.cfg`. Copying is no longer necessary but whitelisting in `ansible.cfg` must be completed.
- dnf module has been rewritten. Some minor changes in behavior may be observed.
- win\_updates has been rewritten and works as expected now.
- from 2.0.1 onwards, the implicit setup task from `gather_facts` now correctly inherits everything from `play`, but this might cause issues for those setting *environment* at the play level and depending on *ansible\_env* existing. Previously this was ignored but now might issue an 'Undefined' error.

## Deprecated

While all items listed here will show a deprecation warning message, they still work as they did in 1.9.x. Please note that they will be removed in 2.2 (Ansible always waits two major releases to remove a deprecated feature).

- Bare variables in `with_` loops should instead use the `"{{ var }}"` syntax, which helps eliminate ambiguity.
- The `ansible-galaxy` text format requirements file. Users should use the YAML format for requirements instead.
- Undefined variables within a `with_` loop's list currently do not interrupt the loop, but they do issue a warning; in the future, they will issue an error.
- Using dictionary variables to set all task parameters is unsafe and will be removed in a future version. For example:

```
- hosts: localhost
  gather_facts: no
  vars:
    debug_params:
      msg: "hello there"
  tasks:
    # These are both deprecated:
    - debug: "{{debug_params}}"
    - debug:
      args: "{{debug_params}}"

    # Use this instead:
    - debug:
      msg: "{{debug_params['msg'] }}"
```

- Host patterns should use a comma (,) or colon (:) instead of a semicolon (;) to separate hosts/groups in the pattern.
- Ranges specified in host patterns should use the `[x:y]` syntax, instead of `[x-y]`.
- Playbooks using privilege escalation should always use `"become*"` options rather than the old `su*/sudo*` options.
- The "short form" for `vars_prompt` is no longer supported. For example:

```
vars_prompt:
  variable_name: "Prompt string"
```

- Specifying variables at the top level of a task include statement is no longer supported. For example:

```
- include_tasks: foo.yml
  a: 1
```

Should now be:

```
- include_tasks: foo.yml
  vars:
    a: 1
```

- Setting `any_errors_fatal` on a task is no longer supported. This should be set at the play level only.
- Bare variables in the `environment` dictionary (for plays/tasks/etc.) are no longer supported. Variables specified there should use the full variable syntax: `'{{foo}}'`.
- Tags (or any directive) should no longer be specified with other parameters in a task include. Instead, they should be specified as an option on the task. For example:

```
- include_tasks: foo.yml tags=a,b,c
```

Should be:



```
- include_tasks: foo.yml
  tags: [a, b, c]
```

- The `first_available_file` option on tasks has been deprecated. Users should use the `with_first_found` option or `lookup ('first_found', ...)` plugin.

## Other caveats

Here are some corner cases encountered when updating. These are mostly caused by the more stringent parser validation and the capture of errors that were previously ignored.

- Bad variable composition:

```
with_items: myvar_{{rest_of_name}}
```

This worked 'by accident' as the errors were retemplated and ended up resolving the variable, it was never intended as valid syntax and now properly returns an error, use the following instead.:

```
hostvars[inventory_hostname]['myvar_' + rest_of_name]
```

- Misspelled directives:

```
- task: dostuf
  becom: yes
```

The task always ran without using privilege escalation (for that you need *become*) but was also silently ignored so the play 'ran' even though it should not, now this is a parsing error.

- Duplicate directives:

```
- task: dostuf
  when: True
  when: False
```

The first *when* was ignored and only the 2nd one was used as the play ran w/o warning it was ignoring one of the directives, now this produces a parsing error.

- Conflating variables and directives:

```
- role: {name=rosy, port=435 }

# in tasks/main.yml
- wait_for: port={{port}}
```

The *port* variable is reserved as a play/task directive for overriding the connection port, in previous versions this got conflated with a variable named *port* and was usable later in the play, this created issues if a host tried to reconnect or was using a non caching connection. Now it will be correctly identified as a directive and the *port* variable will appear as undefined, this now forces the use of non conflicting names and removes ambiguity when adding settings and variables to a role invocation.

- Bare operations on *with\_*:

```
with_items: var1 + var2
```

An issue with the 'bare variable' features, which was supposed only template a single variable without the need of braces (`{{ }}`), would in some versions of Ansible template full expressions. Now you need to use proper templating and braces for all expressions everywhere except conditionals (*when*):

```
with_items: "{{var1 + var2}}"
```

The bare feature itself is deprecated as an undefined variable is indistinguishable from a string which makes it difficult to display a proper error.

## Porting plugins

In ansible-1.9.x, you would generally copy an existing plugin to create a new one. Simply implementing the methods and attributes that the caller of the plugin expected made it a plugin of that type. In ansible-2.0, most plugins are implemented by subclassing a base class for each plugin type. This way the custom plugin does not need to contain methods which are not customized.

## Lookup plugins

- lookup plugins ; import version

## Connection plugins

- connection plugins

## Action plugins

- action plugins

## Callback plugins

Although Ansible 2.0 provides a new callback API the old one continues to work for most callback plugins. However, if your callback plugin makes use of `self.playbook`, `self.play`, or `self.task` then you will have to store the values for these yourself as ansible no longer automatically populates the callback with them. Here's a short snippet that shows you how:

```
import os
from ansible.plugins.callback import CallbackBase

class CallbackModule(CallbackBase):
    def __init__(self):
        self.playbook = None
        self.playbook_name = None
        self.play = None
        self.task = None

    def v2_playbook_on_start(self, playbook):
        self.playbook = playbook
        self.playbook_name = os.path.basename(self.playbook._file_name)

    def v2_playbook_on_play_start(self, play):
        self.play = play

    def v2_playbook_on_task_start(self, task, is_conditional):
        self.task = task
```

(continues on next page)

()

```
def v2_on_any(self, *args, **kwargs):
    self._display.display('%s: %s: %s' % (self.playbook_name,
    self.play.name, self.task))
```

## Connection plugins

- connection plugins

## Hybrid plugins

In specific cases you may want a plugin that supports both ansible-1.9.x *and* ansible-2.0. Much like porting plugins from v1 to v2, you need to understand how plugins work in each version and support both requirements.

Since the ansible-2.0 plugin system is more advanced, it is easier to adapt your plugin to provide similar pieces (subclasses, methods) for ansible-1.9.x as ansible-2.0 expects. This way your code will look a lot cleaner.

You may find the following tips useful:

- Check whether the ansible-2.0 class(es) are available and if they are missing (ansible-1.9.x) mimic them with the needed methods (e.g. `__init__`)
- When ansible-2.0 python modules are imported, and they fail (ansible-1.9.x), catch the `ImportError` exception and perform the equivalent imports for ansible-1.9.x. With possible translations (e.g. importing specific methods).
- Use the existence of these methods as a qualifier to what version of Ansible you are running. So rather than using version checks, you can do capability checks instead. (See examples below)
- Document for each if-then-else case for which specific version each block is needed. This will help others to understand how they have to adapt their plugins, but it will also help you to remove the older ansible-1.9.x support when it is deprecated.
- When doing plugin development, it is very useful to have the `warning()` method during development, but it is also important to emit warnings for deadends (cases that you expect should never be triggered) or corner cases (e.g. cases where you expect misconfigurations).
- It helps to look at other plugins in ansible-1.9.x and ansible-2.0 to understand how the API works and what modules, classes and methods are available.

## Lookup plugins

As a simple example we are going to make a hybrid `fileglob` lookup plugin.

```
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type

import os
import glob

try:
    # ansible-2.0
    from ansible.plugins.lookup import LookupBase
except ImportError:
```

(continues on next page)

```

# ansible-1.9.x

class LookupBase(object):
    def __init__(self, basedir=None, runner=None, **kwargs):
        self.runner = runner
        self.basedir = self.runner.basedir

    def get_basedir(self, variables):
        return self.basedir

try:
    # ansible-1.9.x
    from ansible.utils import (listify_lookup_plugin_terms, path_dwim, warning)
except ImportError:
    # ansible-2.0
    from __main__ import display
    warning = display.warning

class LookupModule(LookupBase):

    # For ansible-1.9.x, we added inject=None as valid argument
    def run(self, terms, inject=None, variables=None, **kwargs):

        # ansible-2.0, but we made this work for ansible-1.9.x too !
        basedir = self.get_basedir(variables)

        # ansible-1.9.x
        if 'listify_lookup_plugin_terms' in globals():
            terms = listify_lookup_plugin_terms(terms, basedir, inject)

        ret = []
        for term in terms:
            term_file = os.path.basename(term)

            # For ansible-1.9.x, we imported path_dwim() from ansible.utils
            if 'path_dwim' in globals():
                # ansible-1.9.x
                dwimmed_path = path_dwim(basedir, os.path.dirname(term))
            else:
                # ansible-2.0
                dwimmed_path = self._loader.path_dwim_relative(basedir, 'files', os.
→path.dirname(term))

            globbed = glob.glob(os.path.join(dwimmed_path, term_file))
            ret.extend(g for g in globbed if os.path.isfile(g))

        return ret

```

: In the above example we did not use the `warning()` method as we had no direct use for it in the final version. However we left this code in so people can use this part during development/porting/use.

## Connection plugins

- connection plugins

## Action plugins

- action plugins

## Callback plugins

- callback plugins

## Connection plugins

- connection plugins

## Porting custom scripts

Custom scripts that used the `ansible.runner.Runner` API in 1.x have to be ported in 2.x. Please refer to: [Python API](#)

## 1.3.2 Ansible 2.3 Porting Guide

This section discusses the behavioral changes between Ansible 2.2 and Ansible 2.3.

It is intended to assist in updating your playbooks, plugins and other parts of your Ansible infrastructure so they will work with this version of Ansible.

We suggest you read this page along with [Ansible Changelog for 2.3](#) to understand what updates you may need to make.

This document is part of a collection on porting. The complete list of porting guides can be found at [porting guides](#).

### Topics

- *Ansible 2.3 Porting Guide*
  - *Playbook*
    - \* *Restructured async to work with action plugins*
    - \* *OpenBSD version facts*
    - \* *Names Blocks*
    - \* *Use of multiple tags*
    - \* *Other caveats*
  - *Modules*
    - \* *Modules removed*
    - \* *Deprecation notices*
    - \* *Noteworthy module changes*
      - *AWS lambda*
      - *Mount*

- *Plugins*
- *Porting custom scripts*
- *Networking*
  - \* *Deprecation of top-level connection arguments*
  - \* *delegate\_to vs ProxyCommand*

## Playbook

### Restructured async to work with action plugins

In Ansible 2.2 (and possibly earlier) the *async*: keyword could not be used in conjunction with the action plugins such as *service*. This limitation has been removed in Ansible 2.3

**NEW** In Ansible 2.3:

```
- name: Install nginx asynchronously
  service:
    name: nginx
    state: restarted
  async: 45
```

### OpenBSD version facts

The *ansible\_distribution\_release* and *ansible\_distribution\_version* facts on OpenBSD hosts were reversed in Ansible 2.2 and earlier. This has been changed so that version has the numeric portion and release has the name of the release.

**OLD** In Ansible 2.2 (and earlier)

```
"ansible_distribution": "OpenBSD"
"ansible_distribution_release": "6.0",
"ansible_distribution_version": "release",
```

**NEW** In Ansible 2.3:

```
"ansible_distribution": "OpenBSD",
"ansible_distribution_release": "release",
"ansible_distribution_version": "6.0",
```

## Names Blocks

Blocks can now have names, this allows you to avoid the ugly *# this block is for...* comments.

**NEW** In Ansible 2.3:

```
- name: Block test case
  hosts: localhost
  tasks:
    - name: Attempt to setup foo
      block:
        - debug: msg='I execute normally'
```

(continues on next page)

()

```

- command: /bin/false
- debug: msg='I never execute, cause ERROR!'
rescue:
- debug: msg='I caught an error'
- command: /bin/false
- debug: msg='I also never execute :-( '
always:
- debug: msg="this always executes"

```

## Use of multiple tags

Specifying `--tags` (or `--skip-tags`) multiple times on the command line currently leads to the last specified tag overriding all the other specified tags. This behaviour is deprecated. In the future, if you specify `--tags` multiple times the tags will be merged together. From now on, using `--tags` multiple times on one command line will emit a deprecation warning. Setting the `merge_multiple_cli_tags` option to `True` in the `ansible.cfg` file will enable the new behaviour.

In 2.4, the default will be to merge the tags. You can enable the old overwriting behavior via the config option. In 2.5, multiple `--tags` options will be merged with no way to go back to the old behaviour.

## Other caveats

Here are some rare cases that might be encountered when updating. These are mostly caused by the more stringent parser validation and the capture of errors that were previously ignored.

- Made `any_errors_fatal` inheritable from play to task and all other objects in between.

## Modules

No major changes in this version.

## Modules removed

No major changes in this version.

## Deprecation notices

The following modules will be removed in Ansible 2.5. Please update your playbooks accordingly.

- `ec2_vpc`
- `cl_bond`
- `cl_bridge`
- `cl_img_install`
- `cl_interface`
- `cl_interface_policy`
- `cl_license`

- `cl_ports`
- `nxos_mtu` use `nxos_system` instead

### Noteworthy module changes

#### AWS lambda

Previously ignored changes that only affected one parameter. Existing deployments may have outstanding changes that this bug fix will apply.

#### Mount

Mount: Some fixes so bind mounts are not mounted each time the playbook runs.

#### Plugins

No major changes in this version.

#### Porting custom scripts

No major changes in this version.

#### Networking

There have been a number of changes to number of changes to how Networking Modules operate.

Playbooks should still use `connection: local`.

The following changes apply to:

- `dellos6`
- `dellos9`
- `dellos10`
- `eos`
- `ios`
- `iosxr`
- `junos`
- `sros`
- `vyos`

#### Deprecation of top-level connection arguments

**OLD** In Ansible 2.2:



```
- name: example of using top-level options for connection properties
  ios_command:
    commands: show version
    host: "{{ inventory_hostname }}"
    username: cisco
    password: cisco
    authorize: yes
    auth_pass: cisco
```

Will result in:

```
[WARNING]: argument username has been deprecated and will be removed in a future_
↪version
[WARNING]: argument host has been deprecated and will be removed in a future version
[WARNING]: argument password has been deprecated and will be removed in a future_
↪version
```

**NEW** In Ansible 2.3:

```
- name: Gather facts
  eos_facts:
    gather_subset: all
    provider:
      username: myuser
      password: "{{ networkpassword }}"
      transport: cli
      host: "{{ ansible_host }}"
```

## delegate\_to vs ProxyCommand

The new connection framework for Network Modules in Ansible 2.3 that uses `cli` transport no longer supports the use of the `delegate_to` directive. In order to use a bastion or intermediate jump host to connect to network devices over `cli` transport, network modules now support the use of `ProxyCommand`.

To use `ProxyCommand` configure the proxy settings in the Ansible inventory file to specify the proxy host via `ansible_ssh_common_args`.

For details on how to do this see the [network proxy guide](#).

## 1.3.3 Ansible 2.4 Porting Guide

This section discusses the behavioral changes between Ansible 2.3 and Ansible 2.4.

It is intended to assist in updating your playbooks, plugins and other parts of your Ansible infrastructure so they will work with this version of Ansible.

We suggest you read this page along with [Ansible Changelog for 2.4](#) to understand what updates you may need to make.

This document is part of a collection on porting. The complete list of porting guides can be found at [porting guides](#).

### Topics

- [Ansible 2.4 Porting Guide](#)

- *Python version*
- *Inventory*
  - \* *Initial playbook relative group\_vars and host\_vars*
- *Deprecated*
  - \* *Specifying Inventory sources*
  - \* *Use of multiple tags*
  - \* *Other caveats*
- *Modules*
  - \* *Modules removed*
  - \* *Deprecation notices*
  - \* *Noteworthy module changes*
- *Plugins*
  - \* *Vars plugin changes*
  - \* *Inventory plugins*
  - \* *Callback plugins*
  - \* *Template lookup plugin: Escaping Strings*
- *Tests*
  - \* *Tests succeeded/failed*
- *Networking*
  - \* *Persistent Connection*
- *Configuration*

## Python version

Ansible will not support Python 2.4 or 2.5 on the target hosts anymore. Going forward, Python 2.6+ will be required on targets, as already is the case on the controller.

## Inventory

Inventory has been refactored to be implemented via plugins and now allows for multiple sources. This change is mostly transparent to users.

One exception is the `inventory_dir`, which is now a host variable; previously it could only have one value so it was set globally. This means you can no longer use it early in plays to determine `hosts`: or similar keywords. This also changes the behaviour of `add_hosts` and the implicit `localhost`; because they no longer automatically inherit the global value, they default to `None`. See the module documentation for more information.

The `inventory_file` remains mostly unchanged, as it was always host specific.

Since there is no longer a single inventory, the 'implicit `localhost`' doesn't get either of these variables defined.

A bug was fixed with the inventory path/directory, which was defaulting to the current working directory. This caused `group_vars` and `host_vars` to be picked up from the current working directory instead of just adjacent to the

playbook or inventory directory when a host list (comma separated host names) was provided as inventory.

### Initial playbook relative `group_vars` and `host_vars`

In Ansible versions prior to 2.4, the inventory system would maintain the context of the initial playbook that was executed. This allowed successively included playbooks from other directories to inherit `group_vars` and `host_vars` placed relative to the top level playbook file.

Due to some behavioral inconsistencies, this functionality will not be included in the new inventory system starting with Ansible version 2.4.

Similar functionality can still be achieved by using `vars_files`, `include_vars`, or `group_vars` and `host_vars` placed relative to the inventory file.

### Deprecated

#### Specifying Inventory sources

Use of `--inventory-file` on the command line is now deprecated. Use `--inventory` or `-i`. The associated ini configuration key, `hostfile`, and environment variable, `ANSIBLE_HOSTS`, are also deprecated. Replace them with the configuration key `inventory` and environment variable `ANSIBLE_INVENTORY`.

#### Use of multiple tags

Specifying `--tags` (or `--skip-tags`) multiple times on the command line currently leads to the last one overriding all the previous ones. This behavior is deprecated. In the future, if you specify `--tags` multiple times the tags will be merged together. From now on, using `--tags` multiple times on one command line will emit a deprecation warning. Setting the `merge_multiple_cli_tags` option to `True` in the `ansible.cfg` file will enable the new behavior.

In 2.4, the default has change to merge the tags. You can enable the old overwriting behavior via the config option.

In 2.5, multiple `--tags` options will be merged with no way to go back to the old behavior.

### Other caveats

No major changes in this version.

### Modules

Major changes in popular modules are detailed here

- The `win_shell` and `win_command` modules now properly preserve quoted arguments in the command-line. Tasks that attempted to work around the issue by adding extra quotes/escaping may need to be reworked to remove the superfluous escaping. See [Issue 23019](#) for additional detail.

### Modules removed

The following modules no longer exist:

- None

### Deprecation notices

The following modules will be removed in Ansible 2.8. Please update your playbooks accordingly.

- `azure`, use `azure_rm_virtualmachine`, which uses the new Resource Manager SDK.
- `win_msi`, use `win_package` instead

### Noteworthy module changes

- The `win_get_url` module has the dictionary `win_get_url` in its results deprecated, its content is now also available directly in the resulting output, like other modules. This dictionary will be removed in Ansible 2.8.
- The `win_unzip` module no longer includes the dictionary `win_unzip` in its results; the contents are now included directly in the resulting output, like other modules.
- The `win_package` module return values `exit_code` and `restart_required` have been deprecated in favour of `rc` and `reboot_required` respectively. The deprecated return values will be removed in Ansible 2.6.

### Plugins

A new way to configure and document plugins has been introduced. This does not require changes to existing setups but developers should start adapting to the new infrastructure now. More details will be available in the developer documentation for each plugin type.

### Vars plugin changes

There have been many changes to the implementation of vars plugins, but both users and developers should not need to change anything to keep current setups working. Developers should consider changing their plugins take advantage of new features.

The most notable difference to users is that vars plugins now get invoked on demand instead of at inventory build time. This should make them more efficient for large inventories, especially when using a subset of the hosts.

---

:

- This also creates a difference with `group/host_vars` when using them adjacent to playbooks. Before, the 'first' playbook loaded determined the variables; now the 'current' playbook does. We are looking to fix this soon, since 'all playbooks' in the path should be considered for variable loading.
  - In 2.4.1 we added a toggle to allow you to control this behaviour, 'top' will be the pre 2.4, 'bottom' will use the current playbook hosting the task and 'all' will use them all from top to bottom.
- 

### Inventory plugins

Developers should start migrating from hardcoded inventory with dynamic inventory scripts to the new Inventory Plugins. The scripts will still work via the `script` inventory plugin but Ansible development efforts will now concentrate on writing plugins rather than enhancing existing scripts.

Both users and developers should look into the new plugins because they are intended to alleviate the need for many of the hacks and workarounds found in the dynamic inventory scripts.

## Callback plugins

Users:

- Callbacks are now using the new configuration system. Users should not need to change anything as the old system still works, but you might see a deprecation notice if any callbacks used are not inheriting from the built in classes. Developers need to update them as stated below.

Developers:

- If your callback does not inherit from `CallbackBase` (directly or indirectly via another callback), it will still work, but issue a deprecation notice. To avoid this and ensure it works in the future change it to inherit from `CallbackBase` so it has the new options handling methods and properties. You can also implement the new options handling methods and properties but that won't automatically inherit changes added in the future. You can look at `CallbackBase` itself and/or `AnsiblePlugin` for details.
- Any callbacks inheriting from other callbacks might need to also be updated to contain the same documented options as the parent or the options won't be available. This is noted in the developer guide.

## Template lookup plugin: Escaping Strings

Prior to Ansible 2.4, backslashes in strings passed to the template lookup plugin would be escaped automatically. In 2.4, users are responsible for escaping backslashes themselves. This change brings the template lookup plugin inline with the template module so that the same backslash escaping rules apply to both.

If you have a template lookup like this:

```
- debug:
  msg: '{{ lookup("template", "template.j2") }}'
```

**OLD** In Ansible 2.3 (and earlier) `template.j2` would look like this:

```
{{ "name surname" | regex_replace("^[^\s]+\s+(.*)", "\1") }}
```

**NEW** In Ansible 2.4 it should be changed to look like this:

```
{{ "name surname" | regex_replace("^[^\\s]+\\s+(.*)", "\\1") }}
```

## Tests

### Tests succeeded/failed

Prior to Ansible version 2.4, a task return code of `rc` would override a return code of `failed`. In version 2.4, both `rc` and `failed` are used to calculate the state of the task. Because of this, test plugins `succeeded/failed` have also been changed. This means that overriding a task failure with `failed_when: no` will result in `succeeded/failed` returning `True/False`. For example:

```
- command: /bin/false
  register: result
  failed_when: no

- debug:
  msg: 'This is printed on 2.3'
  when: result|failed
```

(continues on next page)

```
- debug:
  msg: 'This is printed on 2.4'
  when: result|succeeded

- debug:
  msg: 'This is always printed'
  when: result.rc != 0
```

As we can see from the example above, in Ansible 2.3 `succeeded/failed` only checked the value of `rc`.

## Networking

There have been a number of changes to how Networking Modules operate.

Playbooks should still use `connection: local`.

## Persistent Connection

The configuration variables `connection_retries` and `connect_interval` which were added in Ansible 2.3 are now deprecated. For Ansible 2.4 and later use `connection_retry_timeout`.

To control timeouts use `command_timeout` rather than the previous top level `timeout` variable under `[default]`

See [Ansible Network debug guide](#) for more information.

## Configuration

The configuration system has had some major changes. Users should be unaffected except for the following:

- All relative paths defined are relative to the `ansible.cfg` file itself. Previously they varied by setting. The new behavior should be more predictable.
- A new macro `{{ CWD }}` is available for paths, which will make paths relative to the 'current working directory', this is unsafe but some users really want to rely on this behaviour.

Developers that were working directly with the previous API should revisit their usage as some methods (for example, `get_config`) were kept for backwards compatibility but will warn users that the function has been deprecated.

The new configuration has been designed to minimize the need for code changes in core for new plugins. The plugins just need to document their settings and the configuration system will use the documentation to provide what they need. This is still a work in progress; currently only 'callback' and 'connection' plugins support this. More details will be added to the specific plugin developer guides.

## 1.3.4 Ansible 2.5 Porting Guide

This section discusses the behavioral changes between Ansible 2.4 and Ansible 2.5.

It is intended to assist in updating your playbooks, plugins and other parts of your Ansible infrastructure so they will work with this version of Ansible.

We suggest you read this page along with [Ansible Changelog for 2.5](#) to understand what updates you may need to make.

This document is part of a collection on porting. The complete list of porting guides can be found at [porting guides](#).

## Topics

- *Ansible 2.5 Porting Guide*
  - *Playbook*
    - \* *Dynamic includes and attribute inheritance*
    - \* *Fixed handling of keywords and inline variables*
    - \* *Migrating from with\_X to loop*
    - \* *with\_list*
    - \* *with\_items*
    - \* *with\_indexed\_items*
    - \* *with\_flattened*
    - \* *with\_together*
    - \* *with\_dict*
    - \* *with\_sequence*
    - \* *with\_subelements*
    - \* *with\_nested/with\_cartesian*
    - \* *with\_random\_choice*
  - *Deprecated*
    - \* *Jinja tests used as filters*
  - *Modules*
    - \* *github\_release*
    - \* *Modules removed*
    - \* *Deprecation notices*
    - \* *Noteworthy module changes*
  - *Plugins*
    - \* *Inventory*
    - \* *Shell*
    - \* *Filter*
    - \* *Lookup*
  - *Porting custom scripts*
  - *Network*
    - \* *Expanding documentation*
    - \* *Top-level connection arguments will be removed in 2.9*
    - \* *Adding persistent connection types `network_cli` and `netconf`*
    - \* *Developers: Shared Module Utilities Moved*

### Playbook

#### Dynamic includes and attribute inheritance

In Ansible version 2.4, the concept of dynamic includes (`include_tasks`) versus static imports (`import_tasks`) was introduced to clearly define the differences in how `include` works between dynamic and static includes.

All attributes applied to a dynamic `include_*` would only apply to the include itself, while attributes applied to a static `import_*` would be inherited by the tasks within.

This separation was only partially implemented in Ansible version 2.4. As of Ansible version 2.5, this work is complete and the separation now behaves as designed; attributes applied to an `include_*` task will not be inherited by the tasks within.

To achieve an outcome similar to how Ansible worked prior to version 2.5, playbooks should use an explicit application of the attribute on the needed tasks, or use blocks to apply the attribute to many tasks. Another option is to use a static `import_*` when possible instead of a dynamic task.

#### OLD In Ansible 2.4:

```
- include_tasks: "{{ ansible_distribution }}.yaml"
  tags:
    - distro_include
```

#### Included file:

```
- block:
  - debug:
      msg: "In included file"

  - apt:
      name: nginx
      state: latest
```

#### NEW In Ansible 2.5:

#### Including task:

```
- include_tasks: "{{ ansible_distribution }}.yaml"
  tags:
    - distro_include
```

#### Included file:

```
- block:
  - debug:
      msg: "In included file"

  - apt:
      name: nginx
      state: latest
  tags:
    - distro_include
```

The relevant change in those examples is, that in Ansible 2.5, the included file defines the tag `distro_include` again. The tag is not inherited automatically.



## Fixed handling of keywords and inline variables

We made several fixes to how we handle keywords and 'inline variables', to avoid conflating the two. Unfortunately these changes mean you must specify whether *name* is a keyword or a variable when calling roles. If you have playbooks that look like this:

```
roles:
  - { role: myrole, name: Justin, othervar: othervalue, become: True}
```

You will run into errors because Ansible reads *name* in this context as a keyword. Beginning in 2.5, if you want to use a variable name that is also a keyword, you must explicitly declare it as a variable for the role:

```
roles:
  - { role: myrole, vars: {name: Justin, othervar: othervalue}, become: True}
```

For a full list of keywords see [::ref::Playbook Keywords](#).

## Migrating from with\_X to loop

With the release of Ansible 2.5, the recommended way to perform loops is the use the new `loop` keyword instead of `with_X` style loops.

In many cases, `loop` syntax is better expressed using filters instead of more complex use of `query` or `lookup`.

The following examples will show how to convert many common `with_` style loops to `loop` and filters.

### with\_list

`with_list` is directly replaced by `loop`.

```
- name: with_list
  debug:
    msg: "{{ item }}"
  with_list:
    - one
    - two

- name: with_list -> loop
  debug:
    msg: "{{ item }}"
  loop:
    - one
    - two
```

### with\_items

`with_items` is replaced by `loop` and the `flatten` filter.

```
- name: with_items
  debug:
    msg: "{{ item }}"
  with_items: "{{ items }}"
```

(continues on next page)

()

```
- name: with_items -> loop
  debug:
    msg: "{{ item }}"
  loop: "{{ items|flatten(levels=1) }}"
```

## with\_indexed\_items

with\_indexed\_items is replaced by loop, the flatten filter and loop\_control.index\_var.

```
- name: with_indexed_items
  debug:
    msg: "{{ item.0 }}" - "{{ item.1 }}"
  with_indexed_items: "{{ items }}"

- name: with_indexed_items -> loop
  debug:
    msg: "{{ index }}" - "{{ item }}"
  loop: "{{ items|flatten(levels=1) }}"
  loop_control:
    index_var: index
```

## with\_flattened

with\_flattened is replaced by loop and the flatten filter.

```
- name: with_flattened
  debug:
    msg: "{{ item }}"
  with_flattened: "{{ items }}"

- name: with_flattened -> loop
  debug:
    msg: "{{ item }}"
  loop: "{{ items|flatten }}"
```

## with\_together

with\_together is replaced by loop and the zip filter.

```
- name: with_together
  debug:
    msg: "{{ item.0 }}" - "{{ item.1 }}"
  with_together:
    - "{{ list_one }}"
    - "{{ list_two }}"

- name: with_together -> loop
  debug:
    msg: "{{ item.0 }}" - "{{ item.1 }}"
  loop: "{{ list_one|zip(list_two)|list }}"
```

## with\_dict

`with_dict` can be substituted by `loop` and either the `dictsort` or `dict2items` filters.

```

- name: with_dict
  debug:
    msg: "{{ item.key }} - {{ item.value }}"
  with_dict: "{{ dictionary }}"

- name: with_dict -> loop (option 1)
  debug:
    msg: "{{ item.key }} - {{ item.value }}"
  loop: "{{ dictionary|dict2items }}"

- name: with_dict -> loop (option 2)
  debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  loop: "{{ dictionary|dictsort }}"

```

## with\_sequence

`with_sequence` is replaced by `loop` and the `range` function, and potentially the `format` filter.

```

- name: with_sequence
  debug:
    msg: "{{ item }}"
  with_sequence: start=0 end=4 stride=2 format=testuser%02x

- name: with_sequence -> loop
  debug:
    msg: "{{ 'testuser%02x' | format(item) }}"
    # range is exclusive of the end point
  loop: "{{ range(0, 4 + 1, 2)|list }}"

```

## with\_subelements

`with_subelements` is replaced by `loop` and the `subelements` filter.

```

- name: with_subelements
  debug:
    msg: "{{ item.0.name }} - {{ item.1 }}"
  with_subelements:
    - "{{ users }}"
    - mysql.hosts

- name: with_subelements -> loop
  debug:
    msg: "{{ item.0.name }} - {{ item.1 }}"
  loop: "{{ users|subelements('mysql.hosts') }}"

```

## with\_nested/with\_cartesian

`with_nested` and `with_cartesian` are replaced by `loop` and the `product` filter.

```
- name: with_nested
  debug:
    msg: "{{ item.0 }}" - "{{ item.1 }}"
  with_nested:
    - "{{ list_one }}"
    - "{{ list_two }}"

- name: with_nested -> loop
  debug:
    msg: "{{ item.0 }}" - "{{ item.1 }}"
  loop: "{{ list_one|product(list_two)|list }}"
```

### with\_random\_choice

with\_random\_choice is replaced by just use of the random filter, without need of loop.

```
- name: with_random_choice
  debug:
    msg: "{{ item }}"
  with_random_choice: "{{ my_list }}"

- name: with_random_choice -> loop (No loop is needed here)
  debug:
    msg: "{{ my_list|random }}"
  tags: random
```

## Deprecated

### Jinja tests used as filters

Using Ansible-provided Jinja tests as filters will be removed in Ansible 2.9.

Prior to Ansible 2.5, Jinja tests included within Ansible were most often used as filters. The large difference in use is that filters are referenced as `variable | filter_name` while Jinja tests are referenced as `variable is test_name`.

Jinja tests are used for comparisons, while filters are used for data manipulation and have different applications in Jinja. This change is to help differentiate the concepts for a better understanding of Jinja, and where each can be appropriately used.

As of Ansible 2.5, using an Ansible provided Jinja test with filter syntax, will display a deprecation error.

**OLD** In Ansible 2.4 (and earlier) the use of an Ansible included Jinja test would likely look like this:

```
when:
  - result | failed
  - not result | success
```

**NEW** In Ansible 2.5 it should be changed to look like this:

```
when:
  - result is failed
  - results is not successful
```

In addition to the deprecation warnings, many new tests have been introduced that are aliases of the old tests. These new tests make more sense grammatically with the jinja test syntax, such as the new `successful` test which aliases `success`.

```
when: result is successful
```

See [Tests](#) for more information.

Additionally, a script was created to assist in the conversion for tests using filter syntax to proper jinja test syntax. This script has been used to convert all of the Ansible integration tests to the correct format. There are a few limitations documented, and all changes made by this script should be evaluated for correctness before executing the modified playbooks. The script can be found at [https://github.com/ansible/ansible/blob/devel/hacking/fix\\_test\\_syntax.py](https://github.com/ansible/ansible/blob/devel/hacking/fix_test_syntax.py).

## Modules

Major changes in popular modules are detailed here.

### github\_release

In Ansible versions 2.4 and older, after creating a GitHub release using the `create_release` state, the `github_release` module reported state as `skipped`. In Ansible version 2.5 and later, after creating a GitHub release using the `create_release` state, the `github_release` module now reports state as `changed`.

## Modules removed

The following modules no longer exist:

- `nxos_mtu` use `nxos_system`'s `system_mtu` option or `nxos_interface` instead
- `cl_interface_policy` use `nclu` instead
- `cl_bridge` use `nclu` instead
- `cl_img_install` use `nclu` instead
- `cl_ports` use `nclu` instead
- `cl_license` use `nclu` instead
- `cl_interface` use `nclu` instead
- `cl_bond` use `nclu` instead
- `ec2_vpc` use `ec2_vpc_net` along with supporting modules `ec2_vpc_igw`, `ec2_vpc_route_table`, `ec2_vpc_subnet`, `ec2_vpc_dhcp_option`, `ec2_vpc_nat_gateway`, `ec2_vpc_nacl` instead.
- `ec2_ami_search` use `ec2_ami_facts` instead
- `docker` use `docker_container` and `docker_image` instead

## Deprecation notices

The following modules will be removed in Ansible 2.9. Please update your playbooks accordingly.

- Apstra's `aos_*` modules are deprecated as they do not work with AOS 2.1 or higher. See new modules at <https://github.com/apstra>.
- `nxos_ip_interface` use `nxos_l3_interface` instead.

- `nxos_portchannel` use `nxos_linkagg` instead.
- `nxos_switchport` use `nxos_l2_interface` instead.
- `panos_security_policy` use `panos_security_rule` instead.
- `panos_nat_policy` use `panos_nat_rule` instead.
- `vsphere_guest` use `vmware_guest` instead.

### Noteworthy module changes

- The `stat` and `win_stat` modules have changed the default of the option `get_md5` from `true` to `false`.

This option will be removed starting with Ansible version 2.9. The options `get_checksum: True` and `checksum_algorithm: md5` can still be used if an MD5 checksum is desired.

- `osx_say` module was renamed into `say`.
- Several modules which could deal with symlinks had the default value of their `follow` option changed as part of a feature to [standardize the behavior of follow](#):
  - The `file` module changed from `follow=False` to `follow=True` because its purpose is to modify the attributes of a file and most systems do not allow attributes to be applied to symlinks, only to real files.
  - The `replace` module had its `follow` parameter removed because it inherently modifies the content of an existing file so it makes no sense to operate on the link itself.
  - The `blockinfile` module had its `follow` parameter removed because it inherently modifies the content of an existing file so it makes no sense to operate on the link itself.
  - In Ansible-2.5.3, the `template` module became more strict about its `src` file being proper utf-8. Previously, non-utf8 contents in a template module `src` file would result in a mangled output file (the non-utf8 characters would be replaced with a unicode replacement character). Now, on Python2, the module will error out with the message, "Template source files must be utf-8 encoded". On Python3, the module will first attempt to pass the non-utf8 characters through verbatim and fail if that does not succeed.

### Plugins

As a developer, you can now use 'doc fragments' for common configuration options on plugin types that support the new plugin configuration system.

### Inventory

Inventory plugins have been fine tuned, and we have started to add some common features:

- The ability to use a cache plugin to avoid costly API/DB queries is disabled by default. If using inventory scripts, some may already support a cache, but it is outside of Ansible's knowledge and control. Moving to the internal cache will allow you to use Ansible's existing cache refresh/invalidation mechanisms.
- A new 'auto' plugin, enabled by default, that can automatically detect the correct plugin to use IF that plugin is using our 'common YAML configuration format'. The previous `host_list`, `script`, `yaml` and `ini` plugins still work as they did, the `auto` plugin is now the last one we attempt to use. If you had customized the enabled plugins you should revise the setting to include the new `auto` plugin.

## Shell

Shell plugins have been migrated to the new plugin configuration framework. It is now possible to customize more settings, and settings which were previously 'global' can now also be overridden using host specific variables.

For example, `system_temps` is a new setting that allows you to control what Ansible will consider a 'system temporary dir'. This is used when escalating privileges for a non-administrative user. Previously this was hardcoded to `'/tmp'`, which some systems cannot use for privilege escalation. This setting now defaults to `[ '/var/tmp', '/tmp' ]`.

Another new setting is `admin_users` which allows you to specify a list of users to be considered 'administrators'. Previously this was hardcoded to `root`. It now it defaults to `[root, toor, admin]`. This information is used when choosing between your `remote_temp` and `system_temps` directory.

For a full list, check the shell plugin you are using, the default shell plugin is `sh`.

Those that had to work around the global configuration limitations can now migrate to a per host/group settings, but also note that the new defaults might conflict with existing usage if the assumptions don't correlate to your environment.

## Filter

The lookup plugin API now throws an error if a non-iterable value is returned from a plugin. Previously, numbers or other non-iterable types returned by a plugin were accepted without error or warning. This change was made because plugins should always return a list. Please note that plugins that return strings and other non-list iterable values will not throw an error, but may cause unpredictable behavior. If you have a custom lookup plugin that does not return a list, you should modify it to wrap the return values in a list.

## Lookup

A new option was added to lookup plugins globally named `error` which allows you to control how errors produced by the lookup are handled, before this option they were always fatal. Valid values for this option are `warn`, `ignore` and `strict`. See the [lookup](#) page for more details.

## Porting custom scripts

No notable changes.

## Network

### Expanding documentation

We're expanding the network documentation. There's new content and a [new Ansible Network landing page](#). We will continue to build the network-related documentation moving forward.

### Top-level connection arguments will be removed in 2.9

Top-level connection arguments like `username`, `host`, and `password` are deprecated and will be removed in version 2.9.

**OLD** In Ansible < 2.4

```
- name: example of using top-level options for connection properties
  ios_command:
    commands: show version
    host: "{{ inventory_hostname }}"
    username: cisco
    password: cisco
    authorize: yes
    auth_pass: cisco
```

The deprecation warnings reflect this schedule. The task above, run in Ansible 2.5, will result in:

```
[DEPRECATION WARNING]: Param 'username' is deprecated. See the module docs for more
↪information. This feature will be removed in version
2.9. Deprecation warnings can be disabled by setting deprecation_warnings=False in
↪ansible.cfg.
[DEPRECATION WARNING]: Param 'password' is deprecated. See the module docs for more
↪information. This feature will be removed in version
2.9. Deprecation warnings can be disabled by setting deprecation_warnings=False in
↪ansible.cfg.
[DEPRECATION WARNING]: Param 'host' is deprecated. See the module docs for more
↪information. This feature will be removed in version 2.9.
Deprecation warnings can be disabled by setting deprecation_warnings=False in ansible.
↪cfg.
```

We recommend using the new connection types `network_cli` and `netconf` (see below), using standard Ansible connection properties, and setting those properties in inventory by group. As you update your playbooks and inventory files, you can easily make the change to become for privilege escalation (on platforms that support it). For more information, see the [using become with network modules](#) guide and the [platform documentation](#).

### Adding persistent connection types `network_cli` and `netconf`

Ansible 2.5 introduces two top-level persistent connection types, `network_cli` and `netconf`. With `connection: local`, each task passed the connection parameters, which had to be stored in your playbooks. With `network_cli` and `netconf` the playbook passes the connection parameters once, so you can pass them at the command line if you prefer. We recommend you use `network_cli` and `netconf` whenever possible. Note that eAPI and NX-API still require `local` connections with `provider` dictionaries. See the [platform documentation](#) for more information. Unless you need a `local` connection, update your playbooks to use `network_cli` or `netconf` and to specify your connection variables with standard Ansible connection variables:

#### OLD In Ansible 2.4

```
---
vars:
  cli:
    host: "{{ inventory_hostname }}"
    username: operator
    password: secret
    transport: cli

tasks:
- nxos_config:
  src: config.j2
  provider: "{{ cli }}"
  username: admin
  password: admin
```



**NEW In Ansible 2.5**

```
[nxos:vars]
ansible_connection=network_cli
ansible_network_os=nxos
ansible_user=operator
ansible_password=secret
```

```
tasks:
- nxos_config:
    src: config.j2
```

Using a provider dictionary with either `network_cli` or `netconf` will result in a warning.

**Developers: Shared Module Utilities Moved**

Beginning with Ansible 2.5, shared module utilities for network modules moved to `ansible.module_utils.network`.

- Platform-independent utilities are found in `ansible.module_utils.network.common`
- Platform-specific utilities are found in `ansible.module_utils.network.{{ platform }}`

If your module uses shared module utilities, you must update all references. For example, change:

**OLD In Ansible 2.4**

```
from ansible.module_utils.vyos import get_config, load_config
```

**NEW In Ansible 2.5**

```
from ansible.module_utils.network.vyos.vyos import get_config, load_config
```

See the module utilities developer guide see [Appendix: Module Utilities](#) for more information.

**1.3.5 Ansible 2.6 Porting Guide**

This section discusses the behavioral changes between Ansible 2.5 and Ansible 2.6.

It is intended to assist in updating your playbooks, plugins and other parts of your Ansible infrastructure so they will work with this version of Ansible.

We suggest you read this page along with [Ansible Changelog for 2.6](#) to understand what updates you may need to make.

This document is part of a collection on porting. The complete list of porting guides can be found at [porting guides](#).

**Topics**

- [Ansible 2.6 Porting Guide](#)
  - [Playbook](#)
  - [Deprecated](#)
  - [Modules](#)

- \* *Modules removed*
- \* *Deprecation notices*
- \* *Noteworthy module changes*
- *Plugins*
  - \* *Deprecation notices*
  - \* *Noteworthy plugin changes*
- *Porting custom scripts*
- *Networking*

### Playbook

- The deprecated task option `always_run` has been removed, please use `check_mode: no` instead.

### Deprecated

- In the `nxos_igmp_interface` module, `oif_prefix` and `oif_source` properties are deprecated. Use `ois_ps` parameter with a dictionary of prefix and source to values instead.

### Modules

Major changes in popular modules are detailed here

#### Modules removed

The following modules no longer exist:

#### Deprecation notices

The following modules will be removed in Ansible 2.10. Please update your playbooks accordingly.

- `k8s_raw` use `k8s` instead.
- `openshift_raw` use `k8s` instead.
- `openshift_scale` use `k8s_scale` instead.

#### Noteworthy module changes

- The `upgrade` module option for `win_chocolatey` has been removed; use `state: latest` instead.
- The `reboot` module option for `win_feature` has been removed; use the `win_reboot` action plugin instead
- The `win_iis_webapppool` module no longer accepts a string for the `attributes` module option; use the free form dictionary value instead

- The `name` module option for `win_package` has been removed; this is not used anywhere and should just be removed from your playbooks
- The `win_regedit` module no longer automatically corrects the hive path `HCCC` to `HKCC`; use `HKCC` because this is the correct hive path
- The `file` module now emits a deprecation warning when `src` is specified with a state other than `hard` or `link` as it is only supposed to be useful with those. This could have an effect on people who were depending on a buggy interaction between `src` and other state's to place files into a subdirectory. For instance:

```
$ ansible localhost -m file -a 'path=/var/lib src=/tmp/ state=directory'
```

Would create a directory named `/tmp/lib`. Instead of the above, simply spell out the entire destination path like this:

```
$ ansible localhost -m file -a 'path=/tmp/lib state=directory'
```

- The `k8s_raw` and `openshift_raw` modules have been aliased to the new `k8s` module.
- The `k8s` module supports all Kubernetes resources including those from Custom Resource Definitions and aggregated API servers. This includes all OpenShift resources.
- The `k8s` module will not accept resources where subkeys have been snake\_cased. This was a workaround that was suggested with the `k8s_raw` and `openshift_raw` modules.
- The `k8s` module may not accept resources where the `api_version` has been changed to match the shortened version in the Kubernetes Python client. You should now specify the proper full Kubernetes `api_version` for a resource.
- The `k8s` module can now process multi-document YAML files if they are passed with the `src` parameter. It will process each document as a separate resource. Resources provided inline with the `resource_definition` parameter must still be a single document.
- The `k8s` module will not automatically change `Project` creation requests into `ProjectRequest` creation requests as the `openshift_raw` module did. You must now specify the `ProjectRequest` kind explicitly.
- The `k8s` module will not automatically remove secrets from the Ansible return values (and by extension the log). In order to prevent secret values in a task from being logged, specify the `no_log` parameter on the task block.
- The `k8s_scale` module now supports scalable OpenShift objects, such as `DeploymentConfig`.

## Plugins

### Deprecation notices

The following modules will be removed in Ansible 2.10. Please update your playbooks accordingly.

- `openshift` use `k8s` instead.

### Noteworthy plugin changes

- The `k8s` lookup plugin now supports all Kubernetes resources including those from Custom Resource Definitions and aggregated API servers. This includes all OpenShift resources.
- The `k8s` lookup plugin may not accept resources where the `api_version` has been changed to match the shortened version in the Kubernetes Python client. You should now specify the proper full Kubernetes `api_version` for a resource.

- The `k8s` lookup plugin will no longer remove secrets from the Ansible return values (and by extension the log). In order to prevent secret values in a task from being logged, specify the `no_log` parameter on the task block.

### Porting custom scripts

No notable changes.

### Networking

No notable changes.

## 1.3.6 Ansible 2.7 Porting Guide

This section discusses the behavioral changes between Ansible 2.6 and Ansible 2.7.

It is intended to assist in updating your playbooks, plugins and other parts of your Ansible infrastructure so they will work with this version of Ansible.

We suggest you read this page along with [Ansible Changelog for 2.7](#) to understand what updates you may need to make.

This document is part of a collection on porting. The complete list of porting guides can be found at [porting guides](#).

#### Topics

- *Ansible 2.7 Porting Guide*
  - *Playbook*
    - \* *Role Precedence Fix during Role Loading*
  - *Deprecated*
    - \* *Using a loop on a package module via `squash_actions`*
  - *Modules*
    - \* *Modules removed*
    - \* *Deprecation notices*
    - \* *Noteworthy module changes*
  - *Plugins*
  - *Porting custom scripts*
  - *Networking*

### Playbook

#### Role Precedence Fix during Role Loading

Ansible 2.7 makes a small change to variable precedence when loading roles, resolving a bug, ensuring that role loading matches *variable precedence expectations*.

Before Ansible 2.7, when loading a role, the variables defined in the role's `vars/main.yml` and `defaults/main.yml` were not available when parsing the role's `tasks/main.yml` file. This prevented the role from utilizing these variables when being parsed. The problem manifested when `import_tasks` or `import_role` was used with a variable defined in the role's `vars` or `defaults`.

In Ansible 2.7, role `vars` and `defaults` are now parsed before `tasks/main.yml`. This can cause a change in behavior if the same variable is defined at the play level and the role level with different values, and leveraged in `import_tasks` or `import_role` to define the role or file to import.

## Deprecated

### Using a loop on a package module via `squash_actions`

The use of `squash_actions` to invoke a package module, such as "yum", to only invoke the module once is deprecated, and will be removed in Ansible 2.11.

Instead of relying on implicit squashing, tasks should instead supply the list directly to the `name`, `pkg` or `package` parameter of the module. This functionality has been supported in most modules since Ansible 2.3.

**OLD** In Ansible 2.6 (and earlier) the following task would invoke the "yum" module only 1 time to install multiple packages

```
- name: Install packages
  yum:
    name: "{{ item }}"
    state: present
  with_items: "{{ packages }}"
```

**NEW** In Ansible 2.7 it should be changed to look like this:

```
- name: Install packages
  yum:
    name: "{{ packages }}"
    state: present
```

## Modules

Major changes in popular modules are detailed here

- The `DEFAULT_SYSLOG_FACILITY` configuration option tells Ansible modules to use a specific [syslog facility](#) when logging information on all managed machines. Due to a bug with older Ansible versions, this setting did not affect machines using `journald` with the `systemd` Python bindings installed. On those machines, Ansible log messages were sent to `/var/log/messages`, even if you set `DEFAULT_SYSLOG_FACILITY`. Ansible 2.7 fixes this bug, routing all Ansible log messages according to the value set for `DEFAULT_SYSLOG_FACILITY`. If you have `DEFAULT_SYSLOG_FACILITY` configured, the location of remote logs on systems which use `journald` may change.
- The `lineinfile` module was changed to show a warning when using an empty string as a regexp. Since an empty regexp matches every line in a file, it will replace the last line in a file rather than inserting. If this is the desired behavior, use `'^'` which will match every line and will not trigger the warning.

## Modules removed

The following modules no longer exist:

### Deprecation notices

The following modules will be removed in Ansible 2.10. Please update your playbooks accordingly.

### Noteworthy module changes

No notable changes.

### Plugins

No notable changes.

### Porting custom scripts

No notable changes.

### Networking

No notable changes.

## 1.4 User Guide

Welcome to the Ansible User Guide!

This guide covers how to work with Ansible, including using the command line, working with inventory, and writing playbooks.

### 1.4.1 Ansible Quickstart

We've recorded a short video that introduces Ansible.

The [quickstart video](#) is about 13 minutes long and gives you a high level introduction to Ansible – what it does and how to use it. We'll also tell you about other products in the Ansible ecosystem.

Enjoy, and be sure to visit the rest of the documentation to learn more.

### 1.4.2 Getting Started

#### Topics

- *Getting Started*
  - *Foreword*
  - *Remote Connection Information*
  - *Your first commands*
  - *Host Key Checking*

## Foreword

Now that you’ve read the *installation guide* and installed Ansible, it’s time to get started with some ad-hoc commands.

What we are showing first are not the powerful configuration/deployment/orchestration features of Ansible. These features are handled by playbooks which are covered in a separate section.

This section is about how to initially get Ansible running. Once you understand these concepts, read *Introduction To Ad-Hoc Commands* for some more detail, and then you’ll be ready to begin learning about playbooks and explore the most interesting parts!

## Remote Connection Information

Before we get started, it’s important to understand how Ansible communicates with remote machines over SSH.

By default, Ansible will try to use native OpenSSH for remote communication when possible. This enables ControlPersist (a performance feature), Kerberos, and options in `~/.ssh/config` such as Jump Host setup. However, when using Enterprise Linux 6 operating systems as the control machine (Red Hat Enterprise Linux and derivatives such as CentOS), the version of OpenSSH may be too old to support ControlPersist. On these operating systems, Ansible will fallback into using a high-quality Python implementation of OpenSSH called ‘paramiko’. If you wish to use features like Kerberized SSH and more, consider using Fedora, macOS, or Ubuntu as your control machine until a newer version of OpenSSH is available for your platform.

Occasionally you’ll encounter a device that doesn’t support SFTP. This is rare, but should it occur, you can switch to SCP mode in *Configuring Ansible*.

When speaking with remote machines, Ansible by default assumes you are using SSH keys. SSH keys are encouraged but password authentication can also be used where needed by supplying the option `--ask-pass`. If using sudo features and when sudo requires a password, also supply `--ask-become-pass` (previously `--ask-sudo-pass` which has been deprecated).

---

: Ansible does not expose a channel to allow communication between the user and the ssh process to accept a password manually to decrypt an ssh key when using the ssh connection plugin (which is the default). The use of `ssh-agent` is highly recommended.

---

While it may be common sense, it is worth sharing: Any management system benefits from being run near the machines being managed. If you are running Ansible in a cloud, consider running it from a machine inside that cloud. In most cases this will work better than on the open Internet.

As an advanced topic, Ansible doesn’t just have to connect remotely over SSH. The transports are pluggable, and there are options for managing things locally, as well as managing chroot, lxc, and jail containers. A mode called ‘ansible-pull’ can also invert the system and have systems ‘phone home’ via scheduled git checkouts to pull configuration directives from a central repository.

## Your first commands

Now that you’ve installed Ansible, it’s time to get started with some basics.

Edit (or create) `/etc/ansible/hosts` and put one or more remote systems in it. Your public SSH key should be located in `authorized_keys` on those systems:

```
192.0.2.50
aserver.example.org
bserver.example.org
```

This is an inventory file, which is also explained in greater depth here: [Working with Inventory](#).

We'll assume you are using SSH keys for authentication. To set up SSH agent to avoid retyping passwords, you can do:

```
$ ssh-agent bash
$ ssh-add ~/.ssh/id_rsa
```

(Depending on your setup, you may wish to use Ansible's `--private-key` option to specify a pem file instead)

Now ping all your nodes:

```
$ ansible all -m ping
```

Ansible will attempt to remote connect to the machines using your current user name, just like SSH would. To override the remote user name, just use the `'-u'` parameter.

If you would like to access sudo mode, there are also flags to do that:

```
# as bruce
$ ansible all -m ping -u bruce
# as bruce, sudoing to root
$ ansible all -m ping -u bruce --sudo
# as bruce, sudoing to batman
$ ansible all -m ping -u bruce --sudo --sudo-user batman

# With latest version of ansible `sudo` is deprecated so use become
# as bruce, sudoing to root
$ ansible all -m ping -u bruce -b
# as bruce, sudoing to batman
$ ansible all -m ping -u bruce -b --become-user batman
```

(The sudo implementation is changeable in Ansible's configuration file if you happen to want to use a sudo replacement. Flags passed to sudo (like `-H`) can also be set there.)

Now run a live command on all of your nodes:

```
$ ansible all -a "/bin/echo hello"
```

Congratulations! You've just contacted your nodes with Ansible. It's soon going to be time to: read about some more real-world cases in [Introduction To Ad-Hoc Commands](#), explore what you can do with different modules, and to learn about the Ansible [Working With Playbooks](#) language. Ansible is not just about running commands, it also has powerful configuration management and deployment features. There's more to explore, but you already have a fully working infrastructure!

### Tips

When running commands, you can specify the local server by using `"localhost"` or `"127.0.0.1"` for the server name.

Example:

```
$ ansible localhost -m ping -e 'ansible_python_interpreter="/usr/bin/env python"'
```

You can specify localhost explicitly by adding this to your inventory file:

```
localhost ansible_connection=local ansible_python_interpreter="/usr/bin/env python"
```



## Host Key Checking

Ansible has host key checking enabled by default.

If a host is reinstalled and has a different key in 'known\_hosts', this will result in an error message until corrected. If a host is not initially in 'known\_hosts' this will result in prompting for confirmation of the key, which results in an interactive experience if using Ansible, from say, cron. You might not want this.

If you understand the implications and wish to disable this behavior, you can do so by editing `/etc/ansible/ansible.cfg` or `~/.ansible.cfg`:

```
[defaults]
host_key_checking = False
```

Alternatively this can be set by the `ANSIBLE_HOST_KEY_CHECKING` environment variable:

```
$ export ANSIBLE_HOST_KEY_CHECKING=False
```

Also note that host key checking in paramiko mode is reasonably slow, therefore switching to 'ssh' is also recommended when using this feature.

Ansible will log some information about module arguments on the remote system in the remote syslog, unless a task or play is marked with a "no\_log: True" attribute. This is explained later.

To enable basic logging on the control machine see [Configuring Ansible](#) document and set the 'log\_path' configuration file setting. Enterprise users may also be interested in [Ansible Tower](#). Tower provides a very robust database logging feature where it is possible to drill down and see history based on hosts, projects, and particular inventories over time – explorable both graphically and through a REST API.

:

**Working with Inventory** More information about inventory

**Introduction To Ad-Hoc Commands** Examples of basic commands

**Working With Playbooks** Learning Ansible's configuration management language

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**irc.freenode.net** #ansible IRC chat channel

## 1.4.3 Working with Command Line Tools

Most users are familiar with *ansible* and *ansible-playbook*, but those are not the only utilities Ansible provides. Below is a complete list of Ansible utilities. Each page contains a description of the utility and a listing of supported parameters.

## 1.4.4 Introduction To Ad-Hoc Commands

### Topics

- *Introduction To Ad-Hoc Commands*
  - *Parallelism and Shell Commands*
  - *File Transfer*
  - *Managing Packages*

- *Users and Groups*
- *Deploying From Source Control*
- *Managing Services*
- *Time Limited Background Operations*
- *Gathering Facts*

The following examples show how to use `/usr/bin/ansible` for running ad hoc tasks.

What's an ad-hoc command?

An ad-hoc command is something that you might type in to do something really quick, but don't want to save for later.

This is a good place to start to understand the basics of what Ansible can do prior to learning the playbooks language – ad-hoc commands can also be used to do quick things that you might not necessarily want to write a full playbook for.

Generally speaking, the true power of Ansible lies in playbooks. Why would you use ad-hoc tasks versus playbooks?

For instance, if you wanted to power off all of your lab for Christmas vacation, you could execute a quick one-liner in Ansible without writing a playbook.

For configuration management and deployments, though, you'll want to pick up on using `'/usr/bin/ansible-playbook'` – the concepts you will learn here will port over directly to the playbook language.

(See *Working With Playbooks* for more information about those)

If you haven't read *Working with Inventory* already, please look that over a bit first and then we'll get going.

### Parallelism and Shell Commands

Arbitrary example.

Let's use Ansible's command line tool to reboot all web servers in Atlanta, 10 at a time. First, let's set up SSH-agent so it can remember our credentials:

```
$ ssh-agent bash
$ ssh-add ~/.ssh/id_rsa
```

If you don't want to use ssh-agent and want to instead SSH with a password instead of keys, you can with `--ask-pass (-k)`, but it's much better to just use ssh-agent.

Now to run the command on all servers in a group, in this case, *atlanta*, in 10 parallel forks:

```
$ ansible atlanta -a "/sbin/reboot" -f 10
```

`/usr/bin/ansible` will default to running from your user account. If you do not like this behavior, pass in `"-u username"`. If you want to run commands as a different user, it looks like this:

```
$ ansible atlanta -a "/usr/bin/foo" -u username
```

Often you'll not want to just do things from your user account. If you want to run commands through privilege escalation:

```
$ ansible atlanta -a "/usr/bin/foo" -u username --become [--ask-become-pass]
```

Use `--ask-become-pass (-K)` if you are not using a passwordless privilege escalation method (`sudo/su/pfexec/doas/etc`). This will interactively prompt you for the password to use. Use of a passwordless setup makes things easier to automate, but it's not required.

It is also possible to become a user other than root using `--become-user`:

```
$ ansible atlanta -a "/usr/bin/foo" -u username --become --become-user otheruser [--ask-become-pass]
```

: Rarely, some users have security rules where they constrain their `sudo/pbrun/doas` environment to running specific command paths only. This does not work with ansible's no-bootstrapping philosophy and hundreds of different modules. If doing this, use Ansible from a special account that does not have this constraint. One way of doing this without sharing access to unauthorized users would be gating Ansible with *Ansible Tower*, which can hold on to an SSH credential and let members of certain organizations use it on their behalf without having direct access.

Ok, so those are basics. If you didn't read about patterns and groups yet, go back and read *Working with Patterns*.

The `-f 10` in the above specifies the usage of 10 simultaneous processes to use. You can also set this in *Configuring Ansible* to avoid setting it again. The default is actually 5, which is really small and conservative. You are probably going to want to talk to a lot more simultaneous hosts so feel free to crank this up. If you have more hosts than the value set for the fork count, Ansible will talk to them, but it will take a little longer. Feel free to push this value as high as your system can handle!

You can also select what Ansible "module" you want to run. Normally commands also take a `-m` for module name, but the default module name is 'command', so we didn't need to specify that all of the time. We'll use `-m` in later examples to run some other *Working With Modules*.

: The command module does not support extended shell syntax like piping and redirects (although shell variables will always work). If your command requires shell-specific syntax, use the *shell* module instead. Read more about the differences on the *Working With Modules* page.

Using the shell module looks like this:

```
$ ansible raleigh -m shell -a 'echo $TERM'
```

When running any command with the Ansible *ad hoc* CLI (as opposed to *Playbooks*), pay particular attention to shell quoting rules, so the local shell doesn't eat a variable before it gets passed to Ansible. For example, using double rather than single quotes in the above example would evaluate the variable on the box you were on.

So far we've been demoing simple command execution, but most Ansible modules are not simple imperative scripts. Instead, they use a declarative model, calculating and executing the actions required to reach a specified final state. Furthermore, they achieve a form of idempotence by checking the current state before they begin, and if the current state matches the specified final state, doing nothing. However, we also recognize that running arbitrary commands can be valuable, so Ansible easily supports both.

## File Transfer

Here's another use case for the `/usr/bin/ansible` command line. Ansible can SCP lots of files to multiple machines in parallel.

To transfer a file directly to many servers:

```
$ ansible atlanta -m copy -a "src=/etc/hosts dest=/tmp/hosts"
```

If you use playbooks, you can also take advantage of the `template` module, which takes this another step further. (See module and playbook documentation).

The `file` module allows changing ownership and permissions on files. These same options can be passed directly to the `copy` module as well:

```
$ ansible webservers -m file -a "dest=/srv/foo/a.txt mode=600"
$ ansible webservers -m file -a "dest=/srv/foo/b.txt mode=600 owner=mdehaan
↪group=mdehaan"
```

The `file` module can also create directories, similar to `mkdir -p`:

```
$ ansible webservers -m file -a "dest=/path/to/c mode=755 owner=mdehaan group=mdehaan
↪state=directory"
```

As well as delete directories (recursively) and delete files:

```
$ ansible webservers -m file -a "dest=/path/to/c state=absent"
```

## Managing Packages

There are modules available for `yum` and `apt`. Here are some examples with `yum`.

Ensure a package is installed, but don't update it:

```
$ ansible webservers -m yum -a "name=acme state=present"
```

Ensure a package is installed to a specific version:

```
$ ansible webservers -m yum -a "name=acme-1.5 state=present"
```

Ensure a package is at the latest version:

```
$ ansible webservers -m yum -a "name=acme state=latest"
```

Ensure a package is not installed:

```
$ ansible webservers -m yum -a "name=acme state=absent"
```

Ansible has modules for managing packages under many platforms. If there isn't a module for your package manager, you can install packages using the `command` module or (better!) contribute a module for your package manager. Stop by the mailing list for info/details.

## Users and Groups

The `'user'` module allows easy creation and manipulation of existing user accounts, as well as removal of user accounts that may exist:

```
$ ansible all -m user -a "name=foo password=<crypted password here>"
$ ansible all -m user -a "name=foo state=absent"
```

See the [Working With Modules](#) section for details on all of the available options, including how to manipulate groups and group membership.

## Deploying From Source Control

Deploy your webapp straight from git:

```
$ ansible webserver -m git -a "repo=https://foo.example.org/repo.git dest=/srv/myapp_
↪version=HEAD"
```

Since Ansible modules can notify change handlers it is possible to tell Ansible to run specific tasks when the code is updated, such as deploying Perl/Python/PHP/Ruby directly from git and then restarting apache.

## Managing Services

Ensure a service is started on all webserver:

```
$ ansible webserver -m service -a "name=httpd state=started"
```

Alternatively, restart a service on all webserver:

```
$ ansible webserver -m service -a "name=httpd state=restarted"
```

Ensure a service is stopped:

```
$ ansible webserver -m service -a "name=httpd state=stopped"
```

## Time Limited Background Operations

Long running operations can be run in the background, and it is possible to check their status later. For example, to execute `long_running_operation` asynchronously in the background, with a timeout of 3600 seconds (`-B`), and without polling (`-P`):

```
$ ansible all -B 3600 -P 0 -a "/usr/bin/long_running_operation --do-stuff"
```

If you do decide you want to check on the job status later, you can use the `async_status` module, passing it the job id that was returned when you ran the original job in the background:

```
$ ansible web1.example.com -m async_status -a "jid=488359678239.2844"
```

Polling is built-in and looks like this:

```
$ ansible all -B 1800 -P 60 -a "/usr/bin/long_running_operation --do-stuff"
```

The above example says "run for 30 minutes max (`-B 30*60=1800`), poll for status (`-P`) every 60 seconds".

Poll mode is smart so all jobs will be started before polling will begin on any machine. Be sure to use a high enough `--forks` value if you want to get all of your jobs started very quickly. After the time limit (in seconds) runs out (`-B`), the process on the remote nodes will be terminated.

Typically you'll only be backgrounding long-running shell commands or software upgrades. Backgrounding the copy module does not do a background file transfer. *Playbooks* also support polling, and have a simplified syntax for this.

## Gathering Facts

Facts are described in the playbooks section and represent discovered variables about a system. These can be used to implement conditional execution of tasks but also just to get ad-hoc information about your system. You can see all facts via:

```
$ ansible all -m setup
```

It's also possible to filter this output to just export certain facts, see the "setup" module documentation for details.

Read more about facts at [Variables](#) once you're ready to read up on [Playbooks](#).

:

**Configuring Ansible** All about the Ansible config file

**all\_modules** A list of available modules

**Working With Playbooks** Using Ansible for configuration management & deployment

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**irc.freenode.net** #ansible IRC chat channel

### 1.4.5 Working with Inventory

#### Topics

- *Working with Inventory*
  - *Hosts and Groups*
  - *Host Variables*
  - *Group Variables*
  - *Groups of Groups, and Group Variables*
  - *Default groups*
  - *Splitting Out Host and Group Specific Data*
  - *How Variables Are Merged*
  - *List of Behavioral Inventory Parameters*
  - *Non-SSH connection types*

Ansible works against multiple systems in your infrastructure at the same time. It does this by selecting portions of systems listed in Ansible's inventory, which defaults to being saved in the location `/etc/ansible/hosts`. You can specify a different inventory file using the `-i <path>` option on the command line.

Not only is this inventory configurable, but you can also use multiple inventory files at the same time and pull inventory from dynamic or cloud sources or different formats (YAML, ini, etc), as described in [Working With Dynamic Inventory](#). Introduced in version 2.4, Ansible has inventory plugins to make this flexible and customizable.

#### Hosts and Groups

The inventory file can be in one of many formats, depending on the inventory plugins you have. For this example, the format for `/etc/ansible/hosts` is an INI-like (one of Ansible's defaults) and looks like this:

```
mail.example.com

[webservers]
foo.example.com
```

(continues on next page)

()

```
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

The headings in brackets are group names, which are used in classifying systems and deciding what systems you are controlling at what times and for what purpose.

A YAML version would look like:

```
all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
```

It is ok to put systems in more than one group, for instance a server could be both a webserver and a dbserver. If you do, note that variables will come from all of the groups they are a member of. Variable precedence is detailed in a later chapter.

If you have hosts that run on non-standard SSH ports you can put the port number after the hostname with a colon. Ports listed in your SSH config file won't be used with the *paramiko* connection but will be used with the *openssh* connection.

To make things explicit, it is suggested that you set them if things are not running on the default port:

```
badwolf.example.com:5309
```

Suppose you have just static IPs and want to set up some aliases that live in your host file, or you are connecting through tunnels. You can also describe hosts via variables:

In INI:

```
jumper ansible_port=5555 ansible_host=192.0.2.50
```

In YAML:

```
...
hosts:
  jumper:
    ansible_port: 5555
    ansible_host: 192.0.2.50
```

In the above example, trying to ansible against the host alias "jumper" (which may not even be a real hostname) will contact 192.0.2.50 on port 5555. Note that this is using a feature of the inventory file to define some special variables. Generally speaking, this is not the best way to define variables that describe your system policy, but we'll share suggestions on doing this later.

: Values passed in the INI format using the `key=value` syntax are not interpreted as Python literal structure (strings, numbers, tuples, lists, dicts, booleans, None), but as a string. For example `var=FALSE` would create a string equal to 'FALSE'. Do not rely on types set during definition, always make sure you specify type with a filter when needed when consuming the variable.

---

If you are adding a lot of hosts following similar patterns, you can do this rather than listing each hostname:

```
[webservers]
www[01:50].example.com
```

For numeric patterns, leading zeros can be included or removed, as desired. Ranges are inclusive. You can also define alphabetic ranges:

```
[databases]
db-[a:f].example.com
```

You can also select the connection type and user on a per host basis:

```
[targets]

localhost          ansible_connection=local
other1.example.com  ansible_connection=ssh      ansible_user=mpdehaan
other2.example.com  ansible_connection=ssh      ansible_user=mdehaan
```

As mentioned above, setting these in the inventory file is only a shorthand, and we'll discuss how to store them in individual files in the 'host\_vars' directory a bit later on.

## Host Variables

As described above, it is easy to assign variables to hosts that will be used later in playbooks:

```
[atlanta]
host1 http_port=80 maxRequestsPerChild=808
host2 http_port=303 maxRequestsPerChild=909
```

## Group Variables

Variables can also be applied to an entire group at once:

The INI way:

```
[atlanta]
host1
host2

[atlanta:vars]
ntp_server=ntp.atlanta.example.com
proxy=proxy.atlanta.example.com
```

The YAML version:

```
atlanta:
  hosts:
```

(continues on next page)



()

```

host1:
host2:
vars:
  ntp_server: ntp.atlanta.example.com
  proxy: proxy.atlanta.example.com

```

Be aware that this is only a convenient way to apply variables to multiple hosts at once; even though you can target hosts by group, **variables are always flattened to the host level** before a play is executed.

## Groups of Groups, and Group Variables

It is also possible to make groups of groups using the `:children` suffix in INI or the `children:` entry in YAML. You can apply variables using `:vars` or `vars::`

```

[atlanta]
host1
host2

[raleigh]
host2
host3

[southeast:children]
atlanta
raleigh

[southeast:vars]
some_server=foo.southeast.example.com
halon_system_timeout=30
self_destruct_countdown=60
escape_pods=2

[usa:children]
southeast
northeast
southwest
northwest

```

```

all:
  children:
    usa:
      children:
        southeast:
          children:
            atlanta:
              hosts:
                host1:
                host2:
            raleigh:
              hosts:
                host2:
                host3:
          vars:
            some_server: foo.southeast.example.com
            halon_system_timeout: 30

```

(continues on next page)

```

        self_destruct_countdown: 60
        escape_pods: 2
    northeast:
    northwest:
    southwest:

```

If you need to store lists or hash data, or prefer to keep host and group specific variables separate from the inventory file, see the next section. Child groups have a couple of properties to note:

- Any host that is member of a child group is automatically a member of the parent group.
- A child group's variables will have higher precedence (override) a parent group's variables.
- Groups can have multiple parents and children, but not circular relationships.
- Hosts can also be in multiple groups, but there will only be **one** instance of a host, merging the data from the multiple groups.

## Default groups

There are two default groups: `all` and `ungrouped`. `all` contains every host. `ungrouped` contains all hosts that don't have another group aside from `all`. Every host will always belong to at least 2 groups. Though `all` and `ungrouped` are always present, they can be implicit and not appear in group listings like `group_names`.

## Splitting Out Host and Group Specific Data

The preferred practice in Ansible is to not store variables in the main inventory file.

In addition to storing variables directly in the inventory file, host and group variables can be stored in individual files relative to the inventory file (not directory, it is always the file).

These variable files are in YAML format. Valid file extensions include `.yaml`, `.yml`, `.json`, or no file extension. See [YAML Syntax](#) if you are new to YAML.

Assuming the inventory file path is:

```
/etc/ansible/hosts
```

If the host is named `'foosball'`, and in groups `'raleigh'` and `'webserver'`, variables in YAML files at the following locations will be made available to the host:

```

/etc/ansible/group_vars/raleigh # can optionally end in '.yaml', '.yml', or '.json'
/etc/ansible/group_vars/webserver
/etc/ansible/host_vars/foosball

```

For instance, suppose you have hosts grouped by datacenter, and each datacenter uses some different servers. The data in the groupfile `'/etc/ansible/group_vars/raleigh'` for the `'raleigh'` group might look like:

```

---
ntp_server: acme.example.org
database_server: storage.example.org

```

It is okay if these files do not exist, as this is an optional feature.

As an advanced use case, you can create *directories* named after your groups or hosts, and Ansible will read all the files in these directories in lexicographical order. An example with the `'raleigh'` group:

```
/etc/ansible/group_vars/raleigh/db_settings
/etc/ansible/group_vars/raleigh/cluster_settings
```

All hosts that are in the 'raleigh' group will have the variables defined in these files available to them. This can be very useful to keep your variables organized when a single file starts to be too big, or when you want to use [Ansible Vault](#) on a part of a group's variables.

Tip: The `group_vars/` and `host_vars/` directories can exist in the playbook directory OR the inventory directory. If both paths exist, variables in the playbook directory will override variables set in the inventory directory.

Tip: Keeping your inventory file and variables in a git repo (or other version control) is an excellent way to track changes to your inventory and host variables.

## How Variables Are Merged

By default variables are merged/flattened to the specific host before a play is run. This keeps Ansible focused on the Host and Task, so groups don't really survive outside of inventory and host matching. By default, Ansible overwrites variables including the ones defined for a group and/or host (see the `hash_merge` setting to change this). The order/precedence is (from lowest to highest):

- all group (because it is the 'parent' of all other groups)
- parent group
- child group
- host

When groups of the same parent/child level are merged, it is done alphabetically, and the last group loaded overwrites the previous groups. For example, an `a_group` will be merged with `b_group` and `b_group` vars that match will overwrite the ones in `a_group`.

2.4 .

Starting in Ansible version 2.4, users can use the group variable `ansible_group_priority` to change the merge order for groups of the same level (after the parent/child order is resolved). The larger the number, the later it will be merged, giving it higher priority. This variable defaults to 1 if not set. For example:

```
a_group:
  testvar: a
  ansible_group_priority: 10
b_group
  testvar: b
```

In this example, if both groups have the same priority, the result would normally have been `testvar == b`, but since we are giving the `a_group` a higher priority the result will be `testvar == a`.

## List of Behavioral Inventory Parameters

As described above, setting the following variables control how Ansible interacts with remote hosts.

Host connection:

---

: Ansible does not expose a channel to allow communication between the user and the ssh process to accept a password manually to decrypt an ssh key when using the ssh connection plugin (which is the default). The use of `ssh-agent` is highly recommended.

---

**ansible\_connection** Connection type to the host. This can be the name of any of ansible's connection plugins. SSH protocol types are `smart`, `ssh` or `paramiko`. The default is `smart`. Non-SSH based types are described in the next section.

General for all connections:

**ansible\_host** The name of the host to connect to, if different from the alias you wish to give to it.

**ansible\_port** The ssh port number, if not 22

**ansible\_user** The default ssh user name to use.

Specific to the SSH connection:

**ansible\_ssh\_pass** The ssh password to use (never store this variable in plain text; always use a vault. See [Variables and Vaults](#))

**ansible\_ssh\_private\_key\_file** Private key file used by ssh. Useful if using multiple keys and you don't want to use SSH agent.

**ansible\_ssh\_common\_args** This setting is always appended to the default command line for **sftp**, **scp**, and **ssh**. Useful to configure a `ProxyCommand` for a certain host (or group).

**ansible\_sftp\_extra\_args** This setting is always appended to the default **sftp** command line.

**ansible\_scp\_extra\_args** This setting is always appended to the default **scp** command line.

**ansible\_ssh\_extra\_args** This setting is always appended to the default **ssh** command line.

**ansible\_ssh\_pipelining** Determines whether or not to use SSH pipelining. This can override the `pipelining` setting in `ansible.cfg`.

**ansible\_ssh\_executable** (added in version 2.2) This setting overrides the default behavior to use the system **ssh**. This can override the `ssh_executable` setting in `ansible.cfg`.

Privilege escalation (see [Ansible Privilege Escalation](#) for further details):

**ansible\_become** Equivalent to `ansible_sudo` or `ansible_su`, allows to force privilege escalation

**ansible\_become\_method** Allows to set privilege escalation method

**ansible\_become\_user** Equivalent to `ansible_sudo_user` or `ansible_su_user`, allows to set the user you become through privilege escalation

**ansible\_become\_pass** Equivalent to `ansible_sudo_pass` or `ansible_su_pass`, allows you to set the privilege escalation password (never store this variable in plain text; always use a vault. See [Variables and Vaults](#))

**ansible\_become\_exe** Equivalent to `ansible_sudo_exe` or `ansible_su_exe`, allows you to set the executable for the escalation method selected

**ansible\_become\_flags** Equivalent to `ansible_sudo_flags` or `ansible_su_flags`, allows you to set the flags passed to the selected escalation method. This can be also set globally in `ansible.cfg` in the `sudo_flags` option

Remote host environment parameters:

**ansible\_shell\_type** The shell type of the target system. You should not use this setting unless you have set the [ansible\\_shell\\_executable](#) to a non-Bourne (sh) compatible shell. By default commands are formatted using sh-style syntax. Setting this to `csh` or `fish` will cause commands executed on target systems to follow those shell's syntax instead.

**ansible\_python\_interpreter** The target host python path. This is useful for systems with more than one Python or not located at `/usr/bin/python` such as \*BSD, or where `/usr/bin/python` is not a 2.X series Python. We do not use the `/usr/bin/env` mechanism as that requires the remote user's path to be set right and

also assumes the **python** executable is named `python`, where the executable might be named something like **python2.6**.

**ansible\_\*\_interpreter** Works for anything such as ruby or perl and works just like *ansible\_python\_interpreter*. This replaces shebang of modules which will run on that host.

2.1 .

**ansible\_shell\_executable** This sets the shell the ansible controller will use on the target machine, overrides `executable` in `ansible.cfg` which defaults to **/bin/sh**. You should really only change it if is not possible to use **/bin/sh** (i.e. **/bin/sh** is not installed on the target machine or cannot be run from `sudo`).

Examples from an Ansible-INI host file:

```
some_host      ansible_port=2222      ansible_user=manager
aws_host       ansible_ssh_private_key_file=/home/example/.ssh/aws.pem
freebsd_host   ansible_python_interpreter=/usr/local/bin/python
ruby_module_host ansible_ruby_interpreter=/usr/bin/ruby.1.9.3
```

## Non-SSH connection types

As stated in the previous section, Ansible executes playbooks over SSH but it is not limited to this connection type. With the host specific parameter `ansible_connection=<connector>`, the connection type can be changed. The following non-SSH based connectors are available:

### local

This connector can be used to deploy the playbook to the control machine itself.

### docker

This connector deploys the playbook directly into Docker containers using the local Docker client. The following parameters are processed by this connector:

**ansible\_host** The name of the Docker container to connect to.

**ansible\_user** The user name to operate within the container. The user must exist inside the container.

**ansible\_become** If set to `true` the `become_user` will be used to operate within the container.

**ansible\_docker\_extra\_args** Could be a string with any additional arguments understood by Docker, which are not command specific. This parameter is mainly used to configure a remote Docker daemon to use.

Here is an example of how to instantly deploy to created containers:

```
- name: create jenkins container
  docker_container:
    docker_host: myserver.net:4243
    name: my_jenkins
    image: jenkins

- name: add container to inventory
  add_host:
    name: my_jenkins
    ansible_connection: docker
    ansible_docker_extra_args: "--tlsverify --tlscacert=/path/to/ca.pem --tlscert=/
    ↪path/to/client-cert.pem --tlskey=/path/to/client-key.pem -H=tcp://myserver.net:4243"
    ansible_user: jenkins
    changed_when: false
```

(continues on next page)

```
- name: create directory for ssh keys
  delegate_to: my_jenkins
  file:
    path: "/var/jenkins_home/.ssh/jupiter"
    state: directory
```

: If you're reading the docs from the beginning, this may be the first example you've seen of an Ansible playbook. This is not an inventory file. Playbooks will be covered in great detail later in the docs.

:

**Working With Dynamic Inventory** Pulling inventory from dynamic sources, such as cloud providers

**Introduction To Ad-Hoc Commands** Examples of basic commands

**Working With Playbooks** Learning Ansible's configuration, deployment, and orchestration language.

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**irc.freenode.net** #ansible IRC chat channel

## 1.4.6 Working With Dynamic Inventory

### Topics

- *Working With Dynamic Inventory*
  - *Example: The Cobbler External Inventory Script*
  - *Example: AWS EC2 External Inventory Script*
  - *Example: OpenStack External Inventory Script*
    - \* *Explicit use of inventory script*
    - \* *Implicit use of inventory script*
    - \* *Refresh the cache*
  - *Other inventory scripts*
  - *Using Inventory Directories and Multiple Inventory Sources*
  - *Static Groups of Dynamic Groups*

Often a user of a configuration management system will want to keep inventory in a different software system. Ansible provides a basic text-based system as described in *Working with Inventory* but what if you want to use something else?

Frequent examples include pulling inventory from a cloud provider, LDAP, [Cobbler](#), or a piece of expensive enterprise CMDB software.

Ansible easily supports all of these options via an external inventory system. The contrib/inventory directory contains some of these already – including options for EC2/Eucalyptus, Rackspace Cloud, and OpenStack, examples of some of which will be detailed below.

*Ansible Tower* also provides a database to store inventory results that is both web and REST Accessible. Tower syncs with all Ansible dynamic inventory sources you might be using, and also includes a graphical inventory editor. By

having a database record of all of your hosts, it's easy to correlate past event history and see which ones have had failures on their last playbook runs.

For information about writing your own dynamic inventory source, see *Developing Dynamic Inventory*.

### Example: The Cobbler External Inventory Script

It is expected that many Ansible users with a reasonable amount of physical hardware may also be [Cobbler](#) users. (note: Cobbler was originally written by Michael DeHaan and is now led by James Cammarata, who also works for Ansible, Inc).

While primarily used to kickoff OS installations and manage DHCP and DNS, Cobbler has a generic layer that allows it to represent data for multiple configuration management systems (even at the same time), and has been referred to as a 'lightweight CMDB' by some admins.

To tie Ansible's inventory to Cobbler (optional), copy [this script](#) to `/etc/ansible` and `chmod +x` the file. `cobblerd` will now need to be running when you are using Ansible and you'll need to use Ansible's `-i` command line option (e.g. `-i /etc/ansible/cobbler.py`). This particular script will communicate with Cobbler using Cobbler's XMLRPC API.

Also a `cobbler.ini` file should be added to `/etc/ansible` so Ansible knows where the Cobbler server is and some cache improvements can be used. For example:

```
[cobbler]

# Set Cobbler's hostname or IP address
host = http://127.0.0.1/cobbler_api

# API calls to Cobbler can be slow. For this reason, we cache the results of an API
# call. Set this to the path you want cache files to be written to. Two files
# will be written to this directory:
#   - ansible-cobbler.cache
#   - ansible-cobbler.index

cache_path = /tmp

# The number of seconds a cache file is considered valid. After this many
# seconds, a new API call will be made, and the cache file will be updated.

cache_max_age = 900
```

First test the script by running `/etc/ansible/cobbler.py` directly. You should see some JSON data output, but it may not have anything in it just yet.

Let's explore what this does. In Cobbler, assume a scenario somewhat like the following:

```
cobbler profile add --name=webserver --distro=CentOS6-x86_64
cobbler profile edit --name=webserver --mgmt-classes="webserver" --ksmeta="a=2 b=3"
cobbler system edit --name=foo --dns-name="foo.example.com" --mgmt-classes="atlanta" -
↪-ksmeta="c=4"
cobbler system edit --name=bar --dns-name="bar.example.com" --mgmt-classes="atlanta" -
↪-ksmeta="c=5"
```

In the example above, the system 'foo.example.com' will be addressable by ansible directly, but will also be addressable when using the group names 'webserver' or 'atlanta'. Since Ansible uses SSH, we'll try to contact system foo over 'foo.example.com', only, never just 'foo'. Similarly, if you try "ansible foo" it wouldn't find the system... but "ansible 'foo\*'" would, because the system DNS name starts with 'foo'.

The script doesn't just provide host and group info. In addition, as a bonus, when the 'setup' module is run (which happens automatically when using playbooks), the variables 'a', 'b', and 'c' will all be auto-populated in the templates:

```
# file: /srv/motd.j2
Welcome, I am templated with a value of a={{ a }}, b={{ b }}, and c={{ c }}
```

Which could be executed just like this:

```
ansible webserver -m setup
ansible webserver -m template -a "src=/tmp/motd.j2 dest=/etc/motd"
```

: The name 'webserver' came from Cobbler, as did the variables for the config file. You can still pass in your own variables like normal in Ansible, but variables from the external inventory script will override any that have the same name.

So, with the template above (motd.j2), this would result in the following data being written to /etc/motd for system 'foo':

```
Welcome, I am templated with a value of a=2, b=3, and c=4
```

And on system 'bar' (bar.example.com):

```
Welcome, I am templated with a value of a=2, b=3, and c=5
```

And technically, though there is no major good reason to do it, this also works too:

```
ansible webserver -m shell -a "echo {{ a }}"
```

So in other words, you can use those variables in arguments/actions as well.

### Example: AWS EC2 External Inventory Script

If you use Amazon Web Services EC2, maintaining an inventory file might not be the best approach, because hosts may come and go over time, be managed by external applications, or you might even be using AWS autoscaling. For this reason, you can use the [EC2 external inventory script](#).

You can use this script in one of two ways. The easiest is to use Ansible's `-i` command line option and specify the path to the script after marking it executable:

```
ansible -i ec2.py -u ubuntu us-east-1d -m ping
```

The second option is to copy the script to `/etc/ansible/hosts` and `chmod +x` it. You will also need to copy the `ec2.ini` file to `/etc/ansible/ec2.ini`. Then you can run ansible as you would normally.

To successfully make an API call to AWS, you will need to configure Boto (the Python interface to AWS). There are a [variety of methods](#) available, but the simplest is just to export two environment variables:

```
export AWS_ACCESS_KEY_ID='AK123'
export AWS_SECRET_ACCESS_KEY='abc123'
```

You can test the script by itself to make sure your config is correct:

```
cd contrib/inventory
./ec2.py --list
```



After a few moments, you should see your entire EC2 inventory across all regions in JSON.

If you use Boto profiles to manage multiple AWS accounts, you can pass `--profile PROFILE` name to the `ec2.py` script. An example profile might be:

```
[profile dev]
aws_access_key_id = <dev access key>
aws_secret_access_key = <dev secret key>

[profile prod]
aws_access_key_id = <prod access key>
aws_secret_access_key = <prod secret key>
```

You can then run `ec2.py --profile prod` to get the inventory for the prod account, although this option is not supported by `ansible-playbook`. You can also use the `AWS_PROFILE` variable - for example: `AWS_PROFILE=prod ansible-playbook -i ec2.py myplaybook.yml`

Since each region requires its own API call, if you are only using a small set of regions, you can edit the `ec2.ini` file and comment out the regions you are not using.

There are other config options in `ec2.ini`, including cache control and destination variables. By default, the `ec2.ini` file is configured for **all Amazon cloud services**, but you can comment out any features that aren't applicable. For example, if you don't have RDS or elasticache, you can set them to `False`

```
[ec2]
...

# To exclude RDS instances from the inventory, uncomment and set to False.
rds = False

# To exclude ElastiCache instances from the inventory, uncomment and set to False.
elasticache = False
...
```

At their heart, inventory files are simply a mapping from some name to a destination address. The default `ec2.ini` settings are configured for running Ansible from outside EC2 (from your laptop for example) – and this is not the most efficient way to manage EC2.

If you are running Ansible from within EC2, internal DNS names and IP addresses may make more sense than public DNS names. In this case, you can modify the `destination_variable` in `ec2.ini` to be the private DNS name of an instance. This is particularly important when running Ansible within a private subnet inside a VPC, where the only way to access an instance is via its private IP address. For VPC instances, `vpc_destination_variable` in `ec2.ini` provides a means of using which ever `boto.ec2.instance` variable makes the most sense for your use case.

The EC2 external inventory provides mappings to instances from several groups:

**Global** All instances are in group `ec2`.

**Instance ID** These are groups of one since instance IDs are unique. e.g. `i-00112233 i-a1b1c1d1`

**Region** A group of all instances in an AWS region. e.g. `us-east-1 us-west-2`

**Availability Zone** A group of all instances in an availability zone. e.g. `us-east-1a us-east-1b`

**Security Group** Instances belong to one or more security groups. A group is created for each security group, with all characters except alphanumerics, converted to underscores (`_`). Each group is prefixed by `security_group_`. Currently, dashes (`-`) are also converted to underscores (`_`). You can change using the `replace_dash_in_groups` setting in `ec2.ini` (this has changed across several versions so check the `ec2.ini` for details). e.g. `security_group_default security_group_webservers security_group_Pete_s_Fancy_Group`

**Tags** Each instance can have a variety of key/value pairs associated with it called Tags. The most common tag key is 'Name', though anything is possible. Each key/value pair is its own group of instances, again with special characters converted to underscores, in the format tag\_KEY\_VALUE e.g. tag\_Name\_Web can be used as is tag\_Name\_redis-master-001 becomes tag\_Name\_redis\_master\_001 tag\_aws\_cloudformation\_logical-id\_WebServerGroup becomes tag\_aws\_cloudformation\_logical\_id\_WebServerGroup

When the Ansible is interacting with a specific server, the EC2 inventory script is called again with the `--host HOST` option. This looks up the HOST in the index cache to get the instance ID, and then makes an API call to AWS to get information about that specific instance. It then makes information about that instance available as variables to your playbooks. Each variable is prefixed by `ec2_`. Here are some of the variables available:

- `ec2_architecture`
- `ec2_description`
- `ec2_dns_name`
- `ec2_id`
- `ec2_image_id`
- `ec2_instance_type`
- `ec2_ip_address`
- `ec2_kernel`
- `ec2_key_name`
- `ec2_launch_time`
- `ec2_monitored`
- `ec2_ownerId`
- `ec2_placement`
- `ec2_platform`
- `ec2_previous_state`
- `ec2_private_dns_name`
- `ec2_private_ip_address`
- `ec2_public_dns_name`
- `ec2_ramdisk`
- `ec2_region`
- `ec2_root_device_name`
- `ec2_root_device_type`
- `ec2_security_group_ids`
- `ec2_security_group_names`
- `ec2_spot_instance_request_id`
- `ec2_state`
- `ec2_state_code`
- `ec2_state_reason`
- `ec2_status`

- `ec2_subnet_id`
- `ec2_tag_Name`
- `ec2_tenancy`
- `ec2_virtualization_type`
- `ec2_vpc_id`

Both `ec2_security_group_ids` and `ec2_security_group_names` are comma-separated lists of all security groups. Each EC2 tag is a variable in the format `ec2_tag_KEY`.

To see the complete list of variables available for an instance, run the script by itself:

```
cd contrib/inventory
./ec2.py --host ec2-12-12-12-12.compute-1.amazonaws.com
```

Note that the AWS inventory script will cache results to avoid repeated API calls, and this cache setting is configurable in `ec2.ini`. To explicitly clear the cache, you can run the `ec2.py` script with the `--refresh-cache` parameter:

```
./ec2.py --refresh-cache
```

### Example: OpenStack External Inventory Script

If you use an OpenStack based cloud, instead of manually maintaining your own inventory file, you can use the `openstack.py` dynamic inventory to pull information about your compute instances directly from OpenStack.

You can download the latest version of the OpenStack inventory script [here](#).

You can use the inventory script explicitly (by passing the `-i openstack.py` argument to Ansible) or implicitly (by placing the script at `/etc/ansible/hosts`).

### Explicit use of inventory script

Download the latest version of the OpenStack dynamic inventory script and make it executable:

```
wget https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/
↪openstack.py
chmod +x openstack.py
```

Source an OpenStack RC file:

```
source openstack.rc
```

---

: An OpenStack RC file contains the environment variables required by the client tools to establish a connection with the cloud provider, such as the authentication URL, user name, password and region name. For more information on how to download, create or source an OpenStack RC file, please refer to [Set environment variables using the OpenStack RC file](#).

---

You can confirm the file has been successfully sourced by running a simple command, such as `nova list` and ensuring it return no errors.

: The OpenStack command line clients are required to run the *nova list* command. For more information on how to install them, please refer to [Install the OpenStack command-line clients](#).

---

You can test the OpenStack dynamic inventory script manually to confirm it is working as expected:

```
./openstack.py --list
```

After a few moments you should see some JSON output with information about your compute instances.

Once you confirm the dynamic inventory script is working as expected, you can tell Ansible to use the *openstack.py* script as an inventory file, as illustrated below:

```
ansible -i openstack.py all -m ping
```

### Implicit use of inventory script

Download the latest version of the OpenStack dynamic inventory script, make it executable and copy it to */etc/ansible/hosts*:

```
wget https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/  
↪openstack_inventory.py  
chmod +x openstack.py  
sudo cp openstack.py /etc/ansible/hosts
```

Download the sample configuration file, modify it to suit your needs and copy it to */etc/ansible/openstack.yml*:

```
wget https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/  
↪openstack.yml  
vi openstack.yml  
sudo cp openstack.yml /etc/ansible/
```

You can test the OpenStack dynamic inventory script manually to confirm it is working as expected:

```
/etc/ansible/hosts --list
```

After a few moments you should see some JSON output with information about your compute instances.

### Refresh the cache

Note that the OpenStack dynamic inventory script will cache results to avoid repeated API calls. To explicitly clear the cache, you can run the *openstack.py* (or *hosts*) script with the *--refresh* parameter:

```
./openstack.py --refresh --list
```

### Other inventory scripts

In addition to Cobbler and EC2, inventory scripts are also available for:

```
BSD Jails  
DigitalOcean  
Google Compute Engine
```

(continues on next page)

()

```

Linode
OpenShift
OpenStack Nova
Ovirt
SpaceWalk
Vagrant (not to be confused with the provisioner in vagrant, which is preferred)
Zabbix

```

Sections on how to use these in more detail will be added over time, but by looking at the "contrib/inventory" directory of the Ansible checkout it should be very obvious how to use them. The process for the AWS inventory script is the same.

If you develop an interesting inventory script that might be general purpose, please submit a pull request – we'd likely be glad to include it in the project.

## Using Inventory Directories and Multiple Inventory Sources

If the location given to `-i` in Ansible is a directory (or as so configured in `ansible.cfg`), Ansible can use multiple inventory sources at the same time. When doing so, it is possible to mix both dynamic and statically managed inventory sources in the same ansible run. Instant hybrid cloud!

In an inventory directory, executable files will be treated as dynamic inventory sources and most other files as static sources. Files which end with any of the following will be ignored:

```
~, .orig, .bak, .ini, .cfg, .retry, .pyc, .pyo
```

You can replace this list with your own selection by configuring an `inventory_ignore_extensions` list in `ansible.cfg`, or setting the `ANSIBLE_INVENTORY_IGNORE` environment variable. The value in either case should be a comma-separated list of patterns, as shown above.

Any `group_vars` and `host_vars` subdirectories in an inventory directory will be interpreted as expected, making inventory directories a powerful way to organize different sets of configurations.

## Static Groups of Dynamic Groups

When defining groups of groups in the static inventory file, the child groups must also be defined in the static inventory file, or ansible will return an error. If you want to define a static group of dynamic child groups, define the dynamic groups as empty in the static inventory file. For example:

```

[tag_Name_staging_foo]

[tag_Name_staging_bar]

[staging:children]
tag_Name_staging_foo
tag_Name_staging_bar

```

:

**Working with Inventory** All about static inventory files

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**irc.freenode.net** #ansible IRC chat channel

## 1.4.7 Working With Playbooks

Playbooks are Ansible's configuration, deployment, and orchestration language. They can describe a policy you want your remote systems to enforce, or a set of steps in a general IT process.

If Ansible modules are the tools in your workshop, playbooks are your instruction manuals, and your inventory of hosts are your raw material.

At a basic level, playbooks can be used to manage configurations of and deployments to remote machines. At a more advanced level, they can sequence multi-tier rollouts involving rolling updates, and can delegate actions to other hosts, interacting with monitoring servers and load balancers along the way.

While there's a lot of information here, there's no need to learn everything at once. You can start small and pick up more features over time as you need them.

Playbooks are designed to be human-readable and are developed in a basic text language. There are multiple ways to organize playbooks and the files they include, and we'll offer up some suggestions on that and making the most out of Ansible.

You should look at [Example Playbooks](#) while reading along with the playbook documentation. These illustrate best practices as well as how to put many of the various concepts together.

### Intro to Playbooks

#### About Playbooks

Playbooks are a completely different way to use ansible than in adhoc task execution mode, and are particularly powerful.

Simply put, playbooks are the basis for a really simple configuration management and multi-machine deployment system, unlike any that already exist, and one that is very well suited to deploying complex applications.

Playbooks can declare configurations, but they can also orchestrate steps of any manual ordered process, even as different steps must bounce back and forth between sets of machines in particular orders. They can launch tasks synchronously or asynchronously.

While you might run the main `/usr/bin/ansible` program for ad-hoc tasks, playbooks are more likely to be kept in source control and used to push out your configuration or assure the configurations of your remote systems are in spec.

There are also some full sets of playbooks illustrating a lot of these techniques in the [ansible-examples repository](#). We'd recommend looking at these in another tab as you go along.

There are also many jumping off points after you learn playbooks, so hop back to the documentation index after you're done with this section.

#### Playbook Language Example

Playbooks are expressed in YAML format (see [YAML Syntax](#)) and have a minimum of syntax, which intentionally tries to not be a programming language or script, but rather a model of a configuration or a process.

Each playbook is composed of one or more 'plays' in a list.

The goal of a play is to map a group of hosts to some well defined roles, represented by things ansible calls tasks. At a basic level, a task is nothing more than a call to an ansible module (see [Working With Modules](#)).

By composing a playbook of multiple 'plays', it is possible to orchestrate multi-machine deployments, running certain steps on all machines in the webservers group, then certain steps on the database server group, then more commands back on the webservers group, etc.

"plays" are more or less a sports analogy. You can have quite a lot of plays that affect your systems to do different things. It's not as if you were just defining one particular state or model, and you can run different plays at different times.

For starters, here's a playbook that contains just one play:

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
    remote_user: root
  tasks:
    - name: ensure apache is at the latest version
      yum:
        name: httpd
        state: latest
    - name: write the apache config file
      template:
        src: /srv/httpd.j2
        dest: /etc/httpd.conf
      notify:
        - restart apache
    - name: ensure apache is running
      service:
        name: httpd
        state: started
  handlers:
    - name: restart apache
      service:
        name: httpd
        state: restarted
```

Playbooks can contain multiple plays. You may have a playbook that targets first the web servers, and then the database servers. For example:

```
---
- hosts: webservers
  remote_user: root

  tasks:
    - name: ensure apache is at the latest version
      yum:
        name: httpd
        state: latest
    - name: write the apache config file
      template:
        src: /srv/httpd.j2
        dest: /etc/httpd.conf

- hosts: databases
  remote_user: root

  tasks:
```

(continues on next page)

()

```
- name: ensure postgresql is at the latest version
  yum:
    name: postgresql
    state: latest
- name: ensure that postgresql is started
  service:
    name: postgresql
    state: started
```

You can use this method to switch between the host group you're targeting, the username logging into the remote servers, whether to sudo or not, and so forth. Plays, like tasks, run in the order specified in the playbook: top to bottom.

Below, we'll break down what the various features of the playbook language are.

## Basics

### Hosts and Users

For each play in a playbook, you get to choose which machines in your infrastructure to target and what remote user to complete the steps (called tasks) as.

The `hosts` line is a list of one or more groups or host patterns, separated by colons, as described in the [Working with Patterns](#) documentation. The `remote_user` is just the name of the user account:

```
---
- hosts: webservers
  remote_user: root
```

---

: The `remote_user` parameter was formerly called just `user`. It was renamed in Ansible 1.4 to make it more distinguishable from the **user** module (used to create users on remote systems).

---

Remote users can also be defined per task:

```
---
- hosts: webservers
  remote_user: root
  tasks:
    - name: test connection
      ping:
        remote_user: yourname
```

Support for running things as another user is also available (see [Understanding Privilege Escalation](#)):

```
---
- hosts: webservers
  remote_user: yourname
  become: yes
```

You can also use `become` on a particular task instead of the whole play:

```
---
- hosts: webservers
```

(continues on next page)



()

```
remote_user: yourname
tasks:
  - service:
      name: nginx
      state: started
      become: yes
      become_method: sudo
```

You can also login as you, and then become a user different than root:

```
---
- hosts: webserver
  remote_user: yourname
  become: yes
  become_user: postgres
```

You can also use other privilege escalation methods, like su:

```
---
- hosts: webserver
  remote_user: yourname
  become: yes
  become_method: su
```

If you need to specify a password to sudo, run `ansible-playbook` with `--ask-become-pass` or when using the old sudo syntax `--ask-sudo-pass (-K)`. If you run a become playbook and the playbook seems to hang, it's probably stuck at the privilege escalation prompt. Just *Control-C* to kill it and run it again adding the appropriate password.

---

: When using `become_user` to a user other than root, the module arguments are briefly written into a random tempfile in `/tmp`. These are deleted immediately after the command is executed. This only occurs when changing privileges from a user like 'bob' to 'timmy', not when going from 'bob' to 'root', or logging in directly as 'bob' or 'root'. If it concerns you that this data is briefly readable (not writable), avoid transferring unencrypted passwords with `become_user` set. In other cases, `/tmp` is not used and this does not come into play. Ansible also takes care to not log password parameters.

---

## 2.4 .

You can also control the order in which hosts are run. The default is to follow the order supplied by the inventory:

```
- hosts: all
  order: sorted
  gather_facts: False
  tasks:
    - debug:
        var: inventory_hostname
```

Possible values for order are:

**inventory:** The default. The order is 'as provided' by the inventory

**reverse\_inventory:** As the name implies, this reverses the order 'as provided' by the inventory

**sorted:** Hosts are alphabetically sorted by name

**reverse\_sorted:** Hosts are sorted by name in reverse alphabetical order

**shuffle:** Hosts are randomly ordered each run

### Tasks list

Each play contains a list of tasks. Tasks are executed in order, one at a time, against all machines matched by the host pattern, before moving on to the next task. It is important to understand that, within a play, all hosts are going to get the same task directives. It is the purpose of a play to map a selection of hosts to tasks.

When running the playbook, which runs top to bottom, hosts with failed tasks are taken out of the rotation for the entire playbook. If things fail, simply correct the playbook file and rerun.

The goal of each task is to execute a module, with very specific arguments. Variables, as mentioned above, can be used in arguments to modules.

Modules should be idempotent, that is, running a module multiple times in a sequence should have the same effect as running it just once. One way to achieve idempotency is to have a module check whether its desired final state has already been achieved, and if that state has been achieved, to exit without performing any actions. If all the modules a playbook uses are idempotent, then the playbook itself is likely to be idempotent, so re-running the playbook should be safe.

The **command** and **shell** modules will typically rerun the same command again, which is totally ok if the command is something like `chmod` or `setsebool`, etc. Though there is a `creates` flag available which can be used to make these modules also idempotent.

Every task should have a `name`, which is included in the output from running the playbook. This is human readable output, and so it is useful to provide good descriptions of each task step. If the name is not provided though, the string fed to 'action' will be used for output.

Tasks can be declared using the legacy `action: module options` format, but it is recommended that you use the more conventional `module: options` format. This recommended format is used throughout the documentation, but you may encounter the older format in some playbooks.

Here is what a basic task looks like. As with most modules, the `service` module takes `key=value` arguments:

```
tasks:
- name: make sure apache is running
  service:
    name: httpd
    state: started
```

The **command** and **shell** modules are the only modules that just take a list of arguments and don't use the `key=value` form. This makes them work as simply as you would expect:

```
tasks:
- name: enable selinux
  command: /sbin/setenforce 1
```

The **command** and **shell** module care about return codes, so if you have a command whose successful exit code is not zero, you may wish to do this:

```
tasks:
- name: run this command and ignore the result
  shell: /usr/bin/somecommand || /bin/true
```

Or this:

```
tasks:
  - name: run this command and ignore the result
    shell: /usr/bin/somecommand
    ignore_errors: True
```

If the action line is getting too long for comfort you can break it on a space and indent any continuation lines:

```
tasks:
  - name: Copy ansible inventory file to client
    copy: src=/etc/ansible/hosts dest=/etc/ansible/hosts
          owner=root group=root mode=0644
```

Variables can be used in action lines. Suppose you defined a variable called `vhost` in the `vars` section, you could do this:

```
tasks:
  - name: create a virtual host file for {{ vhost }}
    template:
      src: somefile.j2
      dest: /etc/httpd/conf.d/{{ vhost }}
```

Those same variables are usable in templates, which we'll get to later.

Now in a very basic playbook all the tasks will be listed directly in that play, though it will usually make more sense to break up tasks as described in [Creating Reusable Playbooks](#).

## Action Shorthand

0.8 .

Ansible prefers listing modules like this:

```
template:
  src: templates/foo.j2
  dest: /etc/foo.conf
```

Early versions of Ansible used the following format, which still works:

```
action: template src=templates/foo.j2 dest=/etc/foo.conf
```

## Handlers: Running Operations On Change

As we've mentioned, modules should be idempotent and can relay when they have made a change on the remote system. Playbooks recognize this and have a basic event system that can be used to respond to change.

These 'notify' actions are triggered at the end of each block of tasks in a play, and will only be triggered once even if notified by multiple different tasks.

For instance, multiple resources may indicate that apache needs to be restarted because they have changed a config file, but apache will only be bounced once to avoid unnecessary restarts.

Here's an example of restarting two services when the contents of a file change, but only if the file changes:

```
- name: template configuration file
  template:
    src: template.j2
    dest: /etc/foo.conf
  notify:
    - restart memcached
    - restart apache
```

The things listed in the `notify` section of a task are called handlers.

Handlers are lists of tasks, not really any different from regular tasks, that are referenced by a globally unique name, and are notified by notifiers. If nothing notifies a handler, it will not run. Regardless of how many tasks notify a handler, it will run only once, after all of the tasks complete in a particular play.

Here's an example handlers section:

```
handlers:
  - name: restart memcached
    service:
      name: memcached
      state: restarted
  - name: restart apache
    service:
      name: apache
      state: restarted
```

As of Ansible 2.2, handlers can also "listen" to generic topics, and tasks can notify those topics as follows:

```
handlers:
  - name: restart memcached
    service:
      name: memcached
      state: restarted
    listen: "restart web services"
  - name: restart apache
    service:
      name: apache
      state: restarted
    listen: "restart web services"

tasks:
  - name: restart everything
    command: echo "this task will restart the web services"
    notify: "restart web services"
```

This use makes it much easier to trigger multiple handlers. It also decouples handlers from their names, making it easier to share handlers among playbooks and roles (especially when using 3rd party roles from a shared source like Galaxy).

---

:

- Notify handlers are always run in the same order they are defined, *not* in the order listed in the `notify`-statement. This is also the case for handlers using *listen*.
- Handler names and *listen* topics live in a global namespace.
- If two handler tasks have the same name, only one will run. \*

- You cannot notify a handler that is defined inside of an include. As of Ansible 2.1, this does work, however the include must be *static*.

Roles are described later on, but it's worthwhile to point out that:

- handlers notified within `pre_tasks`, `tasks`, and `post_tasks` sections are automatically flushed in the end of section where they were notified;
- handlers notified within `roles` section are automatically flushed in the end of `tasks` section, but before any `tasks` handlers.

If you ever want to flush all the handler commands immediately you can do this:

```
tasks:
  - shell: some tasks go here
  - meta: flush_handlers
  - shell: some other tasks
```

In the above example any queued up handlers would be processed early when the `meta` statement was reached. This is a bit of a niche case but can come in handy from time to time.

## Executing A Playbook

Now that you've learned playbook syntax, how do you run a playbook? It's simple. Let's run a playbook using a parallelism level of 10:

```
ansible-playbook playbook.yml -f 10
```

## Ansible-Pull

Should you want to invert the architecture of Ansible, so that nodes check in to a central location, instead of pushing configuration out to them, you can.

The `ansible-pull` is a small script that will checkout a repo of configuration instructions from git, and then run `ansible-playbook` against that content.

Assuming you load balance your checkout location, `ansible-pull` scales essentially infinitely.

Run `ansible-pull --help` for details.

There's also a [clever playbook](#) available to configure `ansible-pull` via a crontab from push mode.

## Tips and Tricks

To check the syntax of a playbook, use `ansible-playbook` with the `--syntax-check` flag. This will run the playbook file through the parser to ensure its included files, roles, etc. have no syntax problems.

Look at the bottom of the playbook execution for a summary of the nodes that were targeted and how they performed. General failures and fatal "unreachable" communication attempts are kept separate in the counts.

If you ever want to see detailed output from successful modules as well as unsuccessful ones, use the `--verbose` flag. This is available in Ansible 0.5 and later.

To see what hosts would be affected by a playbook before you run it, you can do this:

```
ansible-playbook playbook.yml --list-hosts
```

:

*YAML Syntax* Learn about YAML syntax

*Best Practices* Various tips about managing playbooks in the real world

**all\_modules** Learn about available modules

*Developing Modules* Learn how to extend Ansible by writing your own modules

*Working with Patterns* Learn about how to select hosts

**Github examples directory** Complete end-to-end playbook examples

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

## Creating Reusable Playbooks

### Including and Importing

#### Topics

- *Including and Importing*
  - *Includes vs. Imports*
  - *Importing Playbooks*
  - *Including and Importing Task Files*
  - *Including and Importing Roles*

### Includes vs. Imports

As noted in *Creating Reusable Playbooks*, include and import statements are very similar, however the Ansible executor engine treats them very differently.

- All `import*` statements are pre-processed at the time playbooks are parsed.
- All `include*` statements are processed as they encountered during the execution of the playbook.

Please refer to *Creating Reusable Playbooks* for documentation concerning the trade-offs one may encounter when using each type.

Also be aware that this behaviour changed in 2.4. Prior to Ansible 2.4, only `include` was available and it behaved differently depending on context.

2.4 .

### Importing Playbooks

It is possible to include playbooks inside a master playbook. For example:

```
- import_playbook: webservers.yml
- import_playbook: databases.yml
```

The plays and tasks in each playbook listed will be run in the order they are listed, just as if they had been defined here directly.

Prior to 2.4 only `include` was available and worked for both playbooks and tasks as both `import` and `include`.

2.4 .

## Including and Importing Task Files

Breaking tasks up into different files is an excellent way to organize complex sets of tasks or reuse them. A task file simply contains a flat list of tasks:

```
# common_tasks.yml
- name: placeholder foo
  command: /bin/foo
- name: placeholder bar
  command: /bin/bar
```

You can then use `import_tasks` or `include_tasks` to execute the tasks in a file in the main task list:

```
tasks:
- import_tasks: common_tasks.yml
# or
- include_tasks: common_tasks.yml
```

You can also pass variables into imports and includes:

```
tasks:
- import_tasks: wordpress.yml
  vars:
    wp_user: timmy
- import_tasks: wordpress.yml
  vars:
    wp_user: alice
- import_tasks: wordpress.yml
  vars:
    wp_user: bob
```

See [Variable Precedence: Where Should I Put A Variable?](#) for more details on variable inheritance and precedence.

Task `include` and `import` statements can be used at arbitrary depth.

---

:

- Static and dynamic can be mixed, however this is not recommended as it may lead to difficult-to-diagnose bugs in your playbooks.
  - The `key=value` syntax for passing variables to `import` and `include` is deprecated. Use `YAML vars:` instead.
- 

Includes and imports can also be used in the `handlers:` section. For instance, if you want to define how to restart Apache, you only have to do that once for all of your playbooks. You might make a `handlers.yml` that looks like:

```
# more_handlers.yml
- name: restart apache
  service: name=apache state=restarted
```

And in your main playbook file:

```
handlers:
- include_tasks: more_handlers.yml
# or
- import_tasks: more_handlers.yml
```

---

: Be sure to refer to the limitations/trade-offs for handlers noted in *Creating Reusable Playbooks*.

---

You can mix in includes along with your regular non-included tasks and handlers.

### Including and Importing Roles

Please refer to *Roles* for details on including and importing roles.

:

**YAML Syntax** Learn about YAML syntax

**Working With Playbooks** Review the basic Playbook language features

**Best Practices** Various tips about managing playbooks in the real world

**Variables** All about variables in playbooks

**Conditionals** Conditionals in playbooks

**Loops** Loops in playbooks

**all\_modules** Learn about available modules

**Developing Modules** Learn how to extend Ansible by writing your own modules

**GitHub Ansible examples** Complete playbook files from the GitHub project source

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

### Roles

#### Topics

- *Roles*
  - *Role Directory Structure*
  - *Using Roles*
  - *Role Duplication and Execution*
  - *Role Default Variables*
  - *Role Dependencies*



- *Embedding Modules and Plugins In Roles*
- *Role Search Path*
- *Ansible Galaxy*

## 1.2 .

Roles are ways of automatically loading certain vars\_files, tasks, and handlers based on a known file structure. Grouping content by roles also allows easy sharing of roles with other users.

## Role Directory Structure

Example project structure:

```
site.yml
webservers.yml
fooservers.yml
roles/
  common/
    tasks/
    handlers/
    files/
    templates/
    vars/
    defaults/
    meta/
  webservers/
    tasks/
    defaults/
    meta/
```

Roles expect files to be in certain directory names. Roles must include at least one of these directories, however it is perfectly fine to exclude any which are not being used. When in use, each directory must contain a `main.yml` file, which contains the relevant content:

- `tasks` - contains the main list of tasks to be executed by the role.
- `handlers` - contains handlers, which may be used by this role or even anywhere outside this role.
- `defaults` - default variables for the role (see [Variables](#) for more information).
- `vars` - other variables for the role (see [Variables](#) for more information).
- `files` - contains files which can be deployed via this role.
- `templates` - contains templates which can be deployed via this role.
- `meta` - defines some meta data for this role. See below for more details.

Other YAML files may be included in certain directories. For example, it is common practice to have platform-specific tasks included from the `tasks/main.yml` file:

```
# roles/example/tasks/main.yml
- name: added in 2.4, previously you used 'include'
  import_tasks: redhat.yml
  when: ansible_os_platform|lower == 'redhat'
- import_tasks: debian.yml
  when: ansible_os_platform|lower == 'debian'
```

(continues on next page)

```
# roles/example/tasks/redhat.yml
- yum:
    name: "httpd"
    state: present

# roles/example/tasks/debian.yml
- apt:
    name: "apache2"
    state: present
```

Roles may also include modules and other plugin types. For more information, please refer to the [Embedding Modules and Plugins In Roles](#) section below.

## Using Roles

The classic (original) way to use roles is via the `roles:` option for a given play:

```
---
- hosts: webservers
  roles:
    - common
    - webservers
```

This designates the following behaviors, for each role 'x':

- If `roles/x/tasks/main.yml` exists, tasks listed therein will be added to the play.
- If `roles/x/handlers/main.yml` exists, handlers listed therein will be added to the play.
- If `roles/x/vars/main.yml` exists, variables listed therein will be added to the play.
- If `roles/x/defaults/main.yml` exists, variables listed therein will be added to the play.
- If `roles/x/meta/main.yml` exists, any role dependencies listed therein will be added to the list of roles (1.3 and later).
- Any copy, script, template or include tasks (in the role) can reference files in `roles/x/{files,templates,tasks}/` (dir depends on task) without having to path them relatively or absolutely.

When used in this manner, the order of execution for your playbook is as follows:

- Any `pre_tasks` defined in the play.
- Any handlers triggered so far will be run.
- Each role listed in `roles` will execute in turn. Any role dependencies defined in the `roles meta/main.yml` will be run first, subject to tag filtering and conditionals.
- Any `tasks` defined in the play.
- Any handlers triggered so far will be run.
- Any `post_tasks` defined in the play.
- Any handlers triggered so far will be run.

---

: See below for more information regarding role dependencies.

---

---

: If using tags with tasks (described later as a means of only running part of a playbook), be sure to also tag your `pre_tasks`, `post_tasks`, and role dependencies and pass those along as well, especially if the pre/post tasks and role dependencies are used for monitoring outage window control or load balancing.

---

As of Ansible 2.4, you can now use roles inline with any other tasks using `import_role` or `include_role`:

```
---
- hosts: webserver
  tasks:
    - debug:
        msg: "before we run our role"
    - import_role:
        name: example
    - include_role:
        name: example
    - debug:
        msg: "after we ran our role"
```

When roles are defined in the classic manner, they are treated as static imports and processed during playbook parsing.

---

: The `include_role` option was introduced in Ansible 2.3. The usage has changed slightly as of Ansible 2.4 to match the `include` (dynamic) vs. `import` (static) usage. See *Dynamic vs. Static* for more details.

---

The name used for the role can be a simple name (see *Role Search Path* below), or it can be a fully qualified path:

```
---
- hosts: webserver
  roles:
    - role: '/path/to/my/roles/common'
```

Roles can accept other keywords:

```
---
- hosts: webserver
  roles:
    - common
    - role: foo_app_instance
      vars:
        dir: '/opt/a'
        app_port: 5000
    - role: foo_app_instance
      vars:
        dir: '/opt/b'
        app_port: 5001
```

Or, using the newer syntax:

```
---
- hosts: webserver
  tasks:
```

(continues on next page)

0

```
- include_role:
  name: foo_app_instance
vars:
  dir: '/opt/a'
  app_port: 5000
...
```

You can conditionally import a role and execute it's tasks:

```
---
- hosts: webservers
  tasks:
    - include_role:
      name: some_role
      when: "ansible_os_family == 'RedHat'"
```

Finally, you may wish to assign tags to the tasks inside the roles you specify. You can do:

```
---
- hosts: webservers
  roles:
    - role: bar
      tags: ["foo"]
      # using YAML shorthand, this is equivalent to the above
    - { role: foo, tags: ["bar", "baz"] }
```

Or, again, using the newer syntax:

```
---
- hosts: webservers
  tasks:
    - import_role:
      name: foo
      tags:
        - bar
        - baz
```

---

: This tags all of the tasks in that role with the tags specified, appending to any tags that are specified inside the role.

---

On the other hand you might just want to tag the import of the role itself:

```
- hosts: webservers
  tasks:
    - include_role:
      name: bar
      tags:
        - foo
```

---

: The tags in this example will *not* be added to tasks inside an `include_role`, you can use a surrounding `block` directive to do both.

---

---

: There is no facility to import a role while specifying a subset of tags to execute. If you find yourself building a role with lots of tags and you want to call subsets of the role at different times, you should consider just splitting that role into multiple roles.

---

## Role Duplication and Execution

Ansible will only allow a role to execute once, even if defined multiple times, if the parameters defined on the role are not different for each definition. For example:

```
---
- hosts: webserver
  roles:
    - foo
    - foo
```

Given the above, the role `foo` will only be run once.

To make roles run more than once, there are two options:

1. Pass different parameters in each role definition.
2. Add `allow_duplicates: true` to the `meta/main.yml` file for the role.

Example 1 - passing different parameters:

```
---
- hosts: webserver
  roles:
    - role: foo
      vars:
        message: "first"
    - { role: foo, vars: { message: "second" } }
```

In this example, because each role definition has different parameters, `foo` will run twice.

Example 2 - using `allow_duplicates: true`:

```
# playbook.yml
---
- hosts: webserver
  roles:
    - foo
    - foo

# roles/foo/meta/main.yml
---
allow_duplicates: true
```

In this example, `foo` will run twice because we have explicitly enabled it to do so.

## Role Default Variables

1.3 .

Role default variables allow you to set default variables for included or dependent roles (see below). To create defaults, simply add a `defaults/main.yml` file in your role directory. These variables will have the lowest priority of any variables available, and can be easily overridden by any other variable, including inventory variables.

## Role Dependencies

### 1.3.

Role dependencies allow you to automatically pull in other roles when using a role. Role dependencies are stored in the `meta/main.yml` file contained within the role directory, as noted above. This file should contain a list of roles and parameters to insert before the specified role, such as the following in an example `roles/myapp/meta/main.yml`:

```
---
dependencies:
  - role: common
    vars:
      some_parameter: 3
  - role: apache
    vars:
      apache_port: 80
  - role: postgres
    vars:
      dbname: blarg
      other_parameter: 12
```

---

: Role dependencies must use the classic role definition style.

---

Role dependencies are always executed before the role that includes them, and may be recursive. Dependencies also follow the duplication rules specified above. If another role also lists it as a dependency, it will not be run again based on the same rules given above.

---

: Always remember that when using `allow_duplicates: true`, it needs to be in the dependent role's `meta/main.yml`, not the parent.

---

For example, a role named `car` depends on a role named `wheel` as follows:

```
---
dependencies:
  - role: wheel
    vars:
      n: 1
  - role: wheel
    vars:
      n: 2
  - role: wheel
    vars:
      n: 3
  - role: wheel
    vars:
      n: 4
```

And the `wheel` role depends on two roles: `tire` and `brake`. The `meta/main.yml` for `wheel` would then contain the following:

```
---
dependencies:
- role: tire
- role: brake
```

And the meta/main.yml for tire and brake would contain the following:

```
---
allow_duplicates: true
```

The resulting order of execution would be as follows:

```
tire(n=1)
brake(n=1)
wheel(n=1)
tire(n=2)
brake(n=2)
wheel(n=2)
...
car
```

Note that we did not have to use `allow_duplicates: true` for wheel, because each instance defined by car uses different parameter values.

---

: Variable inheritance and scope are detailed in the [Variables](#).

---

## Embedding Modules and Plugins In Roles

This is an advanced topic that should not be relevant for most users.

If you write a custom module (see [Developing Modules](#)) or a plugin (see [Developing Plugins](#)), you may wish to distribute it as part of a role. Generally speaking, Ansible as a project is very interested in taking high-quality modules into ansible core for inclusion, so this shouldn't be the norm, but it's quite easy to do.

A good example for this is if you worked at a company called AcmeWidgets, and wrote an internal module that helped configure your internal software, and you wanted other people in your organization to easily use this module – but you didn't want to tell everyone how to configure their Ansible library path.

Alongside the 'tasks' and 'handlers' structure of a role, add a directory named 'library'. In this 'library' directory, then include the module directly inside of it.

Assuming you had this:

```
roles/
  my_custom_modules/
    library/
      module1
      module2
```

The module will be usable in the role itself, as well as any roles that are called *after* this role, as follows:

```
- hosts: webservers
  roles:
    - my_custom_modules
```

(continues on next page)

()

```
- some_other_role_using_my_custom_modules
- yet_another_role_using_my_custom_modules
```

This can also be used, with some limitations, to modify modules in Ansible's core distribution, such as to use development versions of modules before they are released in production releases. This is not always advisable as API signatures may change in core components, however, and is not always guaranteed to work. It can be a handy way of carrying a patch against a core module, however, should you have good reason for this. Naturally the project prefers that contributions be directed back to github whenever possible via a pull request.

The same mechanism can be used to embed and distribute plugins in a role, using the same schema. For example, for a filter plugin:

```
roles/
  my_custom_filter/
    filter_plugins
      filter1
      filter2
```

They can then be used in a template or a jinja template in any role called after 'my\_custom\_filter'

## Role Search Path

Ansible will search for roles in the following way:

- A `roles/` directory, relative to the playbook file.
- By default, in `/etc/ansible/roles`

In Ansible 1.4 and later you can configure an additional `roles_path` to search for roles. Use this to check all of your common roles out to one location, and share them easily between multiple playbook projects. See [Configuring Ansible](#) for details about how to set this up in `ansible.cfg`.

## Ansible Galaxy

[Ansible Galaxy](#) is a free site for finding, downloading, rating, and reviewing all kinds of community developed Ansible roles and can be a great way to get a jumpstart on your automation projects.

The client `ansible-galaxy` is included in Ansible. The Galaxy client allows you to download roles from Ansible Galaxy, and also provides an excellent default framework for creating your own roles.

Read the Ansible Galaxy documentation <<https://galaxy.ansible.com/docs/>>\_ page for more information

:

***Ansible Galaxy*** How to create new roles, share roles on Galaxy, role management

***YAML Syntax*** Learn about YAML syntax

***Working With Playbooks*** Review the basic Playbook language features

***Best Practices*** Various tips about managing playbooks in the real world

***Variables*** All about variables in playbooks

***Conditionals*** Conditionals in playbooks

***Loops*** Loops in playbooks

***all\_modules*** Learn about available modules



**Developing Modules** Learn how to extend Ansible by writing your own modules

**GitHub Ansible examples** Complete playbook files from the GitHub project source

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

While it is possible to write a playbook in one very large file (and you might start out learning playbooks this way), eventually you'll want to reuse files and start to organize things. In Ansible, there are three ways to do this: includes, imports, and roles.

Includes and imports (added in Ansible version 2.4) allow users to break up large playbooks into smaller files, which can be used across multiple parent playbooks or even multiple times within the same Playbook.

Roles allow more than just tasks to be packaged together and can include variables, handlers, or even modules and other plugins. Unlike includes and imports, roles can also be uploaded and shared via Ansible Galaxy.

## Dynamic vs. Static

Ansible has two modes of operation for reusable content: dynamic and static.

In Ansible 2.0, the concept of *dynamic* includes was introduced. Due to some limitations with making all includes dynamic in this way, the ability to force includes to be *static* was introduced in Ansible 2.1. Because the *include* task became overloaded to encompass both static and dynamic syntaxes, and because the default behavior of an include could change based on other options set on the Task, Ansible 2.4 introduces the concept of *include* vs. *import*.

If you use any *import\** Task (*import\_playbook*, *import\_tasks*, etc.), it will be *static*. If you use any *include\** Task (*include\_tasks*, *include\_role*, etc.), it will be *dynamic*.

The bare *include* task (which was used for both Task files and Playbook-level includes) is still available, however it is now considered *deprecated*.

## Differences Between Static and Dynamic

The two modes of operation are pretty simple:

- Ansible pre-processes all static imports during Playbook parsing time.
- Dynamic includes are processed during runtime at the point in which that task is encountered.

When it comes to Ansible task options like `tags` and conditional statements (`when:`):

- For static imports, the parent task options will be copied to all child tasks contained within the import.
- For dynamic includes, the task options will *only* apply to the dynamic task as it is evaluated, and will not be copied to child tasks.

---

: Roles are a somewhat special case. Prior to Ansible 2.3, roles were always statically included via the special `roles:` option for a given play and were always executed first before any other play tasks (unless `pre_tasks` were used). Roles can still be used this way, however, Ansible 2.3 introduced the `include_role` option to allow roles to be executed inline with other tasks.

---

## Tradeoffs and Pitfalls Between Includes and Imports

Using *include\** vs. *import\** has some advantages as well as some tradeoffs which users should consider when choosing to use each:

The primary advantage of using `include*` statements is looping. When a loop is used with an include, the included tasks or role will be executed once for each item in the loop.

Using `include*` does have some limitations when compared to `import*` statements:

- Tags which only exist inside a dynamic include will not show up in `--list-tags` output.
- Tasks which only exist inside a dynamic include will not show up in `--list-tasks` output.
- You cannot use `notify` to trigger a handler name which comes from inside a dynamic include (see note below).
- You cannot use `--start-at-task` to begin execution at a task inside a dynamic include.

Using `import*` can also have some limitations when compared to dynamic includes:

- As noted above, loops cannot be used with imports at all.
- When using variables for the target file or role name, variables from inventory sources (host/group vars, etc.) cannot be used.

---

: Regarding the use of `notify` for dynamic tasks: it is still possible to trigger the dynamic include itself, which would result in all tasks within the include being run.

---

:

**Working With Playbooks** Review the basic Playbook language features

**Variables** All about variables in playbooks

**Conditionals** Conditionals in playbooks

**Loops** Loops in playbooks

**Best Practices** Various tips about managing playbooks in the real world

**Ansible Galaxy** How to share roles on galaxy, role management

**GitHub Ansible examples** Complete playbook files from the GitHub project source

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

## Variables

### Topics

- *Variables*
  - *What Makes A Valid Variable Name*
  - *Variables Defined in Inventory*
  - *Variables Defined in a Playbook*
  - *Variables defined from included files and roles*
  - *Using Variables: About Jinja2*
  - *Jinja2 Filters*
  - *Hey Wait, A YAML Gotcha*
  - *Information discovered from systems: Facts*

- [Turning Off Facts](#)
- [Local Facts \(Facts.d\)](#)
- [Ansible version](#)
- [Fact Caching](#)
- [Registered Variables](#)
- [Accessing Complex Variable Data](#)
- [Magic Variables, and How To Access Information About Other Hosts](#)
- [Variable File Separation](#)
- [Passing Variables On The Command Line](#)
- [Variable Precedence: Where Should I Put A Variable?](#)
- [Variable Scopes](#)
- [Variable Examples](#)
- [Advanced Syntax](#)

While automation exists to make it easier to make things repeatable, all systems are not exactly alike; some may require configuration that is slightly different from others. In some instances, the observed behavior or state of one system might influence how you configure other systems. For example, you might need to find out the IP address of a system and use it as a configuration value on another system.

Ansible uses *variables* to help deal with differences between systems.

To understand variables you'll also want to read [Conditionals](#) and [Loops](#). Useful things like the **group\_by** module and the `when` conditional can also be used with variables, and to help manage differences between systems.

The `ansible-examples` github repository contains many examples of how variables are used in Ansible.

For advice on best practices, refer to [Variables and Vaults](#) in the *Best Practices* chapter.

## What Makes A Valid Variable Name

Before we start using variables, it's important to know what are valid variable names.

Variable names should be letters, numbers, and underscores. Variables should always start with a letter.

`foo_port` is a great variable. `foo5` is fine too.

`foo-port`, `foo port`, `foo.port` and `12` are not valid variable names.

YAML also supports dictionaries which map keys to values. For instance:

```
foo:
  field1: one
  field2: two
```

You can then reference a specific field in the dictionary using either bracket notation or dot notation:

```
foo['field1']
foo.field1
```

These will both reference the same value ("one"). However, if you choose to use dot notation be aware that some keys can cause problems because they collide with attributes and methods of python dictionaries. You should use bracket

notation instead of dot notation if you use keys which start and end with two underscores (Those are reserved for special meanings in python) or are any of the known public attributes:

add, append, as\_integer\_ratio, bit\_length, capitalize, center, clear, conjugate, copy, count, decode, denominator, difference, difference\_update, discard, encode, endswith, expandtabs, extend, find, format, fromhex, fromkeys, get, has\_key, hex, imag, index, insert, intersection, intersection\_update, isalnum, isalpha, isdecimal, isdigit, isdisjoint, is\_integer, islower, isnumeric, isspace, issubset, issuperset, istitle, isupper, items, iteritems, iterkeys, itervalues, join, keys, ljust, lower, lstrip, numerator, partition, pop, popitem, real, remove, replace, reverse, rfind, rindex, rjust, rpartition, rsplit, rstrip, setdefault, sort, split, splitlines, startswith, strip, swapcase, symmetric\_difference, symmetric\_difference\_update, title, translate, union, update, upper, values, viewitems, viewkeys, viewvalues, zfill.

### Variables Defined in Inventory

We've actually already covered a lot about variables in another section, so far this shouldn't be terribly new, but a bit of a refresher.

Often you'll want to set variables based on what groups a machine is in. For instance, maybe machines in Boston want to use 'boston.ntp.example.com' as an NTP server.

See the [Working with Inventory](#) document for multiple ways on how to define variables in inventory.

### Variables Defined in a Playbook

In a playbook, it's possible to define variables directly inline like so:

```
- hosts: webservers
  vars:
    http_port: 80
```

This can be nice as it's right there when you are reading the playbook.

### Variables defined from included files and roles

It turns out we've already talked about variables in another place too.

As described in [Roles](#), variables can also be included in the playbook via include files, which may or may not be part of an "Ansible Role". Usage of roles is preferred as it provides a nice organizational system.

### Using Variables: About Jinja2

It's nice enough to know about how to define variables, but how do you use them?

Ansible allows you to reference variables in your playbooks using the Jinja2 templating system. While you can do a lot of complex things in Jinja, only the basics are things you really need to learn at first.

For example, in a simple template, you can do something like:

```
My amp goes to {{ max_amp_value }}
```

And that will provide the most basic form of variable substitution.

This is also valid directly in playbooks, and you'll occasionally want to do things like:

```
template: src=foo.cfg.j2 dest={{ remote_install_path }}/foo.cfg
```

In the above example, we used a variable to help decide where to place a file.

Inside a template you automatically have access to all of the variables that are in scope for a host. Actually it's more than that – you can also read variables about other hosts. We'll show how to do that in a bit.

---

: ansible allows Jinja2 loops and conditionals in templates, but in playbooks, we do not use them. Ansible playbooks are pure machine-parseable YAML. This is a rather important feature as it means it is possible to code-generate pieces of files, or to have other ecosystem tools read Ansible files. Not everyone will need this but it can unlock possibilities.

---

:

*Templating (Jinja2)* More information about Jinja2 templating

## Jinja2 Filters

---

: These are infrequently utilized features. Use them if they fit a use case you have, but this is optional knowledge.

---

Filters in Jinja2 are a way of transforming template expressions from one kind of data into another. Jinja2 ships with many of these. See [builtin filters](#) in the official Jinja2 template documentation.

In addition to those, Ansible supplies many more. See the [Filters](#) document for a list of available filters and example usage guide.

## Hey Wait, A YAML Gotcha

YAML syntax requires that if you start a value with `{{ foo }}` you quote the whole line, since it wants to be sure you aren't trying to start a YAML dictionary. This is covered on the [YAML Syntax](#) documentation.

This won't work:

```
- hosts: app_servers
  vars:
    app_path: {{ base_path }}/22
```

Do it like this and you'll be fine:

```
- hosts: app_servers
  vars:
    app_path: "{{ base_path }}/22"
```

## Information discovered from systems: Facts

There are other places where variables can come from, but these are a type of variable that are discovered, not set by the user.

Facts are information derived from speaking with your remote systems.

An example of this might be the IP address of the remote host, or what the operating system is.

To see what information is available, try the following:

```
ansible hostname -m setup
```

This will return a large amount of variable data, which may look like this, as taken from Ansible 1.4 running on a Ubuntu 12.04 system

```
{
  "ansible_all_ipv4_addresses": [
    "REDACTED IP ADDRESS"
  ],
  "ansible_all_ipv6_addresses": [
    "REDACTED IPV6 ADDRESS"
  ],
  "ansible_architecture": "x86_64",
  "ansible_bios_date": "09/20/2012",
  "ansible_bios_version": "6.00",
  "ansible_cmdline": {
    "BOOT_IMAGE": "/boot/vmlinuz-3.5.0-23-generic",
    "quiet": true,
    "ro": true,
    "root": "UUID=4195bff4-e157-4e41-8701-e93f0aec9e22",
    "splash": true
  },
  "ansible_date_time": {
    "date": "2013-10-02",
    "day": "02",
    "epoch": "1380756810",
    "hour": "19",
    "iso8601": "2013-10-02T23:33:30Z",
    "iso8601_micro": "2013-10-02T23:33:30.036070Z",
    "minute": "33",
    "month": "10",
    "second": "30",
    "time": "19:33:30",
    "tz": "EDT",
    "year": "2013"
  },
  "ansible_default_ipv4": {
    "address": "REDACTED",
    "alias": "eth0",
    "gateway": "REDACTED",
    "interface": "eth0",
    "macaddress": "REDACTED",
    "mtu": 1500,
    "netmask": "255.255.255.0",
    "network": "REDACTED",
    "type": "ether"
  },
  "ansible_default_ipv6": {},
  "ansible_devices": {
    "fd0": {
      "holders": [],
      "host": "",
      "model": null,
      "partitions": {},
    }
  }
}
```

(continues on next page)

()

```

        "removable": "1",
        "rotational": "1",
        "scheduler_mode": "deadline",
        "sectors": "0",
        "sectorsize": "512",
        "size": "0.00 Bytes",
        "support_discard": "0",
        "vendor": null
    },
    "sda": {
        "holders": [],
        "host": "SCSI storage controller: LSI Logic / Symbios Logic 53c1030 PCI-X_
↪Fusion-MPT Dual Ultra320 SCSI (rev 01)",
        "model": "VMware Virtual S",
        "partitions": {
            "sda1": {
                "sectors": "39843840",
                "sectorsize": 512,
                "size": "19.00 GB",
                "start": "2048"
            },
            "sda2": {
                "sectors": "2",
                "sectorsize": 512,
                "size": "1.00 KB",
                "start": "39847934"
            },
            "sda5": {
                "sectors": "2093056",
                "sectorsize": 512,
                "size": "1022.00 MB",
                "start": "39847936"
            }
        },
        "removable": "0",
        "rotational": "1",
        "scheduler_mode": "deadline",
        "sectors": "41943040",
        "sectorsize": "512",
        "size": "20.00 GB",
        "support_discard": "0",
        "vendor": "VMware,"
    },
    "sr0": {
        "holders": [],
        "host": "IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
↪",
        "model": "VMware IDE CDR10",
        "partitions": {},
        "removable": "1",
        "rotational": "1",
        "scheduler_mode": "deadline",
        "sectors": "2097151",
        "sectorsize": "512",
        "size": "1024.00 MB",
        "support_discard": "0",
        "vendor": "NECVMMWar"
    }
}

```

(continues on next page)

```

    },
    "ansible_distribution": "Ubuntu",
    "ansible_distribution_release": "precise",
    "ansible_distribution_version": "12.04",
    "ansible_domain": "",
    "ansible_env": {
        "COLORTERM": "gnome-terminal",
        "DISPLAY": ":0",
        "HOME": "/home/mdehaan",
        "LANG": "C",
        "LESSCLOSE": "/usr/bin/lesspipe %s %s",
        "LESSOPEN": "| /usr/bin/lesspipe %s",
        "LOGNAME": "root",
        "LS_COLORS": "rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;
↪33;01:cd=40;33;01:or=40;31;01:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;
↪44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.taz=01;31:*.lzh=01;31:*.lzma=01;
↪31:*.tlz=01;31:*.txz=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.
↪lz=01;31:*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.
↪deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.
↪ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.jpg=01;35:*.jpeg=01;35:*.
↪gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.
↪xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;
↪35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.
↪webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;
↪35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;
↪35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;
↪35:*.cgm=01;35:*.emf=01;35:*.axv=01;35:*.anx=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;
↪36:*.au=00;36:*.flac=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.
↪mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.axa=00;36:*.oga=00;36:*.spx=00;36:*.
↪xspf=00;36:",
        "MAIL": "/var/mail/root",
        "OLDPWD": "/root/ansible/docsite",
        "PATH": "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "PWD": "/root/ansible",
        "SHELL": "/bin/bash",
        "SHLVL": "1",
        "SUDO_COMMAND": "/bin/bash",
        "SUDO_GID": "1000",
        "SUDO_UID": "1000",
        "SUDO_USER": "mdehaan",
        "TERM": "xterm",
        "USER": "root",
        "USERNAME": "root",
        "XAUTHORITY": "/home/mdehaan/.Xauthority",
        "_": "/usr/local/bin/ansible"
    },
    "ansible_eth0": {
        "active": true,
        "device": "eth0",
        "ipv4": {
            "address": "REDACTED",
            "netmask": "255.255.255.0",
            "network": "REDACTED"
        },
        "ipv6": [
            {

```

(continues on next page)



```

        "address": "REDACTED",
        "prefix": "64",
        "scope": "link"
    },
    ],
    "macaddress": "REDACTED",
    "module": "e1000",
    "mtu": 1500,
    "type": "ether"
},
"ansible_form_factor": "Other",
"ansible_fqdn": "ubuntu2.example.com",
"ansible_hostname": "ubuntu2",
"ansible_interfaces": [
    "lo",
    "eth0"
],
"ansible_kernel": "3.5.0-23-generic",
"ansible_lo": {
    "active": true,
    "device": "lo",
    "ipv4": {
        "address": "127.0.0.1",
        "netmask": "255.0.0.0",
        "network": "127.0.0.0"
    },
    "ipv6": [
        {
            "address": "::1",
            "prefix": "128",
            "scope": "host"
        }
    ],
    "mtu": 16436,
    "type": "loopback"
},
"ansible_lsb": {
    "codename": "precise",
    "description": "Ubuntu 12.04.2 LTS",
    "id": "Ubuntu",
    "major_release": "12",
    "release": "12.04"
},
"ansible_machine": "x86_64",
"ansible_memfree_mb": 74,
"ansible_memtotal_mb": 991,
"ansible_mounts": [
    {
        "device": "/dev/sda1",
        "fstype": "ext4",
        "mount": "/",
        "options": "rw,errors=remount-ro",
        "size_available": 15032406016,
        "size_total": 20079898624
    }
],
"ansible_nodename": "ubuntu2.example.com",

```

(continues on next page)

```

"ansible_os_family": "Debian",
"ansible_pkg_mgr": "apt",
"ansible_processor": [
    "Intel(R) Core(TM) i7 CPU           860  @ 2.80GHz"
],
"ansible_processor_cores": 1,
"ansible_processor_count": 1,
"ansible_processor_threads_per_core": 1,
"ansible_processor_vcpus": 1,
"ansible_product_name": "VMware Virtual Platform",
"ansible_product_serial": "REDACTED",
"ansible_product_uuid": "REDACTED",
"ansible_product_version": "None",
"ansible_python_version": "2.7.3",
"ansible_selinux": false,
"ansible_ssh_host_key_dsa_public": "REDACTED KEY VALUE",
"ansible_ssh_host_key_ecdsa_public": "REDACTED KEY VALUE",
"ansible_ssh_host_key_rsa_public": "REDACTED KEY VALUE",
"ansible_swapfree_mb": 665,
"ansible_swaptotal_mb": 1021,
"ansible_system": "Linux",
"ansible_system_vendor": "VMware, Inc.",
"ansible_user_id": "root",
"ansible_userspace_architecture": "x86_64",
"ansible_userspace_bits": "64",
"ansible_virtualization_role": "guest",
"ansible_virtualization_type": "VMware"
}

```

In the above the model of the first harddrive may be referenced in a template or playbook as:

```
{{ ansible_devices.sda.model }}
```

Similarly, the hostname as the system reports it is:

```
{{ ansible_nodename }}
```

and the unqualified hostname shows the string before the first period(.):

```
{{ ansible_hostname }}
```

Facts are frequently used in conditionals (see [Conditionals](#)) and also in templates.

Facts can be also used to create dynamic groups of hosts that match particular criteria, see the [Working With Modules](#) documentation on **group\_by** for details, as well as in generalized conditional statements as discussed in the [Conditionals](#) chapter.

## Turning Off Facts

If you know you don't need any fact data about your hosts, and know everything about your systems centrally, you can turn off fact gathering. This has advantages in scaling Ansible in push mode with very large numbers of systems, mainly, or if you are using Ansible on experimental platforms. In any play, just do this:

```

- hosts: whatever
  gather_facts: no

```

## Local Facts (Facts.d)

### 1.3 .

As discussed in the playbooks chapter, Ansible facts are a way of getting data about remote systems for use in playbook variables.

Usually these are discovered automatically by the `setup` module in Ansible. Users can also write custom facts modules, as described in the API guide. However, what if you want to have a simple way to provide system or user provided data for use in Ansible variables, without writing a fact module?

"Facts.d" is one mechanism for users to control some aspect of how their systems are managed.

---

: Perhaps "local facts" is a bit of a misnomer, it means "locally supplied user values" as opposed to "centrally supplied user values", or what facts are – "locally dynamically determined values".

---

If a remotely managed system has an `/etc/ansible/facts.d` directory, any files in this directory ending in `.fact`, can be JSON, INI, or executable files returning JSON, and these can supply local facts in Ansible. An alternate directory can be specified using the `fact_path` play keyword.

For example, assume `/etc/ansible/facts.d/preferences.fact` contains:

```
[general]
asdf=1
bar=2
```

This will produce a hash variable fact named `general` with `asdf` and `bar` as members. To validate this, run the following:

```
ansible <hostname> -m setup -a "filter=ansible_local"
```

And you will see the following fact added:

```
"ansible_local": {
    "preferences": {
        "general": {
            "asdf" : "1",
            "bar"  : "2"
        }
    }
}
```

And this data can be accessed in a template/playbook as:

```
{{ ansible_local.preferences.general.asdf }}
```

The local namespace prevents any user supplied fact from overriding system facts or variables defined elsewhere in the playbook.

---

: The key part in the key=value pairs will be converted into lowercase inside the `ansible_local` variable. Using the example above, if the ini file contained `XYZ=3` in the `[general]` section, then you should expect to access it as: `{{ ansible_local.preferences.general.xyz }}` and not `{{ ansible_local.preferences.general.XYZ }}`. This is because Ansible uses Python's `ConfigParser` which passes all option names through the `optionxform` method and this method's default implementation converts option names to lower case.

---

If you have a playbook that is copying over a custom fact and then running it, making an explicit call to re-run the setup module can allow that fact to be used during that particular play. Otherwise, it will be available in the next play that gathers fact information. Here is an example of what that might look like:

```
- hosts: webservers
  tasks:
    - name: create directory for ansible custom facts
      file: state=directory recurse=yes path=/etc/ansible/facts.d
    - name: install custom ipmi fact
      copy: src=ipmi.fact dest=/etc/ansible/facts.d
    - name: re-read facts after adding custom fact
      setup: filter=ansible_local
```

In this pattern however, you could also write a fact module as well, and may wish to consider this as an option.

### Ansible version

1.8 .

To adapt playbook behavior to specific version of ansible, a variable `ansible_version` is available, with the following structure:

```
"ansible_version": {
  "full": "2.0.0.2",
  "major": 2,
  "minor": 0,
  "revision": 0,
  "string": "2.0.0.2"
}
```

### Fact Caching

1.8 .

As shown elsewhere in the docs, it is possible for one server to reference variables about another, like so:

```
{{ hostvars['asdf.example.com']['ansible_os_family'] }}
```

With "Fact Caching" disabled, in order to do this, Ansible must have already talked to 'asdf.example.com' in the current play, or another play up higher in the playbook. This is the default configuration of ansible.

To avoid this, Ansible 1.8 allows the ability to save facts between playbook runs, but this feature must be manually enabled. Why might this be useful?

With a very large infrastructure with thousands of hosts, fact caching could be configured to run nightly. Configuration of a small set of servers could run ad-hoc or periodically throughout the day. With fact caching enabled, it would not be necessary to "hit" all servers to reference variables and information about them.

With fact caching enabled, it is possible for machine in one group to reference variables about machines in the other group, despite the fact that they have not been communicated with in the current execution of `/usr/bin/ansible-playbook`.

To benefit from cached facts, you will want to change the `gathering` setting to `smart` or `explicit` or set `gather_facts` to `False` in most plays.

Currently, Ansible ships with two persistent cache plugins: `redis` and `jsonfile`.

To configure fact caching using `redis`, enable it in `ansible.cfg` as follows:

```
[defaults]
gathering = smart
fact_caching = redis
fact_caching_timeout = 86400
# seconds
```

To get redis up and running, perform the equivalent OS commands:

```
yum install redis
service redis start
pip install redis
```

Note that the Python redis library should be installed from pip, the version packaged in EPEL is too old for use by Ansible.

In current embodiments, this feature is in beta-level state and the Redis plugin does not support port or password configuration, this is expected to change in the near future.

To configure fact caching using jsonfile, enable it in `ansible.cfg` as follows:

```
[defaults]
gathering = smart
fact_caching = jsonfile
fact_caching_connection = /path/to/cachedir
fact_caching_timeout = 86400
# seconds
```

`fact_caching_connection` is a local filesystem path to a writeable directory (ansible will attempt to create the directory if one does not exist).

`fact_caching_timeout` is the number of seconds to cache the recorded facts.

## Registered Variables

Another major use of variables is running a command and using the result of that command to save the result into a variable. Results will vary from module to module. Use of `-v` when executing playbooks will show possible values for the results.

The value of a task being executed in ansible can be saved in a variable and used later. See some examples of this in the [Conditionals](#) chapter.

While it's mentioned elsewhere in that document too, here's a quick syntax example:

```
- hosts: web_servers

  tasks:

    - shell: /usr/bin/foo
      register: foo_result
      ignore_errors: True

    - shell: /usr/bin/bar
      when: foo_result.rc == 5
```

Registered variables are valid on the host the remainder of the playbook run, which is the same as the lifetime of "facts" in Ansible. Effectively registered variables are just like facts.

When using `register` with a loop, the data structure placed in the variable during the loop will contain a `results` attribute, that is a list of all responses from the module. For a more in-depth example of how this works, see the [Loops](#) section on using `register` with a loop.

---

: If a task fails or is skipped, the variable still is registered with a failure or skipped status, the only way to avoid registering a variable is using tags.

---

### Accessing Complex Variable Data

We already described facts a little higher up in the documentation.

Some provided facts, like networking information, are made available as nested data structures. To access them a simple `{{ foo }}` is not sufficient, but it is still easy to do. Here's how we get an IP address:

```
{{ ansible_eth0["ipv4"]["address"] }}
```

OR alternatively:

```
{{ ansible_eth0.ipv4.address }}
```

Similarly, this is how we access the first element of an array:

```
{{ foo[0] }}
```

### Magic Variables, and How To Access Information About Other Hosts

Even if you didn't define them yourself, Ansible provides a few variables for you automatically. The most important of these are `hostvars`, `group_names`, and `groups`. Users should not use these names themselves as they are reserved. `environment` is also reserved.

`hostvars` lets you ask about the variables of another host, including facts that have been gathered about that host. If, at this point, you haven't talked to that host yet in any play in the playbook or set of playbooks, you can still get the variables, but you will not be able to see the facts.

If your database server wants to use the value of a 'fact' from another node, or an inventory variable assigned to another node, it's easy to do so within a template or even an action line:

```
{{ hostvars['test.example.com']['ansible_distribution'] }}
```

Additionally, `group_names` is a list (array) of all the groups the current host is in. This can be used in templates using Jinja2 syntax to make template source files that vary based on the group membership (or role) of the host

```
{% if 'webserver' in group_names %}
    # some part of a configuration file that only applies to webservers
{% endif %}
```

`groups` is a list of all the groups (and hosts) in the inventory. This can be used to enumerate all hosts within a group. For example:

```
{% for host in groups['app_servers'] %}
    # something that applies to all app servers.
{% endfor %}
```

A frequently used idiom is walking a group to find all IP addresses in that group

```
{% for host in groups['app_servers'] %}
  {{ hostvars[host]['ansible_eth0']['ipv4']['address'] }}
{% endfor %}
```

An example of this could include pointing a frontend proxy server to all of the app servers, setting up the correct firewall rules between servers, etc. You need to make sure that the facts of those hosts have been populated before though, for example by running a play against them if the facts have not been cached recently (fact caching was added in Ansible 1.8).

Additionally, `inventory_hostname` is the name of the hostname as configured in Ansible's inventory host file. This can be useful for when you don't want to rely on the discovered hostname `ansible_hostname` or for other mysterious reasons. If you have a long FQDN, `inventory_hostname_short` also contains the part up to the first period, without the rest of the domain.

`play_hosts` has been deprecated in 2.2, it was the same as the new `ansible_play_batch` variable.

2.2 .

`ansible_play_hosts` is the full list of all hosts still active in the current play.

2.2 .

`ansible_play_batch` is available as a list of hostnames that are in scope for the current 'batch' of the play. The batch size is defined by `serial`, when not set it is equivalent to the whole play (making it the same as `ansible_play_hosts`).

2.3 .

`ansible_playbook_python` is the path to the python executable used to invoke the Ansible command line tool.

These vars may be useful for filling out templates with multiple hostnames or for injecting the list into the rules for a load balancer.

Don't worry about any of this unless you think you need it. You'll know when you do.

Also available, `inventory_dir` is the pathname of the directory holding Ansible's inventory host file, `inventory_file` is the pathname and the filename pointing to the Ansible's inventory host file.

`playbook_dir` contains the playbook base directory.

We then have `role_path` which will return the current role's pathname (since 1.8). This will only work inside a role.

And finally, `ansible_check_mode` (added in version 2.1), a boolean magic variable which will be set to `True` if you run Ansible with `--check`.

## Variable File Separation

It's a great idea to keep your playbooks under source control, but you may wish to make the playbook source public while keeping certain important variables private. Similarly, sometimes you may just want to keep certain information in different files, away from the main playbook.

You can do this by using an external variables file, or files, just like this:

```
---
- hosts: all
  remote_user: root
  vars:
    favcolor: blue
```

(continues on next page)

```
vars_files:
  - /vars/external_vars.yml

tasks:

- name: this is just a placeholder
  command: /bin/echo foo
```

This removes the risk of sharing sensitive data with others when sharing your playbook source with them.

The contents of each variables file is a simple YAML dictionary, like this:

```
---
# in the above example, this would be vars/external_vars.yml
somevar: somevalue
password: magic
```

---

: It's also possible to keep per-host and per-group variables in very similar files, this is covered in *Splitting Out Host and Group Specific Data*.

---

## Passing Variables On The Command Line

In addition to `vars_prompt` and `vars_files`, it is possible to set variables at the command line using the `--extra-vars` (or `-e`) argument. Variables can be defined using a single quoted string (containing one or more variables) using one of the formats below

key=value format:

```
ansible-playbook release.yml --extra-vars "version=1.23.45 other_variable=foo"
```

---

: Values passed in using the `key=value` syntax are interpreted as strings. Use the JSON format if you need to pass in anything that shouldn't be a string (Booleans, integers, floats, lists etc).

---

1.2 .

JSON string format:

```
ansible-playbook release.yml --extra-vars '{"version":"1.23.45","other_variable":"foo"}'
ansible-playbook arcade.yml --extra-vars '{"pacman":"mrs","ghosts":["inky","pinky","clyde","sue"]}'
```

1.3 .

YAML string format:

```
ansible-playbook release.yml --extra-vars '
version: "1.23.45"
other_variable: foo'

ansible-playbook arcade.yml --extra-vars '
pacman: mrs
```

(continues on next page)



()

```
ghosts:
- inky
- pinky
- clyde
- sue'
```

### 1.3 .

vars from a JSON or YAML file:

```
ansible-playbook release.yml --extra-vars "@some_file.json"
```

This is useful for, among other things, setting the hosts group or the user for the playbook.

Escaping quotes and other special characters:

### 1.2 .

Ensure you're escaping quotes appropriately for both your markup (e.g. JSON), and for the shell you're operating in.:

```
ansible-playbook arcade.yml --extra-vars "{\"name\":\"Conan O'Brien\"}"
ansible-playbook arcade.yml --extra-vars '{"name":"Conan O\\\\\\'Brien"}'
ansible-playbook script.yml --extra-vars "{\"dialog\":\"He said \\\"I just can't get_
↪ enough of those single and double-quotes\"!\"\\\\\"\"}"
```

### 1.3 .

In these cases, it's probably best to use a JSON or YAML file containing the variable definitions.

## Variable Precedence: Where Should I Put A Variable?

A lot of folks may ask about how variables override another. Ultimately it's Ansible's philosophy that it's better you know where to put a variable, and then you have to think about it a lot less.

Avoid defining the variable "x" in 47 places and then ask the question "which x gets used". Why? Because that's not Ansible's Zen philosophy of doing things.

There is only one Empire State Building. One Mona Lisa, etc. Figure out where to define a variable, and don't make it complicated.

However, let's go ahead and get precedence out of the way! It exists. It's a real thing, and you might have a use for it.

If multiple variables of the same name are defined in different places, they get overwritten in a certain order.

Here is the order of precedence from least to greatest (the last listed variables winning prioritization):

- role defaults<sup>1</sup>
- inventory file or script group vars<sup>2</sup>
- inventory group\_vars/all<sup>3</sup>
- playbook group\_vars/all<sup>3</sup>
- inventory group\_vars/\*<sup>3</sup>
- playbook group\_vars/\*<sup>3</sup>

<sup>1</sup> Tasks in each role will see their own role's defaults. Tasks defined outside of a role will see the last role's defaults.

<sup>2</sup> Variables defined in inventory file or provided by dynamic inventory.

<sup>3</sup> Includes vars added by 'vars plugins' as well as host\_vars and group\_vars which are added by the default vars plugin shipped with Ansible.

- inventory file or script host vars<sup>2</sup>
- inventory host\_vars/\*
- playbook host\_vars/\*
- host facts / cached set\_facts<sup>4</sup>
- inventory host\_vars/\*<sup>3</sup>
- playbook host\_vars/\*<sup>3</sup>
- host facts
- play vars
- play vars\_prompt
- play vars\_files
- role vars (defined in role/vars/main.yml)
- block vars (only for tasks in block)
- task vars (only for the task)
- include\_vars
- set\_facts / registered vars
- role (and include\_role) params
- include params
- extra vars (always win precedence)

Basically, anything that goes into "role defaults" (the defaults folder inside the role) is the most malleable and easily overridden. Anything in the vars directory of the role overrides previous versions of that variable in namespace. The idea here to follow is that the more explicit you get in scope, the more precedence it takes with command line `-e` extra vars always winning. Host and/or inventory variables can win over role defaults, but not explicit includes like the vars directory or an `include_vars` task.

---

: Within any section, redefining a var will overwrite the previous instance. If multiple groups have the same variable, the last one loaded wins. If you define a variable twice in a play's `vars:` section, the second one wins.

---

---

: The previous describes the default config `hash_behaviour=replace`, switch to `merge` to only partially overwrite.

---

---

: Group loading follows parent/child relationships. Groups of the same 'parent/child' level are then merged following alphabetical order. This last one can be superseded by the user via `ansible_group_priority`, which defaults to 1 for all groups.

---

Another important thing to consider (for all versions) is that connection variables override config, command line and play/role/task specific options and keywords. For example:

```
ansible -u lola myhost
```

---

<sup>4</sup> When created with `set_facts's` `cacheable` option, variables will have the high precedence in the play, but will be the same as a host facts precedence when they come from the cache.

This will still connect as `ramon` because `ansible_ssh_user` is set to `ramon` in inventory for `myhost`. For plays/tasks this is also true for `remote_user`:

```
- hosts: myhost
  tasks:
    - command: i'll connect as ramon still
      remote_user: lola
```

This is done so host-specific settings can override the general settings. These variables are normally defined per host or group in inventory, but they behave like other variables. If you want to override the remote user globally (even over inventory) you can use extra vars:

```
ansible... -e "ansible_user=<user>"
```

You can also override as a normal variable in a play:

```
- hosts: all
  vars:
    ansible_user: lola
  tasks:
    - command: i'll connect as lola!
```

## Variable Scopes

Ansible has three main scopes:

- Global: this is set by config, environment variables and the command line
- Play: each play and contained structures, vars entries (`vars`; `vars_files`; `vars_prompt`), role defaults and vars.
- Host: variables directly associated to a host, like inventory, `include_vars`, facts or registered task outputs

## Variable Examples

Let's show some examples and where you would choose to put what based on the kind of control you might want over values.

First off, group variables are powerful.

Site wide defaults should be defined as a `group_vars/all` setting. Group variables are generally placed alongside your inventory file. They can also be returned by a dynamic inventory script (see [Working With Dynamic Inventory](#)) or defined in things like [Ansible Tower](#) from the UI or API:

```
---
# file: /etc/ansible/group_vars/all
# this is the site wide default
ntp_server: default-time.example.com
```

Regional information might be defined in a `group_vars/region` variable. If this group is a child of the `all` group (which it is, because all groups are), it will override the group that is higher up and more general:

```
---
# file: /etc/ansible/group_vars/boston
ntp_server: boston-time.example.com
```

If for some crazy reason we wanted to tell just a specific host to use a specific NTP server, it would then override the group variable!:

```
---
# file: /etc/ansible/host_vars/xyz.boston.example.com
ntp_server: override.example.com
```

So that covers inventory and what you would normally set there. It's a great place for things that deal with geography or behavior. Since groups are frequently the entity that maps roles onto hosts, it is sometimes a shortcut to set variables on the group instead of defining them on a role. You could go either way.

Remember: Child groups override parent groups, and hosts always override their groups.

Next up: learning about role variable precedence.

We'll pretty much assume you are using roles at this point. You should be using roles for sure. Roles are great. You are using roles aren't you? Hint hint.

If you are writing a redistributable role with reasonable defaults, put those in the `roles/x/defaults/main.yml` file. This means the role will bring along a default value but **ANYTHING** in Ansible will override it. See [Roles](#) for more info about this:

```
---
# file: roles/x/defaults/main.yml
# if not overridden in inventory or as a parameter, this is the value that will be_
↪used
http_port: 80
```

If you are writing a role and want to ensure the value in the role is absolutely used in that role, and is not going to be overridden by inventory, you should put it in `roles/x/vars/main.yml` like so, and inventory values cannot override it. `-e` however, still will:

```
---
# file: roles/x/vars/main.yml
# this will absolutely be used in this role
http_port: 80
```

This is one way to plug in constants about the role that are always true. If you are not sharing your role with others, app specific behaviors like ports is fine to put in here. But if you are sharing roles with others, putting variables in here might be bad. Nobody will be able to override them with inventory, but they still can by passing a parameter to the role.

Parameterized roles are useful.

If you are using a role and want to override a default, pass it as a parameter to the role like so:

```
roles:
  - role: apache
    vars:
      http_port: 8080
```

This makes it clear to the playbook reader that you've made a conscious choice to override some default in the role, or pass in some configuration that the role can't assume by itself. It also allows you to pass something site-specific that isn't really part of the role you are sharing with others.

This can often be used for things that might apply to some hosts multiple times. For example:

```
roles:
  - role: app_user
    vars:
      myname: Ian
  - role: app_user
```

(continues on next page)

()

```
vars:
  myname: Terry
- role: app_user
vars:
  myname: Graham
- role: app_user
vars:
  myname: John
```

In this example, the same role was invoked multiple times. It's quite likely there was no default for `name` supplied at all. Ansible can warn you when variables aren't defined – it's the default behavior in fact.

There are a few other things that go on with roles.

Generally speaking, variables set in one role are available to others. This means if you have a `roles/common/vars/main.yml` you can set variables in there and make use of them in other roles and elsewhere in your playbook:

```
roles:
  - role: common_settings
  - role: something
  vars:
    foo: 12
  - role: something_else
```

---

: There are some protections in place to avoid the need to namespace variables. In the above, variables defined in `common_settings` are most definitely available to 'something' and 'something\_else' tasks, but if "something's" guaranteed to have `foo` set at 12, even if somewhere deep in common settings it set `foo` to 20.

---

So, that's precedence, explained in a more direct way. Don't worry about precedence, just think about if your role is defining a variable that is a default, or a "live" variable you definitely want to use. Inventory lies in precedence right in the middle, and if you want to forcibly override something, use `-e`.

If you found that a little hard to understand, take a look at the [ansible-examples](#) repo on our github for a bit more about how all of these things can work together.

## Advanced Syntax

For information about advanced YAML syntax used to declare variables and have more control over the data placed in YAML files used by Ansible, see [Advanced Syntax](#).

:

**[Working With Playbooks](#)** An introduction to playbooks

**[Conditionals](#)** Conditional statements in playbooks

**[Filters](#)** Jinja2 filters and their uses

**[Loops](#)** Looping in playbooks

**[Roles](#)** Playbook organization by roles

**[Best Practices](#)** Best practices in playbooks

**[User Mailing List](#)** Have a question? Stop by the google group!

**[irc.freenode.net](#)** #ansible IRC chat channel

### Templating (Jinja2)

As already referenced in the variables section, Ansible uses Jinja2 templating to enable dynamic expressions and access to variables. Ansible greatly expands the number of filters and tests available, as well as adding a new plugin type: lookups.

Please note that all templating happens on the Ansible controller before the task is sent and executed on the target machine. This is done to minimize the requirements on the target (jinja2 is only required on the controller) and to be able to pass the minimal information needed for the task, so the target machine does not need a copy of all the data that the controller has access to.

#### Topics

- *Templating (Jinja2)*

### Filters

#### Topics

- *Filters*
  - *Filters For Formatting Data*
  - *Forcing Variables To Be Defined*
  - *Defaulting Undefined Variables*
  - *Omitting Parameters*
  - *List Filters*
  - *Set Theory Filters*
  - *Dict Filter*
  - *items2dict filter*
  - *zip and zip\_longest filters*
  - *subelements Filter*
  - *Random Mac Address Filter*
  - *Random Number Filter*
  - *Shuffle Filter*
  - *Math*
  - *JSON Query Filter*
  - *IP address filter*
  - *Network CLI filters*
  - *Network XML filters*
  - *Hashing filters*
  - *Combining hashes/dictionaries*

- *Extracting values from containers*
- *Comment Filter*
- *URL Split Filter*
- *Regular Expression Filters*
- *Other Useful Filters*
- *Combination Filters*
- *Debugging Filters*

Filters in Ansible are from Jinja2, and are used for transforming data inside a template expression. Jinja2 ships with many filters. See [builtin filters](#) in the official Jinja2 template documentation.

Take into account that templating happens on the Ansible controller, **not** on the task's target host, so filters also execute on the controller as they manipulate local data.

In addition the ones provided by Jinja2, Ansible ships with it's own and allows users to add their own custom filters.

## Filters For Formatting Data

The following filters will take a data structure in a template and render it in a slightly different format. These are occasionally useful for debugging:

```
{{ some_variable | to_json }}
{{ some_variable | to_yaml }}
```

For human readable output, you can use:

```
{{ some_variable | to_nice_json }}
{{ some_variable | to_nice_yaml }}
```

It's also possible to change the indentation of both (new in version 2.2):

```
{{ some_variable | to_nice_json(indent=2) }}
{{ some_variable | to_nice_yaml(indent=8) }}
```

Alternatively, you may be reading in some already formatted data:

```
{{ some_variable | from_json }}
{{ some_variable | from_yaml }}
```

for example:

```
tasks:
  - shell: cat /some/path/to/file.json
    register: result

  - set_fact:
    myvar: "{{ result.stdout | from_json }}"
```

## Forcing Variables To Be Defined

The default behavior from ansible and ansible.cfg is to fail if variables are undefined, but you can turn this off.

This allows an explicit check with this feature off:

```
{{ variable | mandatory }}
```

The variable value will be used as is, but the template evaluation will raise an error if it is undefined.

### Defaulting Undefined Variables

Jinja2 provides a useful 'default' filter that is often a better approach to failing if a variable is not defined:

```
{{ some_variable | default(5) }}
```

In the above example, if the variable 'some\_variable' is not defined, the value used will be 5, rather than an error being raised.

If the variable evaluates to an empty string, the second parameter of the filter should be set to *true*:

```
{{ lookup('env', 'MY_USER') | default('admin', true) }}
```

### Omitting Parameters

As of Ansible 1.8, it is possible to use the default filter to omit module parameters using the special *omit* variable:

```
- name: touch files with an optional mode
  file: dest={{item.path}} state=touch mode={{item.mode|default(omit) }}
  loop:
    - path: /tmp/foo
    - path: /tmp/bar
    - path: /tmp/baz
    mode: "0444"
```

For the first two files in the list, the default mode will be determined by the umask of the system as the *mode=* parameter will not be sent to the file module while the final file will receive the *mode=0444* option.

---

: If you are "chaining" additional filters after the *default(omit)* filter, you should instead do something like this: `"{{ foo | default(None) | some_filter or omit }}"`. In this example, the default *None* (python null) value will cause the later filters to fail, which will trigger the *or omit* portion of the logic. Using *omit* in this manner is very specific to the later filters you're chaining though, so be prepared for some trial and error if you do this.

---

### List Filters

These filters all operate on list variables.

1.8 .

To get the minimum value from list of numbers:

```
{{ list1 | min }}
```

To get the maximum value from a list of numbers:



```
{{ [3, 4, 2] | max }}
```

2.5 .

Flatten a list (same thing the *flatten* lookup does):

```
{{ [3, [4, 2]] | flatten }}
```

Flatten only the first level of a list (akin to the *items* lookup):

```
{{ [3, [4, [2]]] | flatten(levels=1) }}
```

## Set Theory Filters

All these functions return a unique set from sets or lists.

1.4 .

To get a unique set from a list:

```
{{ list1 | unique }}
```

To get a union of two lists:

```
{{ list1 | union(list2) }}
```

To get the intersection of 2 lists (unique list of all items in both):

```
{{ list1 | intersect(list2) }}
```

To get the difference of 2 lists (items in 1 that don't exist in 2):

```
{{ list1 | difference(list2) }}
```

To get the symmetric difference of 2 lists (items exclusive to each list):

```
{{ list1 | symmetric_difference(list2) }}
```

## Dict Filter

2.6 .

To turn a dictionary into a list of items, suitable for looping, use *dict2items*:

```
{{ dict | dict2items }}
```

Which turns:

```
tags:
  Application: payment
  Environment: dev
```

into:

```
- key: Application
  value: payment
- key: Environment
  value: dev
```

### items2dict filter

2.7 .

This filter turns a list of dicts with 2 keys, into a dict, mapping the values of those keys into key: value pairs:

```
{{ tags | items2dict }}
```

Which turns:

```
tags:
- key: Application
  value: payment
- key: Environment
  value: dev
```

into:

```
Application: payment
Environment: dev
```

This is the reverse of the dict2items filter.

items2dict accepts 2 keyword arguments, key\_name and value\_name that allow configuration of the names of the keys to use for the transformation:

```
{{ tags | items2dict(key_name='key', value_name='value') }}
```

### zip and zip\_longest filters

2.3 .

To get a list combining the elements of other lists use zip:

```
- name: give me list combo of two lists
  debug:
    msg: "{{ [1,2,3,4,5]|zip(['a','b','c','d','e','f'])|list }}"

- name: give me shortest combo of two lists
  debug:
    msg: "{{ [1,2,3]|zip(['a','b','c','d','e','f'])|list }}"
```

To always exhaust all list use zip\_longest:

```
- name: give me longest combo of three lists , fill with X
  debug:
    msg: "{{ [1,2,3]|zip_longest(['a','b','c','d','e','f'], [21, 22, 23], fillvalue='X
    ↪')|list }}"
```

Similarly to the output of the items2dict filter mentioned above, these filters can be used to construct a dict:

```
{{ dict(keys_list | zip(values_list)) }}
```

Which turns:

```
list_one:
- one
- two
list_two:
- apple
- orange
```

into:

```
one: apple
two: orange
```

## subelements Filter

2.7 .

Produces a product of an object, and subelement values of that object, similar to the `subelements` lookup:

```
{{ users|subelements('groups', skip_missing=True) }}
```

Which turns:

```
users:
- name: alice
  authorized:
    - /tmp/alice/onekey.pub
    - /tmp/alice/twokey.pub
  groups:
    - wheel
    - docker
- name: bob
  authorized:
    - /tmp/bob/id_rsa.pub
  groups:
    - docker
```

Into:

```
-
- name: alice
  groups:
    - wheel
    - docker
  authorized:
    - /tmp/alice/onekey.pub
- wheel
-
- name: alice
  groups:
    - wheel
    - docker
  authorized:
```

(continues on next page)

```

- /tmp/alice/onekey.pub
- docker
-
- name: bob
  authorized:
    - /tmp/bob/id_rsa.pub
  groups:
    - docker
- docker

```

An example of using this filter with loop:

```

- name: Set authorized ssh key, extracting just that data from 'users'
  authorized_key:
    user: "{{ item.0.name }}"
    key: "{{ lookup('file', item.1) }}"
  loop: "{{ users|subelements('authorized') }}"

```

## Random Mac Address Filter

2.6 .

This filter can be used to generate a random MAC address from a string prefix.

To get a random MAC address from a string prefix starting with '52:54:00':

```

"{{ '52:54:00'|random_mac }}"
# => '52:54:00:ef:1c:03'

```

Note that if anything is wrong with the prefix string, the filter will issue an error.

## Random Number Filter

1.6 .

This filter can be used similar to the default jinja2 random filter (returning a random item from a sequence of items), but can also generate a random number based on a range.

To get a random item from a list:

```

"{{ ['a', 'b', 'c']|random }}"
# => 'c'

```

To get a random number between 0 and a specified number:

```

"{{ 60 |random }}" * * * * root /script/from/cron"
# => '21 * * * * root /script/from/cron'

```

Get a random number from 0 to 100 but in steps of 10:

```

"{{ 101 |random(step=10) }}"
# => 70

```

Get a random number from 1 to 100 but in steps of 10:

```
{{ 101 | random(1, 10) }}
```

```
# => 31
```

```
{{ 101 | random(start=1, step=10) }}
```

```
# => 51
```

As of Ansible version 2.3, it's also possible to initialize the random number generator from a seed. This way, you can create random-but-idempotent numbers:

```
"{{ 60 | random(seed=inventory_hostname) }} * * * * root /script/from/cron"
```

## Shuffle Filter

### 1.8 .

This filter will randomize an existing list, giving a different order every invocation.

To get a random list from an existing list:

```
{{ ['a', 'b', 'c'] | shuffle }}
```

```
# => ['c', 'a', 'b']
```

```
{{ ['a', 'b', 'c'] | shuffle }}
```

```
# => ['b', 'c', 'a']
```

As of Ansible version 2.3, it's also possible to shuffle a list idempotent. All you need is a seed.:

```
{{ ['a', 'b', 'c'] | shuffle(seed=inventory_hostname) }}
```

```
# => ['b', 'a', 'c']
```

note that when used with a non 'listable' item it is a noop, otherwise it always returns a list

## Math

### 1.9 .

Get the logarithm (default is e):

```
{{ myvar | log }}
```

Get the base 10 logarithm:

```
{{ myvar | log(10) }}
```

Give me the power of 2! (or 5):

```
{{ myvar | pow(2) }}
```

```
{{ myvar | pow(5) }}
```

Square root, or the 5th:

```
{{ myvar | root }}
```

```
{{ myvar | root(5) }}
```

Note that jinja2 already provides some like `abs()` and `round()`.

## JSON Query Filter

2.2 .

Sometimes you end up with a complex data structure in JSON format and you need to extract only a small set of data within it. The **json\_query** filter lets you query a complex JSON structure and iterate over it using a loop structure.

---

: This filter is built upon **jmespath**, and you can use the same syntax. For examples, see [jmespath examples](#).

---

Now, let's take the following data structure:

```
domain_definition:
  domain:
    cluster:
      - name: "cluster1"
      - name: "cluster2"
    server:
      - name: "server11"
        cluster: "cluster1"
        port: "8080"
      - name: "server12"
        cluster: "cluster1"
        port: "8090"
      - name: "server21"
        cluster: "cluster2"
        port: "9080"
      - name: "server22"
        cluster: "cluster2"
        port: "9090"
    library:
      - name: "lib1"
        target: "cluster1"
      - name: "lib2"
        target: "cluster2"
```

To extract all clusters from this structure, you can use the following query:

```
- name: "Display all cluster names"
  debug: var=item
  loop: "{{domain_definition|json_query('domain.cluster[*].name')}}"
```

Same thing for all server names:

```
- name: "Display all server names"
  debug: var=item
  loop: "{{domain_definition|json_query('domain.server[*].name')}}"
```

This example shows ports from cluster1:

```
- name: "Display all server names from cluster1"
  debug: var=item
  loop: "{{domain_definition|json_query(server_name_cluster1_query)}}"
  vars:
    server_name_cluster1_query: "domain.server[?cluster=='cluster1'].port"
```

---

: You can use a variable to make the query more readable.

---

Or, alternatively:

```
- name: "Display all server names from cluster1"
  debug:
    var: item
  loop: "{{domain_definition|json_query('domain.server[?cluster=`cluster1`].port')}}"
```

---

: Here, quoting literals using backticks avoids escaping quotes and maintains readability.

---

In this example, we get a hash map with all ports and names of a cluster:

```
- name: "Display all server ports and names from cluster1"
  debug:
    var: item
  loop: "{{domain_definition|json_query(server_name_cluster1_query)}}"
  vars:
    server_name_cluster1_query: "domain.server[?cluster=='cluster2'].{name: name,
    ↪port: port}"
```

## IP address filter

1.9 .

To test if a string is a valid IP address:

```
{{ myvar | ipaddr }}
```

You can also require a specific IP protocol version:

```
{{ myvar | ipv4 }}
{{ myvar | ipv6 }}
```

IP address filter can also be used to extract specific information from an IP address. For example, to get the IP address itself from a CIDR, you can use:

```
{{ '192.0.2.1/24' | ipaddr('address') }}
```

More information about `ipaddr` filter and complete usage guide can be found in `playbooks_filters_ipaddr`.

## Network CLI filters

2.4 .

To convert the output of a network device CLI command into structured JSON output, use the `parse_cli` filter:

```
{{ output | parse_cli('path/to/spec') }}
```

The `parse_cli` filter will load the spec file and pass the command output through it, returning JSON output. The YAML spec file defines how to parse the CLI output.

The spec file should be valid formatted YAML. It defines how to parse the CLI output and return JSON data. Below is an example of a valid spec file that will parse the output from the `show vlan` command.

```
---
vars:
  vlan:
    vlan_id: "{{ item.vlan_id }}"
    name: "{{ item.name }}"
    enabled: "{{ item.state != 'act/lshut' }}"
    state: "{{ item.state }}"

keys:
  vlans:
    value: "{{ vlan }}"
    items: "^ (?P<vlan_id>\\d+) \\s+ (?P<name>\\w+) \\s+ (?P<state>active|act/
↳ lshut|suspended) "
    state_static:
      value: present
```

The spec file above will return a JSON data structure that is a list of hashes with the parsed VLAN information.

The same command could be parsed into a hash by using the key and values directives. Here is an example of how to parse the output into a hash value using the same `show vlan` command.

```
---
vars:
  vlan:
    key: "{{ item.vlan_id }}"
    values:
      vlan_id: "{{ item.vlan_id }}"
      name: "{{ item.name }}"
      enabled: "{{ item.state != 'act/lshut' }}"
      state: "{{ item.state }}"

keys:
  vlans:
    value: "{{ vlan }}"
    items: "^ (?P<vlan_id>\\d+) \\s+ (?P<name>\\w+) \\s+ (?P<state>active|act/
↳ lshut|suspended) "
    state_static:
      value: present
```

Another common use case for parsing CLI commands is to break a large command into blocks that can be parsed. This can be done using the `start_block` and `end_block` directives to break the command into blocks that can be parsed.

```
---
vars:
  interface:
    name: "{{ item[0].match[0] }}"
    state: "{{ item[1].state }}"
    mode: "{{ item[2].match[0] }}"

keys:
  interfaces:
    value: "{{ interface }}"
    start_block: "^Ethernet.*$"
    end_block: "$"
```

(continues on next page)



()

```

items:
  - "(?P<name>Ethernet\\d\\/\\d*)"
  - "admin state is (?P<state>.+), "
  - "Port mode is (.+)"

```

The example above will parse the output of `show interface` into a list of hashes.

The network filters also support parsing the output of a CLI command using the TextFSM library. To parse the CLI output with TextFSM use the following filter:

```
{{ output.stdout[0] | parse_cli_textfsm('path/to/fsm') }}
```

Use of the TextFSM filter requires the TextFSM library to be installed.

## Network XML filters

2.5 .

To convert the XML output of a network device command into structured JSON output, use the `parse_xml` filter:

```
{{ output | parse_xml('path/to/spec') }}
```

The `parse_xml` filter will load the spec file and pass the command output through formatted as JSON.

The spec file should be valid formatted YAML. It defines how to parse the XML output and return JSON data.

Below is an example of a valid spec file that will parse the output from the `show vlan | display xml` command.

```

---
vars:
  vlan:
    vlan_id: "{{ item.vlan_id }}"
    name: "{{ item.name }}"
    desc: "{{ item.desc }}"
    enabled: "{{ item.state.get('inactive') != 'inactive' }}"
    state: "{{ % if item.state.get('inactive') == 'inactive' %} inactive {% else %}
↳ active {% endif % }}"
keys:
  vlans:
    value: "{{ vlan }}"
    top: configuration/vlans/vlan
    items:
      vlan_id: vlan-id
      name: name
      desc: description
      state: "[@inactive='inactive']"

```

The spec file above will return a JSON data structure that is a list of hashes with the parsed VLAN information.

The same command could be parsed into a hash by using the key and values directives. Here is an example of how to parse the output into a hash value using the same `show vlan | display xml` command.

```

---
vars:
  vlan:

```

(continues on next page)

```

key: "{{ item.vlan_id }}"
values:
  vlan_id: "{{ item.vlan_id }}"
  name: "{{ item.name }}"
  desc: "{{ item.desc }}"
  enabled: "{{ item.state.get('inactive') != 'inactive' }}"
  state: "% if item.state.get('inactive') == 'inactive'%} inactive {% else %}
↳active {% endif %}"

keys:
  vlans:
    value: "{{ vlan }}"
    top: configuration/vlans/vlan
    items:
      vlan_id: vlan-id
      name: name
      desc: description
      state: ".[@inactive='inactive']"

```

The value of `top` is the XPath relative to the XML root node. In the example XML output given below, the value of `top` is `configuration/vlans/vlan`, which is an XPath expression relative to the root node (`<rpc-reply>`). `configuration` in the value of `top` is the outer most container node, and `vlan` is the inner-most container node.

`items` is a dictionary of key-value pairs that map user-defined names to XPath expressions that select elements. The XPath expression is relative to the value of the XPath value contained in `top`. For example, the `vlan_id` in the spec file is a user defined name and its value `vlan-id` is the relative to the value of XPath in `top`

Attributes of XML tags can be extracted using XPath expressions. The value of `state` in the spec is an XPath expression used to get the attributes of the `vlan` tag in output XML.:

```

<rpc-reply>
  <configuration>
    <vlans>
      <vlan inactive="inactive">
        <name>vlan-1</name>
        <vlan-id>200</vlan-id>
        <description>This is vlan-1</description>
      </vlan>
    </vlans>
  </configuration>
</rpc-reply>

```

: For more information on supported XPath expressions, see <https://docs.python.org/2/library/xml.etree.elementtree.html#xpath-support>.

## Hashing filters

### 1.9 .

To get the sha1 hash of a string:

```
{{ 'test1'|hash('sha1') }}
```

To get the md5 hash of a string:

```
{{ 'test1'|hash('md5') }}
```

Get a string checksum:

```
{{ 'test2'|checksum }}
```

Other hashes (platform dependent):

```
{{ 'test2'|hash('blowfish') }}
```

To get a sha512 password hash (random salt):

```
{{ 'passwordsaresecret'|password_hash('sha512') }}
```

To get a sha256 password hash with a specific salt:

```
{{ 'secretpassword'|password_hash('sha256', 'mysecretsalt') }}
```

An idempotent method to generate unique hashes per system is to use a salt that is consistent between runs:

```
{{ 'secretpassword'|password_hash('sha512', 65534|random(seed=inventory_
↪hostname)|string) }}
```

Hash types available depend on the master system running ansible, 'hash' depends on hashlib password\_hash depends on passlib (<http://passlib.readthedocs.io/en/stable/lib/passlib.hash.html>).

## Combining hashes/dictionaries

2.0 .

The *combine* filter allows hashes to be merged. For example, the following would override keys in one hash:

```
{{ {'a':1, 'b':2}|combine({'b':3}) }}
```

The resulting hash would be:

```
{'a':1, 'b':3}
```

The filter also accepts an optional *recursive=True* parameter to not only override keys in the first hash, but also recurse into nested hashes and merge their keys too

```
{{ {'a':{'foo':1, 'bar':2}, 'b':2}|combine({'a':{'bar':3, 'baz':4}}, recursive=True) }
↪}}
```

This would result in:

```
{'a':{'foo':1, 'bar':3, 'baz':4}, 'b':2}
```

The filter can also take multiple arguments to merge:

```
{{ a|combine(b, c, d) }}
```

In this case, keys in *d* would override those in *c*, which would override those in *b*, and so on.

This behaviour does not depend on the value of the *hash\_behaviour* setting in *ansible.cfg*.

## Extracting values from containers

### 2.1 .

The *extract* filter is used to map from a list of indices to a list of values from a container (hash or array):

```
{{ [0,2]|map('extract', ['x','y','z'])|list }}
{{ ['x','y']|map('extract', {'x': 42, 'y': 31})|list }}
```

The results of the above expressions would be:

```
['x', 'z']
[42, 31]
```

The filter can take another argument:

```
{{ groups['x']|map('extract', hostvars, 'ec2_ip_address')|list }}
```

This takes the list of hosts in group 'x', looks them up in *hostvars*, and then looks up the *ec2\_ip\_address* of the result. The final result is a list of IP addresses for the hosts in group 'x'.

The third argument to the filter can also be a list, for a recursive lookup inside the container:

```
{{ ['a']|map('extract', b, ['x','y'])|list }}
```

This would return a list containing the value of *b['a']['x']['y']*.

## Comment Filter

### 2.0 .

The *comment* filter allows to decorate the text with a chosen comment style. For example the following:

```
{{ "Plain style (default)" | comment }}
```

will produce this output:

```
#
# Plain style (default)
#
```

Similar way can be applied style for C (*// . . .*), C block (*/\* . . . \*/*), Erlang (*% . . .*) and XML (*<!-- . . . -->*):

```
{{ "C style" | comment('c') }}
{{ "C block style" | comment('cblock') }}
{{ "Erlang style" | comment('erlang') }}
{{ "XML style" | comment('xml') }}
```

If you need a specific comment character that is not included by any of the above, you can customize it with:

```
{{ "My Special Case" | comment(decoration="! ") }}
```

producing:

```
!
! My Special Case
!
```

It is also possible to fully customize the comment style:

```
{{ "Custom style" | comment('plain', prefix='#####\n', postfix='#\n#####\n  ##
->#\n  #') }}
```

That will create the following output:

```
#####
#
# Custom style
#
#####
  ###
  #
```

The filter can also be applied to any Ansible variable. For example to make the output of the `ansible_managed` variable more readable, we can change the definition in the `ansible.cfg` file to this:

```
[defaults]

ansible_managed = This file is managed by Ansible.%n
    template: {file}
    date: %Y-%m-%d %H:%M:%S
    user: {uid}
    host: {host}
```

and then use the variable with the `comment` filter:

```
{{ ansible_managed | comment }}
```

which will produce this output:

```
#
# This file is managed by Ansible.
#
# template: /home/ansible/env/dev/ansible_managed/roles/role1/templates/test.j2
# date: 2015-09-10 11:02:58
# user: ansible
# host: myhost
#
```

## URL Split Filter

### 2.4 .

The `urlsplit` filter extracts the fragment, hostname, netloc, password, path, port, query, scheme, and username from an URL. With no arguments, returns a dictionary of all the fields:

```
{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#fragment" |
->urlsplit('hostname') }}
# => 'www.acme.com'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#fragment" |
->urlsplit('netloc') }}
# => 'user:password@www.acme.com:9000'
```

(continues on next page)

```

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#fragment" |
  ↪urlsplit('username') }}
# => 'user'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#fragment" |
  ↪urlsplit('password') }}
# => 'password'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#fragment" |
  ↪urlsplit('path') }}
# => '/dir/index.html'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#fragment" |
  ↪urlsplit('port') }}
# => '9000'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#fragment" |
  ↪urlsplit('scheme') }}
# => 'http'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#fragment" |
  ↪urlsplit('query') }}
# => 'query=term'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#fragment" |
  ↪urlsplit('fragment') }}
# => 'fragment'

{{ "http://user:password@www.acme.com:9000/dir/index.html?query=term#fragment" |
  ↪urlsplit }}
# =>
# {
#     "fragment": "fragment",
#     "hostname": "www.acme.com",
#     "netloc": "user:password@www.acme.com:9000",
#     "password": "password",
#     "path": "/dir/index.html",
#     "port": 9000,
#     "query": "query=term",
#     "scheme": "http",
#     "username": "user"
# }

```

## Regular Expression Filters

To search a string with a regex, use the "regex\_search" filter:

```

# search for "foo" in "foobar"
{{ 'foobar' | regex_search('(foo)') }}

# will return empty if it cannot find a match
{{ 'ansible' | regex_search('(foobar)') }}

# case insensitive search in multiline mode
{{ 'foo\nBAR' | regex_search('^bar", multiline=True, ignorecase=True) }}

```

To search for all occurrences of regex matches, use the "regex\_findall" filter:

```
# Return a list of all IPv4 addresses in the string
{{ 'Some DNS servers are 8.8.8.8 and 8.8.4.4' | regex_findall('\b(?:[0-9]{1,3}\.){3}
↪[0-9]{1,3}\b') }}
```

To replace text in a string with regex, use the "regex\_replace" filter:

```
# convert "ansible" to "able"
{{ 'ansible' | regex_replace('^a.*i(.*)$', 'a\\1') }}

# convert "foobar" to "bar"
{{ 'foobar' | regex_replace('^f.*o(.*)$', '\\1') }}

# convert "localhost:80" to "localhost, 80" using named groups
{{ 'localhost:80' | regex_replace('^(?P<host>.+):(P<port>\\d+)$', '\\g<host>, \\g
↪<port>') }}

# convert "localhost:80" to "localhost"
{{ 'localhost:80' | regex_replace(':80') }}

# add "https://" prefix to each item in a list
{{ hosts | map('regex_replace', '^(.*)$', 'https://\\1') | list }}
```

: Prior to ansible 2.0, if "regex\_replace" filter was used with variables inside YAML arguments (as opposed to simpler 'key=value' arguments), then you needed to escape backreferences (e.g. \\1) with 4 backslashes (\\\\\\1) instead of 2 (\\).

2.0 .

To escape special characters within a regex, use the "regex\_escape" filter:

```
# convert '^f.*o(.*)$' to '\\^f\\.\\*o\\(\\.\\*\\)\\$'
{{ '^f.*o(.*)$' | regex_escape() }}
```

## Other Useful Filters

To add quotes for shell usage:

```
- shell: echo {{ string_value | quote }}
```

To use one value on true and another on false (new in version 1.9):

```
{{ (name == "John") | ternary('Mr', 'Ms') }}
```

To concatenate a list into a string:

```
{{ list | join(" ") }}
```

To get the last name of a file path, like 'foo.txt' out of '/etc/asdf/foo.txt':

```
{{ path | basename }}
```

To get the last name of a windows style file path (new in version 2.0):

```
{{ path | win_basename }}
```

To separate the windows drive letter from the rest of a file path (new in version 2.0):

```
{{ path | win_splitdrive }}
```

To get only the windows drive letter:

```
{{ path | win_splitdrive | first }}
```

To get the rest of the path without the drive letter:

```
{{ path | win_splitdrive | last }}
```

To get the directory from a path:

```
{{ path | dirname }}
```

To get the directory from a windows path (new version 2.0):

```
{{ path | win_dirname }}
```

To expand a path containing a tilde (~) character (new in version 1.5):

```
{{ path | expanduser }}
```

To expand a path containing environment variables:

```
{{ path | expandvars }}
```

---

: *expandvars* expands local variables; using it on remote paths can lead to errors.

---

2.6 .

To get the real path of a link (new in version 1.8):

```
{{ path | realpath }}
```

To get the relative path of a link, from a start point (new in version 1.7):

```
{{ path | relpath('/etc') }}
```

To get the root and extension of a path or filename (new in version 2.0):

```
# with path == 'nginx.conf' the return would be ('nginx', '.conf')
{{ path | splitext }}
```

To work with Base64 encoded strings:

```
{{ encoded | b64decode }}
{{ decoded | b64encode }}
```

As of version 2.6, you can define the type of encoding to use, the default is `utf-8`:

```
{{ encoded | b64decode(encoding='utf-16-le') }}
{{ decoded | b64encode(encoding='utf-16-le') }}
```



## 2.6.

To create a UUID from a string (new in version 1.9):

```
{{ hostname | to_uuid }}
```

To cast values as certain types, such as when you input a string as "True" from a vars\_prompt and the system doesn't know it is a boolean value:

```
- debug:
  msg: test
  when: some_string_value | bool
```

## 1.6.

To make use of one attribute from each item in a list of complex variables, use the "map" filter (see the Jinja2 map() docs for more):

```
# get a comma-separated list of the mount points (e.g. "/", /mnt/stuff") on a host
{{ ansible_mounts|map(attribute='mount')|join(',') }}
```

To get date object from string use the `to_datetime` filter, (new in version in 2.2):

```
# Get total amount of seconds between two dates. Default date format is %Y-%m-%d %H:
↪%M:%S but you can pass your own format
{{ ((("2016-08-14 20:00:12"|to_datetime) - ("2015-12-25"|to_datetime('%Y-%m-%d'))).
↪total_seconds()  }}
```

```
# Get remaining seconds after delta has been calculated. NOTE: This does NOT convert
↪years, days, hours, etc to seconds. For that, use total_seconds()
{{ ((("2016-08-14 20:00:12"|to_datetime) - ("2016-08-14 18:00:00"|to_datetime)).
↪seconds  )}}
```

```
# This expression evaluates to "12" and not "132". Delta is 2 hours, 12 seconds
```

```
# get amount of days between two dates. This returns only number of days and discards
↪remaining hours, minutes, and seconds
{{ ((("2016-08-14 20:00:12"|to_datetime) - ("2015-12-25"|to_datetime('%Y-%m-%d'))).
↪days  )}}
```

## Combination Filters

## 2.3.

This set of filters returns a list of combined lists. To get permutations of a list:

```
- name: give me largest permutations (order matters)
  debug:
    msg: "{{ [1,2,3,4,5]|permutations|list }}"

- name: give me permutations of sets of three
  debug:
    msg: "{{ [1,2,3,4,5]|permutations(3)|list }}"
```

Combinations always require a set size:

```
- name: give me combinations for sets of two
  debug:
    msg: "{{ [1,2,3,4,5]|combinations(2)|list }}"
```

Also see the *zip* and *zip\_longest* filters

2.4 .

To format a date using a string (like with the shell date command), use the "strftime" filter:

```
# Display year-month-day
{{ '%Y-%m-%d' | strftime }}

# Display hour:min:sec
{{ '%H:%M:%S' | strftime }}

# Use ansible_date_time.epoch fact
{{ '%Y-%m-%d %H:%M:%S' | strftime(ansible_date_time.epoch) }}

# Use arbitrary epoch value
{{ '%Y-%m-%d' | strftime(0) }}           # => 1970-01-01
{{ '%Y-%m-%d' | strftime(1441357287) }} # => 2015-09-04
```

---

: To get all string possibilities, check <https://docs.python.org/2/library/time.html#time.strftime>

---

## Debugging Filters

2.3 .

Use the `type_debug` filter to display the underlying Python type of a variable. This can be useful in debugging in situations where you may need to know the exact type of a variable:

```
{{ myvar | type_debug }}
```

A few useful filters are typically added with each new Ansible release. The development documentation shows how to extend Ansible filters by writing your own as plugins, though in general, we encourage new ones to be added to core so everyone can make use of them.

:

**Working With Playbooks** An introduction to playbooks

**Conditionals** Conditional statements in playbooks

**Variables** All about variables

**Loops** Looping in playbooks

**Roles** Playbook organization by roles

**Best Practices** Best practices in playbooks

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Tests

**Topics**

- *Tests*
  - *Test syntax*
  - *Testing strings*
  - *Version Comparison*
  - *Group theory tests*
  - *Testing paths*
  - *Task results*

**Tests** in Jinja are a way of evaluating template expressions and returning True or False. Jinja ships with many of these. See [builtin tests](#) in the official Jinja template documentation.

The main difference between tests and filters are that Jinja tests are used for comparisons, whereas filters are used for data manipulation, and have different applications in jinja. Tests can also be used in list processing filters, like `map()` and `select()` to choose items in the list.

Like all templating, tests always execute on the Ansible controller, **not** on the target of a task, as they test local data.

In addition to those Jinja2 tests, Ansible supplies a few more and users can easily create their own.

**Test syntax**

**Test syntax** varies from **filter syntax** (`variable | filter`). Historically Ansible has registered tests as both jinja tests and jinja filters, allowing for them to be referenced using filter syntax.

As of Ansible 2.5, using a jinja test as a filter will generate a warning.

The syntax for using a jinja test is as follows:

```
variable is test_name
```

Such as:

```
result is failed
```

**Testing strings**

To match strings against a substring or a regex, use the "match" or "search" filter:

```
vars:
  url: "http://example.com/users/foo/resources/bar"

tasks:
  - debug:
      msg: "matched pattern 1"
      when: url is match("http://example.com/users/.*resources/.+")

  - debug:
      msg: "matched pattern 2"
      when: url is search("/users/.*resources/.+")
```

(continues on next page)

```
- debug:
  msg: "matched pattern 3"
  when: url is search("/users/")
```

'match' requires a complete match in the string, while 'search' only requires matching a subset of the string.

## Version Comparison

1.6 .

---

: In 2.5 `version_compare` was renamed to `version`

---

To compare a version number, such as checking if the `ansible_distribution_version` version is greater than or equal to '12.04', you can use the `version` test.

The `version` test can also be used to evaluate the `ansible_distribution_version`:

```
{{ ansible_distribution_version is version('12.04', '>=') }}
```

If `ansible_distribution_version` is greater than or equal to 12.04, this test returns True, otherwise False.

The `version` test accepts the following operators:

```
<, lt, <=, le, >, gt, >=, ge, ==, =, eq, !=, <>, ne
```

This test also accepts a 3rd parameter, `strict` which defines if strict version parsing should be used. The default is False, but this setting as True uses more strict version parsing:

```
{{ sample_version_var is version('1.0', operator='lt', strict=True) }}
```

## Group theory tests

2.1 .

---

: In 2.5 `issubset` and `issuperset` were renamed to `subset` and `superset`

---

To see if a list includes or is included by another list, you can use 'subset' and 'superset':

```
vars:
  a: [1,2,3,4,5]
  b: [2,3]
tasks:
  - debug:
    msg: "A includes B"
    when: a is superset(b)

  - debug:
    msg: "B is included in A"
    when: b is subset(a)
```

## 2.4.

You can use *any* and *all* to check if any or all elements in a list are true or not:

```
vars:
  mylist:
    - 1
    - "{{ 3 == 3 }}"
    - True
  myotherlist:
    - False
    - True
tasks:
  - debug:
      msg: "all are true!"
      when: mylist is all

  - debug:
      msg: "at least one is true"
      when: myotherlist is any
```

## Testing paths

---

: In 2.5 the following tests were renamed to remove the `is_` prefix

---

The following tests can provide information about a path on the controller:

```
- debug:
    msg: "path is a directory"
    when: mypath is directory

- debug:
    msg: "path is a file"
    when: mypath is file

- debug:
    msg: "path is a symlink"
    when: mypath is link

- debug:
    msg: "path already exists"
    when: mypath is exists

- debug:
    msg: "path is {{ (mypath is abs) | ternary('absolute','relative') }}"

- debug:
    msg: "path is the same file as path2"
    when: mypath is same_file(path2)

- debug:
    msg: "path is a mount"
    when: mypath is mount
```

### Task results

The following tasks are illustrative of the tests meant to check the status of tasks:

```
tasks:

- shell: /usr/bin/foo
  register: result
  ignore_errors: True

- debug:
  msg: "it failed"
  when: result is failed

# in most cases you'll want a handler, but if you want to do something right now, ↵
↵this is nice
- debug:
  msg: "it changed"
  when: result is changed

- debug:
  msg: "it succeeded in Ansible >= 2.1"
  when: result is succeeded

- debug:
  msg: "it succeeded"
  when: result is success

- debug:
  msg: "it was skipped"
  when: result is skipped
```

---

: From 2.1, you can also use success, failure, change, and skip so that the grammar matches, for those who need to be strict about it.

---

:

**Working With Playbooks** An introduction to playbooks

**Conditionals** Conditional statements in playbooks

**Variables** All about variables

**Loops** Looping in playbooks

**Roles** Playbook organization by roles

**Best Practices** Best practices in playbooks

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

### Lookups

Lookup plugins allow access to outside data sources. Like all templating, these plugins are evaluated on the Ansible control machine, and can include reading the filesystem as well as contacting external datastores and services. This data is then made available using the standard templating system in Ansible.

:

- Lookups occur on the local computer, not on the remote computer.
- They are executed with in the directory containing the role or play, as opposed to local tasks which are executed with the directory of the executed script.
- You can pass `wantlist=True` to lookups to use in jinja2 template "for" loops.
- Lookups are an advanced feature. You should have a good working knowledge of Ansible plays before incorporating them.

: Some lookups pass arguments to a shell. When using variables from a remote/untrusted source, use the `lquote` filter to ensure safe usage.

### Topics

- *Lookups*
  - *Lookups and loops*
  - *Lookups and variables*

## Lookups and loops

*lookup plugins* are a way to query external data sources, such as shell commands or even key value stores.

Before Ansible 2.5, lookups were mostly used indirectly in `with_<lookup>` constructs for looping. Starting with Ansible version 2.5, lookups are used more explicitly as part of Jinja2 expressions fed into the `loop` keyword.

## Lookups and variables

One way of using lookups is to populate variables. These macros are evaluated each time they are used in a task (or template):

```
vars:
  motd_value: "{{ lookup('file', '/etc/motd') }}"
tasks:
  - debug:
      msg: "motd value is {{ motd_value }}"
```

For more details and a complete list of lookup plugins available, please see *Working With Plugins*.

:

*Working With Playbooks* An introduction to playbooks

*Conditionals* Conditional statements in playbooks

*Variables* All about variables

*Loops* Looping in playbooks

*User Mailing List* Have a question? Stop by the google group!

[irc.freenode.net](#) #ansible IRC chat channel

## Python Version and Templating

Jinja2 templates leverage Python data types and standard functions. This makes for a rich set of operations that can be performed on data. However, this also means that certain specifics of the underlying Python becomes visible to template authors. Since Ansible playbooks use Jinja2 for templates and variables, this means that playbook authors need to be aware of these specifics as well.

Unless otherwise noted, these differences are only of interest when running Ansible in Python2 versus Python3. Changes within Python2 and Python3 are generally small enough that they are not visible at the Jinja2 level.

## Dictionary Views

In Python2, the `dict.keys()`, `dict.values()`, and `dict.items()` methods return a list. Jinja2 returns that to Ansible via a string representation that Ansible can turn back into a list. In Python3, those methods return a [dictionary view](#) object. The string representation that Jinja2 returns for dictionary views cannot be parsed back into a list by Ansible. It is, however, easy to make this portable by using the `list` filter whenever using `dict.keys()`, `dict.values()`, or `dict.items()`:

```
vars:
  hosts:
    testhost1: 127.0.0.2
    testhost2: 127.0.0.3
tasks:
  - debug:
      msg: '{{ item }}'
      # Only works with Python 2
      #loop: "{{ hosts.keys() }}"
      # Works with both Python 2 and Python 3
      loop: "{{ hosts.keys() | list }}"
```

## `dict.iteritems()`

In Python2, dictionaries have `iterkeys()`, `itervalues()`, and `iteritems()` methods. These methods have been removed in Python3. Playbooks and Jinja2 templates should use `dict.keys()`, `dict.values()`, and `dict.items()` in order to be compatible with both Python2 and Python3:

```
vars:
  hosts:
    testhost1: 127.0.0.2
    testhost2: 127.0.0.3
tasks:
  - debug:
      msg: '{{ item }}'
      # Only works with Python 2
      #loop: "{{ hosts.iteritems() }}"
      # Works with both Python 2 and Python 3
      loop: "{{ hosts.items() | list }}"
```

:

- The [Dictionary Views](#) entry for information on why the `list` filter is necessary here.



:

*Working With Playbooks* An introduction to playbooks

*Conditionals* Conditional statements in playbooks

*Loops* Looping in playbooks

*Roles* Playbook organization by roles

*Best Practices* Best practices in playbooks

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Conditionals

### Topics

- *Conditionals*
  - *The When Statement*
  - *Loops and Conditionals*
  - *Loading in Custom Facts*
  - *Applying 'when' to roles, imports, and includes*
  - *Conditional Imports*
  - *Selecting Files And Templates Based On Variables*
  - *Register Variables*
  - *Commonly Used Facts*
    - \* *ansible\_distribution*
    - \* *ansible\_distribution\_major\_version*
    - \* *ansible\_os\_family*

Often the result of a play may depend on the value of a variable, fact (something learned about the remote system), or previous task result. In some cases, the values of variables may depend on other variables. Additional groups can be created to manage hosts based on whether the hosts match other criteria. This topic covers how conditionals are used in playbooks.

---

: There are many options to control execution flow in Ansible. More examples of supported conditionals can be located here: <http://jinja.pocoo.org/docs/dev/templates/#comparisons>.

---

## The When Statement

Sometimes you will want to skip a particular step on a particular host. This could be something as simple as not installing a certain package if the operating system is a particular version, or it could be something like performing some cleanup steps if a filesystem is getting full.

This is easy to do in Ansible with the *when* clause, which contains a raw Jinja2 expression without double curly braces (see *Variables*). It's actually pretty simple:

```
tasks:
- name: "shut down Debian flavored systems"
  command: /sbin/shutdown -t now
  when: ansible_os_family == "Debian"
  # note that Ansible facts and vars like ansible_os_family can be used
  # directly in conditionals without double curly braces
```

You can also use parentheses to group conditions:

```
tasks:
- name: "shut down CentOS 6 and Debian 7 systems"
  command: /sbin/shutdown -t now
  when: (ansible_distribution == "CentOS" and ansible_distribution_major_version ==
↪ "6") or
      (ansible_distribution == "Debian" and ansible_distribution_major_version ==
↪ "7")
```

Multiple conditions that all need to be true (a logical 'and') can also be specified as a list:

```
tasks:
- name: "shut down CentOS 6 systems"
  command: /sbin/shutdown -t now
  when:
    - ansible_distribution == "CentOS"
    - ansible_distribution_major_version == "6"
```

A number of Jinja2 "filters" can also be used in when statements, some of which are unique and provided by Ansible. Suppose we want to ignore the error of one statement and then decide to do something conditionally based on success or failure:

```
tasks:
- command: /bin/false
  register: result
  ignore_errors: True

- command: /bin/something
  when: result is failed

  # In older versions of ansible use ``success``, now both are valid but succeeded_
↪ uses the correct tense.
- command: /bin/something_else
  when: result is succeeded

- command: /bin/still/something_else
  when: result is skipped
```

---

: both *success* and *succeeded* work (*faillfailed*, etc).

---

As a reminder, to see what facts are available on a particular system, you can do the following:

```
ansible hostname.example.com -m setup
```

Tip: Sometimes you'll get back a variable that's a string and you'll want to do a math operation comparison on it. You can do this like so:

```
tasks:
  - shell: echo "only on Red Hat 6, derivatives, and later"
    when: ansible_os_family == "RedHat" and ansible_lsb.major_release|int >= 6
```

: the above example requires the `lsb_release` package on the target host in order to return the `ansible_lsb.major_release` fact.

Variables defined in the playbooks or inventory can also be used. An example may be the execution of a task based on a variable's boolean value:

```
vars:
  epic: true
```

Then a conditional execution might look like:

```
tasks:
  - shell: echo "This certainly is epic!"
    when: epic
```

or:

```
tasks:
  - shell: echo "This certainly isn't epic!"
    when: not epic
```

If a required variable has not been set, you can skip or fail using Jinja2's *defined* test. For example:

```
tasks:
  - shell: echo "I've got '{{ foo }}' and am not afraid to use it!"
    when: foo is defined

  - fail: msg="Bailing out. this play requires 'bar'"
    when: bar is undefined
```

This is especially useful in combination with the conditional import of vars files (see below). As the examples show, you don't need to use `{{ }}` to use variables inside conditionals, as these are already implied.

## Loops and Conditionals

Combining *when* with loops (see [Loops](#)), be aware that the *when* statement is processed separately for each item. This is by design:

```
tasks:
  - command: echo {{ item }}
    loop: [ 0, 2, 4, 6, 8, 10 ]
    when: item > 5
```

If you need to skip the whole task depending on the loop variable being defined, used the *default* filter to provide an empty iterator:

```
- command: echo {{ item }}
  loop: "{{ mylist|default([]) }}"
  when: item > 5
```

If using a dict in a loop:

```
- command: echo {{ item.key }}
  loop: "{{ query('dict', mydict|default({})) }}"
  when: item.value > 5
```

### Loading in Custom Facts

It's also easy to provide your own facts if you want, which is covered in [Developing Modules](#). To run them, just make a call to your own custom fact gathering module at the top of your list of tasks, and variables returned there will be accessible to future tasks:

```
tasks:
  - name: gather site specific fact data
    action: site_facts
  - command: /usr/bin/thingy
    when: my_custom_fact_just_retrieved_from_the_remote_system == '1234'
```

### Applying 'when' to roles, imports, and includes

Note that if you have several tasks that all share the same conditional statement, you can affix the conditional to a task include statement as below. All the tasks get evaluated, but the conditional is applied to each and every task:

```
- import_tasks: tasks/sometasks.yml
  when: "'reticulating splines' in output"
```

---

: In versions prior to 2.0 this worked with task includes but not playbook includes. 2.0 allows it to work with both.

---

Or with a role:

```
- hosts: webservers
  roles:
    - role: debian_stock_config
      when: ansible_os_family == 'Debian'
```

You will note a lot of 'skipped' output by default in Ansible when using this approach on systems that don't match the criteria. Read up on the 'group\_by' module in the [Working With Modules](#) docs for a more streamlined way to accomplish the same thing.

When used with *include\_\** tasks instead of imports, the conditional is applied *\_only\_* to the include task itself and not any other tasks within the included file(s). A common situation where this distinction is important is as follows:

```
# include a file to define a variable when it is not already defined

# main.yml
- include_tasks: other_tasks.yml
  when: x is not defined

# other_tasks.yml
- set_fact:
    x: foo
- debug:
    var: x
```

In the above example, if `import_tasks` had been used instead both included tasks would have also been skipped. With `include_tasks` instead, the tasks are executed as expected because the conditional is not applied to them.

## Conditional Imports

---

: This is an advanced topic that is infrequently used.

---

Sometimes you will want to do certain things differently in a playbook based on certain criteria. Having one playbook that works on multiple platforms and OS versions is a good example.

As an example, the name of the Apache package may be different between CentOS and Debian, but it is easily handled with a minimum of syntax in an Ansible Playbook:

```
---
- hosts: all
  remote_user: root
  vars_files:
    - "vars/common.yml"
    - [ "vars/{{ ansible_os_family }}.yml", "vars/os_defaults.yml" ]
  tasks:
    - name: make sure apache is started
      service: name={{ apache }} state=started
```

---

: The variable `'ansible_os_family'` is being interpolated into the list of filenames being defined for `vars_files`.

---

As a reminder, the various YAML files contain just keys and values:

```
---
# for vars/RedHat.yml
apache: httpd
somethingelse: 42
```

How does this work? For Red Hat operating systems ('CentOS', for example), the first file Ansible tries to import is `'vars/RedHat.yml'`. If that file does not exist, Ansible attempts to load `'vars/os_defaults.yml'`. If no files in the list were found, an error is raised.

On Debian, Ansible first looks for `'vars/Debian.yml'` instead of `'vars/RedHat.yml'`, before falling back on `'vars/os_defaults.yml'`.

Ansible's approach to configuration – separating variables from tasks, keeping your playbooks from turning into arbitrary code with nested conditionals - results in more streamlined and auditable configuration rules because there are fewer decision points to track.

## Selecting Files And Templates Based On Variables

---

: This is an advanced topic that is infrequently used. You can probably skip this section.

---

Sometimes a configuration file you want to copy, or a template you will use may depend on a variable. The following construct selects the first available file appropriate for the variables of a given host, which is often much cleaner than putting a lot of if conditionals in a template.

The following example shows how to template out a configuration file that was very different between, say, CentOS and Debian:

```
- name: template a file
  template:
    src: "{{ item }}"
    dest: /etc/myapp/foo.conf
  loop: "{{ query('first_found', { 'files': myfiles, 'paths': mypaths}) }}"
  vars:
    myfiles:
      - "{{ansible_distribution}}.conf"
      - default.conf
    mypaths: ['search_location_one/somedir/', '/opt/other_location/somedir/']
```

## Register Variables

Often in a playbook it may be useful to store the result of a given command in a variable and access it later. Use of the `command` module in this way can in many ways eliminate the need to write site specific facts, for instance, you could test for the existence of a particular program.

The `'register'` keyword decides what variable to save a result in. The resulting variables can be used in templates, action lines, or *when* statements. It looks like this (in an obviously trivial example):

```
- name: test play
  hosts: all

  tasks:

    - shell: cat /etc/motd
      register: motd_contents

    - shell: echo "motd contains the word hi"
      when: motd_contents.stdout.find('hi') != -1
```

As shown previously, the registered variable's string contents are accessible with the `'stdout'` value. The registered result can be used in the loop of a task if it is converted into a list (or already is a list) as shown below. `"stdout_lines"` is already available on the object as well though you could also call `"home_dirs.stdout.split()"` if you wanted, and could split by other fields:

```
- name: registered variable usage as a loop list
  hosts: all
  tasks:

    - name: retrieve the list of home directories
      command: ls /home
      register: home_dirs

    - name: add home dirs to the backup spooler
      file:
        path: /mnt/bkspool/{{ item }}
        src: /home/{{ item }}
        state: link
      loop: "{{ home_dirs.stdout_lines }}"
      # same as loop: "{{ home_dirs.stdout.split() }}"
```

As shown previously, the registered variable's string contents are accessible with the `'stdout'` value. You may check the registered variable's string contents for emptiness:

```
- name: check registered variable for emptiness
  hosts: all

  tasks:

    - name: list contents of directory
      command: ls mydir
      register: contents

    - name: check contents for emptiness
      debug:
        msg: "Directory is empty"
      when: contents.stdout == ""
```

## Commonly Used Facts

The following Facts are frequently used in Conditionals - see above for examples.

### **ansible\_distribution**

Possible values:

```
Alpine
Altlinux
Amazon
Archlinux
ClearLinux
Coreos
Debian
Fedora
Gentoo
Mandriva
NA
OpenWrt
OracleLinux
RedHat
Slackware
SMGL
SUSE
VMwareESX
```

### **ansible\_distribution\_major\_version**

This will be the major version of the operating system. For example, the value will be *16* for Ubuntu 16.04.

### **ansible\_os\_family**

Possible values:

AIX  
Alpine  
Altlinux  
Archlinux  
Darwin  
Debian  
FreeBSD  
Gentoo  
HP-UX  
Mandrake  
RedHat  
SGML  
Slackware  
Solaris  
Suse

:

***Working With Playbooks*** An introduction to playbooks

***Roles*** Playbook organization by roles

***Best Practices*** Best practices in playbooks

***Variables*** All about variables

***User Mailing List*** Have a question? Stop by the google group!

***irc.freenode.net*** #ansible IRC chat channel

## Loops

Often you'll want to do many things in one task, such as create a lot of users, install a lot of packages, or repeat a polling step until a certain result is reached.

This chapter is all about how to use loops in playbooks.

### Topics

- *Loops*
  - *Standard Loops*
  - *Complex loops*
  - *Using lookup vs query with loop*
  - *Do-Until Loops*
  - *Using register with a loop*
  - *Looping over the inventory*
  - *Loop Control*
  - *Migrating from with\_X to loop*
    - \* *with\_list*
    - \* *with\_items*
    - \* *with\_indexed\_items*



```

* with_flattened
* with_together
* with_dict
* with_sequence
* with_subelements
* with_nested/with_cartesian
* with_random_choice

```

## Standard Loops

To save some typing, repeated tasks can be written in short-hand like so:

```

- name: add several users
  user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  loop:
    - testuser1
    - testuser2

```

If you have defined a YAML list in a variables file, or the 'vars' section, you can also do:

```

loop: "{{ somelist }}"

```

The above would be the equivalent of:

```

- name: add user testuser1
  user:
    name: "testuser1"
    state: present
    groups: "wheel"
- name: add user testuser2
  user:
    name: "testuser2"
    state: present
    groups: "wheel"

```

---

: Before 2.5 Ansible mainly used the `with_<lookup>` keywords to create loops, the `loop` keyword is basically analogous to `with_list`.

---

Some plugins like, the `yum` and `apt` modules can take lists directly to their options, this is more optimal than looping over the task. See each action's documentation for details, for now here is an example:

```

- name: optimal yum
  yum:
    name: "{{ list_of_packages }}"
    state: present

```

(continues on next page)

()

```
- name: non optimal yum, not only slower but might cause issues with interdependencies
yum:
  name: "{{item}}"
  state: present
  loop: "{{list_of_packages}}"
```

Note that the types of items you iterate over do not have to be simple lists of strings. If you have a list of hashes, you can reference subkeys using things like:

```
- name: add several users
user:
  name: "{{ item.name }}"
  state: present
  groups: "{{ item.groups }}"
loop:
  - { name: 'testuser1', groups: 'wheel' }
  - { name: 'testuser2', groups: 'root' }
```

Also be aware that when combining *Conditionals* with a loop, the `when:` statement is processed separately for each item. See *The When Statement* for an example.

To loop over a dict, use the `dict2items` *Dict Filter*:

```
- name: create a tag dictionary of non-empty tags
set_fact:
  tags_dict: "{{ (tags_dict|default({}))|combine({item.key: item.value}) }}"
loop: "{{ tags|dict2items }}"
vars:
  tags:
    Environment: dev
    Application: payment
    Another: "{{ doesnotexist|default() }}"
when: item.value != ""
```

Here, we don't want to set empty tags, so we create a dictionary containing only non-empty tags.

## Complex loops

Sometimes you need more than what a simple list provides, you can use Jinja2 expressions to create complex lists: For example, using the 'nested' lookup, you can combine lists:

```
- name: give users access to multiple databases
mysql_user:
  name: "{{ item[0] }}"
  priv: "{{ item[1] }}.*:ALL"
  append_privs: yes
  password: "foo"
loop: "{{ ['alice', 'bob'] |product(['clientdb', 'employeedb', 'providerdb'])|list }}"
↪}}
```

: `with_` loops are actually a combination of things with `with_ + lookup()`, even `items` is a lookup. `loop` can be used in the same way as shown above.

## Using lookup vs query with loop

In Ansible 2.5 a new jinja2 function was introduced named `query`, that offers several benefits over `lookup` when using the new `loop` keyword.

This is described more in the lookup documentation, however, `query` provides a more simple interface and a more predictable output from lookup plugins, ensuring better compatibility with `loop`.

In certain situations the `lookup` function may not return a list which `loop` requires.

The following invocations are equivalent, using `wantlist=True` with `lookup` to ensure a return type of a list:

```
loop: "{{ query('inventory_hostnames', 'all') }}"
loop: "{{ lookup('inventory_hostnames', 'all', wantlist=True) }}"
```

## Do-Until Loops

### 1.4 .

Sometimes you would want to retry a task until a certain condition is met. Here's an example:

```
- shell: /usr/bin/foo
  register: result
  until: result.stdout.find("all systems go") != -1
  retries: 5
  delay: 10
```

The above example run the shell module recursively till the module's result has "all systems go" in its stdout or the task has been retried for 5 times with a delay of 10 seconds. The default value for "retries" is 3 and "delay" is 5.

The task returns the results returned by the last task run. The results of individual retries can be viewed by `-vv` option. The registered variable will also have a new key "attempts" which will have the number of the retries for the task.

---

: If the `until` parameter isn't defined, the value for the `retries` parameter is forced to 1.

---

## Using register with a loop

After using `register` with a loop, the data structure placed in the variable will contain a `results` attribute that is a list of all responses from the module.

Here is an example of using `register` with `loop`:

```
- shell: "echo {{ item }}"
  loop:
    - "one"
    - "two"
  register: echo
```

This differs from the data structure returned when using `register` without a loop:

```
{
  "changed": true,
  "msg": "All items completed",
```

(continues on next page)

```

"results": [
  {
    "changed": true,
    "cmd": "echo \"one\" ",
    "delta": "0:00:00.003110",
    "end": "2013-12-19 12:00:05.187153",
    "invocation": {
      "module_args": "echo \"one\"",
      "module_name": "shell"
    },
    "item": "one",
    "rc": 0,
    "start": "2013-12-19 12:00:05.184043",
    "stderr": "",
    "stdout": "one"
  },
  {
    "changed": true,
    "cmd": "echo \"two\" ",
    "delta": "0:00:00.002920",
    "end": "2013-12-19 12:00:05.245502",
    "invocation": {
      "module_args": "echo \"two\"",
      "module_name": "shell"
    },
    "item": "two",
    "rc": 0,
    "start": "2013-12-19 12:00:05.242582",
    "stderr": "",
    "stdout": "two"
  }
]
}

```

Subsequent loops over the registered variable to inspect the results may look like:

```

- name: Fail if return code is not 0
  fail:
    msg: "The command ({{ item.cmd }}) did not have a 0 return code"
  when: item.rc != 0
  loop: "{{ echo.results }}"

```

During iteration, the result of the current item will be placed in the variable:

```

- shell: echo "{{ item }}"
  loop:
    - one
    - two
  register: echo
  changed_when: echo.stdout != "one"

```

## Looping over the inventory

If you wish to loop over the inventory, or just a subset of it, there are multiple ways. One can use a regular `loop` with the `ansible_play_batch` or `groups` variables, like this:

```
# show all the hosts in the inventory
- debug:
    msg: "{{ item }}"
    loop: "{{ groups['all'] }}"

# show all the hosts in the current play
- debug:
    msg: "{{ item }}"
    loop: "{{ ansible_play_batch }}"
```

There is also a specific lookup plugin `inventory_hostnames` that can be used like this:

```
# show all the hosts in the inventory
- debug:
    msg: "{{ item }}"
    loop: "{{ query('inventory_hostnames', 'all') }}"

# show all the hosts matching the pattern, ie all but the group www
- debug:
    msg: "{{ item }}"
    loop: "{{ query('inventory_hostnames', 'all!www') }}"
```

More information on the patterns can be found on [Working with Patterns](#)

## Loop Control

### 2.1 .

In 2.0 you are again able to use loops and task includes (but not playbook includes). This adds the ability to loop over the set of tasks in one shot. Ansible by default sets the loop variable `item` for each loop, which causes these nested loops to overwrite the value of `item` from the "outer" loops. As of Ansible 2.1, the `loop_control` option can be used to specify the name of the variable to be used for the loop:

```
# main.yml
- include_tasks: inner.yml
  loop:
    - 1
    - 2
    - 3
  loop_control:
    loop_var: outer_item

# inner.yml
- debug:
    msg: "outer item={{ outer_item }} inner item={{ item }}"
  loop:
    - a
    - b
    - c
```

---

: If Ansible detects that the current loop is using a variable which has already been defined, it will raise an error to fail the task.

---

### 2.2 .

When using complex data structures for looping the display might get a bit too "busy", this is where the `label` directive comes to help:

```
- name: create servers
  digital_ocean:
    name: "{{ item.name }}"
    state: present
  loop:
    - name: server1
      disks: 3gb
      ram: 15Gb
      network:
        nic01: 100Gb
        nic02: 10Gb
        ...
  loop_control:
    label: "{{ item.name }}"
```

This will now display just the `label` field instead of the whole structure per `item`, it defaults to `{{ item }}` to display things as usual.

2.2 .

Another option to loop control is `pause`, which allows you to control the time (in seconds) between execution of items in a task loop.:

```
# main.yml
- name: create servers, pause 3s before creating next
  digital_ocean:
    name: "{{ item }}"
    state: present
  loop:
    - server1
    - server2
  loop_control:
    pause: 3
```

2.5 .

If you need to keep track of where you are in a loop, you can use the `index_var` option to loop control to specify a variable name to contain the current loop index.:

```
- name: count our fruit
  debug:
    msg: "{{ item }} with index {{ my_idx }}"
  loop:
    - apple
    - banana
    - pear
  loop_control:
    index_var: my_idx
```

## Migrating from `with_X` to `loop`

With the release of Ansible 2.5, the recommended way to perform loops is the use the new `loop` keyword instead of `with_X` style loops.

In many cases, `loop` syntax is better expressed using filters instead of more complex use of `query` or `lookup`.

The following examples will show how to convert many common `with_` style loops to `loop` and filters.

### with\_list

`with_list` is directly replaced by `loop`.

```
- name: with_list
  debug:
    msg: "{{ item }}"
  with_list:
    - one
    - two

- name: with_list -> loop
  debug:
    msg: "{{ item }}"
  loop:
    - one
    - two
```

### with\_items

`with_items` is replaced by `loop` and the `flatten` filter.

```
- name: with_items
  debug:
    msg: "{{ item }}"
  with_items: "{{ items }}"

- name: with_items -> loop
  debug:
    msg: "{{ item }}"
  loop: "{{ items|flatten(levels=1) }}"
```

### with\_indexed\_items

`with_indexed_items` is replaced by `loop`, the `flatten` filter and `loop_control.index_var`.

```
- name: with_indexed_items
  debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  with_indexed_items: "{{ items }}"

- name: with_indexed_items -> loop
  debug:
    msg: "{{ index }} - {{ item }}"
  loop: "{{ items|flatten(levels=1) }}"
  loop_control:
    index_var: index
```

## **with\_flattened**

`with_flattened` is replaced by `loop` and the `flatten` filter.

```
- name: with_flattened
  debug:
    msg: "{{ item }}"
  with_flattened: "{{ items }}"

- name: with_flattened -> loop
  debug:
    msg: "{{ item }}"
  loop: "{{ items|flatten }}"
```

## **with\_together**

`with_together` is replaced by `loop` and the `zip` filter.

```
- name: with_together
  debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  with_together:
    - "{{ list_one }}"
    - "{{ list_two }}"

- name: with_together -> loop
  debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  loop: "{{ list_one|zip(list_two)|list }}"
```

## **with\_dict**

`with_dict` can be substituted by `loop` and either the `dictsort` or `dict2items` filters.

```
- name: with_dict
  debug:
    msg: "{{ item.key }} - {{ item.value }}"
  with_dict: "{{ dictionary }}"

- name: with_dict -> loop (option 1)
  debug:
    msg: "{{ item.key }} - {{ item.value }}"
  loop: "{{ dictionary|dict2items }}"

- name: with_dict -> loop (option 2)
  debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  loop: "{{ dictionary|dictsort }}"
```

## **with\_sequence**

`with_sequence` is replaced by `loop` and the `range` function, and potentially the `format` filter.



```
- name: with_sequence
  debug:
    msg: "{{ item }}"
  with_sequence: start=0 end=4 stride=2 format=testuser%02x

- name: with_sequence -> loop
  debug:
    msg: "{{ 'testuser%02x' | format(item) }}"
    # range is exclusive of the end point
  loop: "{{ range(0, 4 + 1, 2)|list }}"
```

## with\_subelements

`with_subelements` is replaced by `loop` and the `subelements` filter.

```
- name: with_subelements
  debug:
    msg: "{{ item.0.name }} - {{ item.1 }}"
  with_subelements:
    - "{{ users }}"
    - mysql.hosts

- name: with_subelements -> loop
  debug:
    msg: "{{ item.0.name }} - {{ item.1 }}"
  loop: "{{ users|subelements('mysql.hosts') }}"
```

## with\_nested/with\_cartesian

`with_nested` and `with_cartesian` are replaced by `loop` and the `product` filter.

```
- name: with_nested
  debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  with_nested:
    - "{{ list_one }}"
    - "{{ list_two }}"

- name: with_nested -> loop
  debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  loop: "{{ list_one|product(list_two)|list }}"
```

## with\_random\_choice

`with_random_choice` is replaced by just use of the `random` filter, without need of `loop`.

```
- name: with_random_choice
  debug:
    msg: "{{ item }}"
  with_random_choice: "{{ my_list }}"
```

(continues on next page)

```
- name: with_random_choice -> loop (No loop is needed here)
  debug:
    msg: "{{ my_list|random }}"
  tags: random
```

:

**Working With Playbooks** An introduction to playbooks

**Roles** Playbook organization by roles

**Best Practices** Best practices in playbooks

**Conditionals** Conditional statements in playbooks

**Variables** All about variables

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Blocks

Blocks allow for logical grouping of tasks and in play error handling. Most of what you can apply to a single task can be applied at the block level, which also makes it much easier to set data or directives common to the tasks. This does not mean the directive affects the block itself, but is inherited by the tasks enclosed by a block. i.e. a *when* will be applied to the tasks, not the block itself.

### 1: Block example

```
tasks:
  - name: Install Apache
    block:
      - yum:
          name: "{{ item }}"
          state: installed
        with_items:
          - httpd
          - memcached
      - template:
          src: templates/src.j2
          dest: /etc/foo.conf
      - service:
          name: bar
          state: started
          enabled: True
    when: ansible_distribution == 'CentOS'
    become: true
    become_user: root
```

In the example above, each of the 3 tasks will be executed after appending the *when* condition from the block and evaluating it in the task's context. Also they inherit the privilege escalation directives enabling "become to root" for all the enclosed tasks.

2.3 : The `name :` keyword for `block :` was added in Ansible 2.3.

## Error Handling

Blocks also introduce the ability to handle errors in a way similar to exceptions in most programming languages.

### 2: Block error handling example

```
tasks:
- name: Attempt and graceful roll back demo
  block:
    - debug:
      msg: 'I execute normally'
    - command: /bin/false
    - debug:
      msg: 'I never execute, due to the above task failing'
  rescue:
    - debug:
      msg: 'I caught an error'
    - command: /bin/false
    - debug:
      msg: 'I also never execute :-( '
  always:
    - debug:
      msg: "This always executes"
```

The tasks in the `block` would execute normally, if there is any error the `rescue` section would get executed with whatever you need to do to recover from the previous error. The `always` section runs no matter what previous error did or did not occur in the `block` and `rescue` sections. It should be noted that the play continues if a `rescue` section completes successfully as it 'erases' the error status (but not the reporting), this means it won't trigger `max_fail_percentage` nor `any_errors_fatal` configurations but will appear in the playbook statistics.

Another example is how to run handlers after an error occurred :

### 3: Block run handlers in error handling

```
tasks:
- name: Attempt and graceful roll back demo
  block:
    - debug:
      msg: 'I execute normally'
    notify: run me even after an error
    - command: /bin/false
  rescue:
    - name: make sure all handlers run
      meta: flush_handlers
handlers:
- name: run me even after an error
  debug:
    msg: 'This handler runs even on error'
```

:

**Working With Playbooks** An introduction to playbooks

**Roles** Playbook organization by roles

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

### Advanced Playbooks Features

Here are some playbook features that not everyone may need to learn, but can be quite useful for particular applications. Browsing these topics is recommended as you may find some useful tips here, but feel free to learn the basics of Ansible first and adopt these only if they seem relevant or useful to your environment.

### Understanding Privilege Escalation

Ansible can use existing privilege escalation systems to allow a user to execute tasks as another.

#### Topics

- *Understanding Privilege Escalation*
  - *Become*
  - *Directives*
    - \* *Connection variables*
    - \* *Command line options*
    - \* *For those from Pre 1.9 , sudo and su still work!*
    - \* *Limitations*
      - *Becoming an Unprivileged User*
      - *Connection Plugin Support*
      - *Only one method may be enabled per host*
      - *Can't limit escalation to certain commands*
      - *Environment variables populated by pam\_systemd*
  - *Become and Networks*
    - \* *Setting enable mode for all tasks*
      - *Passwords for enable mode*
    - \* *authorize and auth\_pass*
  - *Become and Windows*
    - \* *Administrative Rights*
    - \* *Local Service Accounts*
    - \* *Accounts without a Password*
    - \* *Become Flags*
    - \* *Limitations*

### Become

Ansible allows you to 'become' another user, different from the user that logged into the machine (remote user). This is done using existing privilege escalation tools such as *sudo*, *su*, *pfexec*, *doas*, *pbrun*, *dzdo*, *ksu*, *runas*, *machinectl* and others.

---

: Prior to version 1.9, Ansible mostly allowed the use of *sudo* and a limited use of *su* to allow a login/remote user to become a different user and execute tasks and create resources with the second user's permissions. As of Ansible version 1.9, *become* supersedes the old *sudo*/*su*, while still being backwards compatible. This new implementation also makes it easier to add other privilege escalation tools, including *pbrun* (Powerbroker), *pfexec*, *dzdo* (Centrify), and others.

---



---

: Become vars and directives are independent. For example, setting `become_user` does not set `become`.

---

## Directives

These can be set from play to task level, but are overridden by connection variables as they can be host specific.

**become** set to `yes` to activate privilege escalation.

**become\_user** set to user with desired privileges — the user you *become*, NOT the user you login as. Does NOT imply `become: yes`, to allow it to be set at host level.

**become\_method** (at play or task level) overrides the default method set in `ansible.cfg`, set to `sudo/su/pbrun/pfexec/doas/dzdo/ksu/runas/machinectl`

**become\_flags** (at play or task level) permit the use of specific flags for the tasks or role. One common use is to change the user to nobody when the shell is set to no login. Added in Ansible 2.2.

For example, to manage a system service (which requires `root` privileges) when connected as a non-`root` user (this takes advantage of the fact that the default value of `become_user` is `root`):

```
- name: Ensure the httpd service is running
  service:
    name: httpd
    state: started
  become: yes
```

To run a command as the apache user:

```
- name: Run a command as the apache user
  command: somecommand
  become: yes
  become_user: apache
```

To do something as the nobody user when the shell is nologin:

```
- name: Run a command as nobody
  command: somecommand
  become: yes
  become_method: su
  become_user: nobody
  become_flags: '-s /bin/sh'
```

## Connection variables

Each allows you to set an option per group and/or host, these are normally defined in inventory but can be used as normal variables.

**ansible\_become** equivalent of the become directive, decides if privilege escalation is used or not.

**ansible\_become\_method** which privilege escalation method should be used

**ansible\_become\_user** set the user you become through privilege escalation; does not imply `ansible_become: yes`

**ansible\_become\_pass** set the privilege escalation password. See *Using Vault in playbooks* for details on how to avoid having secrets in plain text

For example, if you want to run all tasks as `root` on a server named `webserver`, but you can only connect as the `manager` user, you could use an inventory entry like this:

```
webserver ansible_user=manager ansible_become=yes
```

### Command line options

**--ask-become-pass, -K** ask for privilege escalation password; does not imply become will be used.  
Note that this password will be used for all hosts.

**--become, -b** run operations with become (no password implied)

**--become-method=BECOME\_METHOD** privilege escalation method to use (default=sudo), valid choices: [ sudo | su | pbrun | pfexec | doas | dzdo | ksu | runas | machinectl ]

**--become-user=BECOME\_USER** run operations as this user (default=root), does not imply `become/-b`

### For those from Pre 1.9 , sudo and su still work!

For those using old playbooks will not need to be changed, even though they are deprecated, `sudo` and `su` directives, variables and options will continue to work. It is recommended to move to `become` as they may be retired at one point. You cannot mix directives on the same object (`become` and `sudo`) though, Ansible will complain if you try to.

Become will default to using the old `sudo/su` configs and variables if they exist, but will override them if you specify any of the new ones.

### Limitations

Although privilege escalation is mostly intuitive, there are a few limitations on how it works. Users should be aware of these to avoid surprises.

### Becoming an Unprivileged User

Ansible 2.0.x and below has a limitation with regards to becoming an unprivileged user that can be a security risk if users are not aware of it. Ansible modules are executed on the remote machine by first substituting the parameters into the module file, then copying the file to the remote machine, and finally executing it there.

Everything is fine if the module file is executed without using `become`, when the `become_user` is `root`, or when the connection to the remote machine is made as `root`. In these cases the module file is created with permissions that only allow reading by the user and `root`.

The problem occurs when the `become_user` is an unprivileged user. Ansible 2.0.x and below make the module file world readable in this case, as the module file is written as the user that Ansible connects as, but the file needs to be readable by the user Ansible is set to `become`.

---

: In Ansible 2.1, this window is further narrowed: If the connection is made as a privileged user (root), then Ansible 2.1 and above will use `chown` to set the file's owner to the unprivileged user being switched to. This means both the user making the connection and the user being switched to via `become` must be unprivileged in order to trigger this problem.

---

If any of the parameters passed to the module are sensitive in nature, then those pieces of data are located in a world readable module file for the duration of the Ansible module execution. Once the module is done executing, Ansible will delete the temporary file. If you trust the client machines then there's no problem here. If you do not trust the client machines then this is a potential danger.

Ways to resolve this include:

- Use *pipelining*. When pipelining is enabled, Ansible doesn't save the module to a temporary file on the client. Instead it pipes the module to the remote python interpreter's stdin. Pipelining does not work for python modules involving file transfer (for example: `copy`, `fetch`, `template`), or for non-python modules.
- (Available in Ansible 2.1) Install POSIX.1e filesystem acl support on the managed host. If the temporary directory on the remote host is mounted with POSIX acls enabled and the `setfacl` tool is in the remote `PATH` then Ansible will use POSIX acls to share the module file with the second unprivileged user instead of having to make the file readable by everyone.
- Don't perform an action on the remote machine by becoming an unprivileged user. Temporary files are protected by UNIX file permissions when you `become` root or do not use `become`. In Ansible 2.1 and above, UNIX file permissions are also secure if you make the connection to the managed machine as root and then use `become` to an unprivileged account.

: Although the Solaris ZFS filesystem has filesystem ACLs, the ACLs are not POSIX.1e filesystem acls (they are NFSv4 ACLs instead). Ansible cannot use these ACLs to manage its temp file permissions so you may have to resort to `allow_world_readable_tmpfiles` if the remote machines use ZFS.

## 2.1 .

In addition to the additional means of doing this securely, Ansible 2.1 also makes it harder to unknowingly do this insecurely. Whereas in Ansible 2.0.x and below, Ansible will silently allow the insecure behaviour if it was unable to find another way to share the files with the unprivileged user, in Ansible 2.1 and above Ansible defaults to issuing an error if it can't do this securely. If you can't make any of the changes above to resolve the problem, and you decide that the machine you're running on is secure enough for the modules you want to run there to be world readable, you can turn on `allow_world_readable_tmpfiles` in the `ansible.cfg` file. Setting `allow_world_readable_tmpfiles` will change this from an error into a warning and allow the task to run as it did prior to 2.1.

## Connection Plugin Support

Privilege escalation methods must also be supported by the connection plugin used. Most connection plugins will warn if they do not support `become`. Some will just ignore it as they always run as root (jail, chroot, etc).

## Only one method may be enabled per host

Methods cannot be chained. You cannot use `sudo /bin/su -` to become a user, you need to have privileges to run the command as that user in `sudo` or be able to `su` directly to it (the same for `pbrun`, `pfexec` or other supported methods).

## Can't limit escalation to certain commands

Privilege escalation permissions have to be general. Ansible does not always use a specific command to do something but runs modules (code) from a temporary file name which changes every time. If you have `'/sbin/service'` or `'/bin/chmod'` as the allowed commands this will fail with ansible as those paths won't match with the temporary file that ansible creates to run the module.

## Environment variables populated by pam\_systemd

For most Linux distributions using `systemd` as their init, the default methods used by `become` do not open a new "session", in the sense of `systemd`. Because the `pam_systemd` module will not fully initialize a new session, you might have surprises compared to a normal session opened through `ssh`: some environment variables set by `pam_systemd`, most notably `XDG_RUNTIME_DIR`, are not populated for the new user and instead inherited or just emptied.

This might cause trouble when trying to invoke `systemd` commands that depend on `XDG_RUNTIME_DIR` to access the bus:

```
$ echo $XDG_RUNTIME_DIR

$ systemctl --user status
Failed to connect to bus: Permission denied
```

To force `become` to open a new `systemd` session that goes through `pam_systemd`, you can use `become_method: machinectl`.

For more information, see [this systemd issue](#).

## Become and Networks

As of version 2.6, Ansible supports `become` for privilege escalation (entering `enable` mode or privileged EXEC mode) on all Ansible-maintained platforms that support `enable` mode: *eos*, *ios*, and *nxos*. Using `become` replaces the `authorize` and `auth_pass` options in a provider dictionary.

You must set the connection type to either `connection: network_cli` or `connection: httpapi` to use `become` for privilege escalation on network devices. Check the [Platform Options](#) and `network_modules` documentation for details.

You can use escalated privileges on only the specific tasks that need them, on an entire play, or on all plays. Adding `become: yes` and `become_method: enable` instructs Ansible to enter `enable` mode before executing the task, play, or playbook where those parameters are set.

If you see this error message, the task that generated it requires `enable` mode to succeed:

```
Invalid input (privileged mode required)
```

To set `enable` mode for a specific task, add `become` at the task level:

```
- name: Gather facts (eos)
  eos_facts:
    gather_subset:
      - "!hardware"
  become: yes
  become_method: enable
```

To set `enable` mode for all tasks in a single play, add `become` at the play level:



```
- hosts: eos-switches
  become: yes
  become_method: enable
  tasks:
    - name: Gather facts (eos)
      eos_facts:
        gather_subset:
          - "!hardware"
```

## Setting enable mode for all tasks

Often you wish for all tasks in all plays to run using privilege mode, that is best achieved by using `group_vars`:

### `group_vars/eos.yml`

```
ansible_connection: network_cli
ansible_network_os: eos
ansible_user: myuser
ansible_become: yes
ansible_become_method: enable
```

## Passwords for enable mode

If you need a password to enter `enable` mode, you can specify it in one of two ways:

- providing the `--ask-become-pass` command line option
- setting the `ansible_become_pass` connection variable

: As a reminder passwords should never be stored in plain text. For information on encrypting your passwords and other secrets with Ansible Vault, see [Using Vault in playbooks](#).

## authorize and auth\_pass

Ansible still supports `enable` mode with `connection: local` for legacy playbooks. To enter `enable` mode with `connection: local`, use the module options `authorize` and `auth_pass`:

```
- hosts: eos-switches
  ansible_connection: local
  tasks:
    - name: Gather facts (eos)
      eos_facts:
        gather_subset:
          - "!hardware"
      provider:
        authorize: yes
        auth_pass: "{{ secret_auth_pass }}"
```

We recommend updating your playbooks to use `become` for network-device `enable` mode consistently. The use of `authorize` and of `provider` dictionaries will be deprecated in future. Check the [Platform Options](#) and [network\\_modules](#) documentation for details.

### Become and Windows

Since Ansible 2.3, `become` can be used on Windows hosts through the `runas` method. Become on Windows uses the same inventory setup and invocation arguments as `become` on a non-Windows host, so the setup and variable names are the same as what is defined in this document.

While `become` can be used to assume the identity of another user, there are other uses for it with Windows hosts. One important use is to bypass some of the limitations that are imposed when running on WinRM, such as constrained network delegation or accessing forbidden system calls like the WUA API. You can use `become` with the same user as `ansible_user` to bypass these limitations and run commands that are not normally accessible in a WinRM session.

---

: Prior to Ansible 2.4, `become` would only work when `ansible_winrm_transport` was set to either `basic` or `credssp`, but since Ansible 2.4 `become` now works on all transport types.

---

### Administrative Rights

Many tasks in Windows require administrative privileges to complete. When using the `runas` `become` method, Ansible will attempt to run the module with the full privileges that are available to the remote user. If it fails to elevate the user token, it will continue to use the limited token during execution.

Before Ansible 2.5, a token was only able to be elevated when UAC was disabled or the remote user had the `SeTcbPrivilege` assigned. This restriction has been lifted in Ansible 2.5 and a user that is a member of the `BUILTIN\Administrators` group should have an elevated token during the module execution.

To determine the type of token that Ansible was able to get, run the following task and check the output:

```
- win_whoami:
  become: yes
```

Under the `GROUP_INFORMATION` section, the `Mandatory_Label` entry determines whether the user has Administrative rights. Here are the labels that can be returned and what they mean:

- **Medium:** Ansible failed to get an elevated token and ran under a limited token. Only a subset of the privileges assigned to user are available during the module execution and the user does not have administrative rights.
- **High:** An elevated token was used and all the privileges assigned to the user are available during the module execution.
- **System:** The `NT AUTHORITY\System` account is used and has the highest level of privileges available.

The output will also show the list of privileges that have been granted to the user. When `State==Disabled`, the privileges have not been enabled but can be if required. In most scenarios these privileges are automatically enabled when required.

If running on a version of Ansible that is older than 2.5 or the normal `runas` escalation process fails, an elevated token can be retrieved by:

- Set the `become_user` to `System` which has full control over the operating system.
- Grant `SeTcbPrivilege` to the user Ansible connects with on WinRM. `SeTcbPrivilege` is a high-level privilege that grants full control over the operating system. No user is given this privilege by default, and care should be taken if you grant this privilege to a user or group. For more information on this privilege, please see [Act as part of the operating system](#). You can use the below task to set this privilege on a Windows host:

```
- name: grant the ansible user the SeTcbPrivilege right
  win_user_right:
    name: SeTcbPrivilege
    users: '{{ansible_user}}'
    action: add
```

- Turn UAC off on the host and reboot before trying to become the user. UAC is a security protocol that is designed to run accounts with the least privilege principle. You can turn UAC off by running the following tasks:

```
- name: turn UAC off
  win_regedit:
    path: HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\policies\system
    name: EnableLUA
    data: 0
    type: dword
    state: present
    register: uac_result

- name: reboot after disabling UAC
  win_reboot:
    when: uac_result is changed
```

---

: Granting the `SeTcbPrivilege` or turning UAC off can cause Windows security vulnerabilities and care should be given if these steps are taken.

---

## Local Service Accounts

Prior to Ansible version 2.5, `become` only worked with a local or domain user account. Local service accounts like `System` or `NetworkService` could not be used as `become_user` in these older versions. This restriction has been lifted since the 2.5 release of Ansible. The three service accounts that can be set under `become_user` are:

- `System`
- `NetworkService`
- `LocalService`

Because local service accounts do not have passwords, the `ansible_become_password` parameter is not required and is ignored if specified.

## Accounts without a Password

: As a general security best practice, you should avoid allowing accounts without passwords.

Ansible can be used to become an account that does not have a password (like the `Guest` account). To become an account without a password, set up the variables like normal but either do not define `ansible_become_pass` or set `ansible_become_pass: ''`.

Before `become` can work on an account like this, the local policy [Accounts: Limit local account use of blank passwords to console logon only](#) must be disabled. This can either be done through a Group Policy Object (GPO) or with this Ansible task:

```
- name: allow blank password on become
  win_regedit:
    path: HKLM:\SYSTEM\CurrentControlSet\Control\Lsa
    name: LimitBlankPasswordUse
    data: 0
    type: dword
    state: present
```

---

: This is only for accounts that do not have a password. You still need to set the account's password under `ansible_become_pass` if the `become_user` has a password.

---

### Become Flags

Ansible 2.5 adds the `become_flags` parameter to the `runas` `become` method. This parameter can be set using the `become_flags` task directive or set in Ansible's configuration using `ansible_become_flags`. The two valid values that are initially supported for this parameter are `logon_type` and `logon_flags`.

---

: These flags should only be set when becoming a normal user account, not a local service account like `LocalSystem`.

---

The key `logon_type` sets the type of logon operation to perform. The value can be set to one of the following:

- `interactive`: The default logon type. The process will be run under a context that is the same as when running a process locally. This bypasses all WinRM restrictions and is the recommended method to use.
- `batch`: Runs the process under a batch context that is similar to a scheduled task with a password set. This should bypass most WinRM restrictions and is useful if the `become_user` is not allowed to log on interactively.
- `new_credentials`: Runs under the same credentials as the calling user, but outbound connections are run under the context of the `become_user` and `become_password`, similar to `runas.exe /netonly`. The `logon_flags` flag should also be set to `netcredentials_only`. Use this flag if the process needs to access a network resource (like an SMB share) using a different set of credentials.
- `network`: Runs the process under a network context without any cached credentials. This results in the same type of logon session as running a normal WinRM process without credential delegation, and operates under the same restrictions.
- `network_cleartext`: Like the `network` logon type, but instead caches the credentials so it can access network resources. This is the same type of logon session as running a normal WinRM process with credential delegation.

For more information, see [dwLogonType](#).

The `logon_flags` key specifies how Windows will log the user on when creating the new process. The value can be set to none or multiple of the following:

- `with_profile`: The default logon flag set. The process will load the user's profile in the `HKEY_USERS` registry key to `HKEY_CURRENT_USER`.
- `netcredentials_only`: The process will use the same token as the caller but will use the `become_user` and `become_password` when accessing a remote resource. This is useful in inter-domain scenarios where there is no trust relationship, and should be used with the `new_credentials` `logon_type`.

By default `logon_flags=with_profile` is set, if the profile should not be loaded set `logon_flags=` or if the profile should be loaded with `netcredentials_only`, set `logon_flags=with_profile, netcredentials_only`.

For more information, see [dwLogonFlags](#).

Here are some examples of how to use `become_flags` with Windows tasks:

```
- name: copy a file from a fileshare with custom credentials
  win_copy:
    src: \\server\share\data\file.txt
    dest: C:\temp\file.txt
    remote_src: yex
  vars:
    ansible_become: yes
    ansible_become_method: runas
    ansible_become_user: DOMAIN\user
    ansible_become_pass: Password01
    ansible_become_flags: logon_type=new_credentials logon_flags=netcredentials_only

- name: run a command under a batch logon
  win_whoami:
    become: yes
    become_flags: logon_type=batch

- name: run a command and not load the user profile
  win_whomai:
    become: yes
    become_flags: logon_flags=
```

## Limitations

Be aware of the following limitations with `become` on Windows:

- Running a task with `async` and `become` on Windows Server 2008, 2008 R2 and Windows 7 only works when using Ansible 2.7 or newer.
- By default, the `become` user logs on with an interactive session, so it must have the right to do so on the Windows host. If it does not inherit the `SeAllowLogOnLocally` privilege or inherits the `SeDenyLogOnLocally` privilege, the `become` process will fail. Either add the privilege or set the `logon_type` flag to change the logon type used.
- Prior to Ansible version 2.3, `become` only worked when `ansible_winrm_transport` was either `basic` or `credssp`. This restriction has been lifted since the 2.4 release of Ansible for all hosts except Windows Server 2008 (non R2 version).

:

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**webchat.freenode.net** #ansible IRC chat channel

## Asynchronous Actions and Polling

By default tasks in playbooks block, meaning the connections stay open until the task is done on each node. This may not always be desirable, or you may be running operations that take longer than the SSH timeout.

To avoid blocking or timeout issues, you can use asynchronous mode to run all of your tasks at once and then poll until they are done.

To launch a task asynchronously, specify its maximum runtime and how frequently you would like to poll for status. The default poll value is 10 seconds if you do not specify a value for *poll*:

```
---
- hosts: all
  remote_user: root

  tasks:

  - name: simulate long running op (15 sec), wait for up to 45 sec, poll every 5 sec
    command: /bin/sleep 15
    async: 45
    poll: 5
```

---

: There is no default for the async time limit. If you leave off the 'async' keyword, the task runs synchronously, which is Ansible's default.

---

Alternatively, if you do not need to wait on the task to complete, you may run the task asynchronously by specifying a poll value of 0:

```
---
- hosts: all
  remote_user: root

  tasks:

  - name: simulate long running op, allow to run for 45 sec, fire and forget
    command: /bin/sleep 15
    async: 45
    poll: 0
```

---

: You shouldn't attempt run a task asynchronously by specifying a poll value of 0:: to with operations that require exclusive locks (such as yum transactions) if you expect to run other commands later in the playbook against those same resources.

---

---

: Using a higher value for `--forks` will result in kicking off asynchronous tasks even faster. This also increases the efficiency of polling.

---

If you would like to perform a task asynchronously and check on it later you can perform a task similar to the following:

```
---
# Requires ansible 1.8+
- name: 'YUM - async task'
  yum:
    name: docker-io
    state: installed
    async: 1000
```

(continues on next page)

()

```

poll: 0
register: yum_sleeper

- name: 'YUM - check on async task'
  async_status:
    jid: "{{ yum_sleeper.ansible_job_id }}"
  register: job_result
  until: job_result.finished
  retries: 30

```

: If the value of `async:` is not high enough, this will cause the "check on it later" task to fail because the temporary status file that the `async_status:` is looking for will not have been written or no longer exist

If you would like to run multiple asynchronous tasks while limiting the amount of tasks running concurrently, you can do it this way:

```

#####
# main.yml
#####
- name: Run items asynchronously in batch of two items
  vars:
    sleep_durations:
      - 1
      - 2
      - 3
      - 4
      - 5
    durations: "{{ item }}"
  include_tasks: execute_batch.yml
  loop:
    - "{{ sleep_durations | batch(2) | list }}"

#####
# execute_batch.yml
#####
- name: Async sleeping for batched_items
  command: sleep {{ async_item }}
  async: 45
  poll: 0
  loop: "{{ durations }}"
  loop_control:
    loop_var: "async_item"
  register: async_results

- name: Check sync status
  async_status:
    jid: "{{ async_result_item.ansible_job_id }}"
  loop: "{{ async_results.results }}"
  loop_control:
    loop_var: "async_result_item"
  register: async_poll_results
  until: async_poll_results.finished
  retries: 30

```

:

*Working With Playbooks* An introduction to playbooks

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

### Check Mode ("Dry Run")

1.1 .

#### Topics

- *Check Mode ("Dry Run")*
  - *Enabling or disabling check mode for tasks*
  - *Information about check mode in variables*
  - *Showing Differences with `--diff`*

When `ansible-playbook` is executed with `--check` it will not make any changes on remote systems. Instead, any module instrumented to support 'check mode' (which contains most of the primary core modules, but it is not required that all modules do this) will report what changes they would have made rather than making them. Other modules that do not support check mode will also take no action, but just will not report what changes they might have made.

Check mode is just a simulation, and if you have steps that use conditionals that depend on the results of prior commands, it may be less useful for you. However it is great for one-node-at-time basic configuration management use cases.

Example:

```
ansible-playbook foo.yml --check
```

### Enabling or disabling check mode for tasks

2.2 .

Sometimes you may want to modify the check mode behavior of individual tasks. This is done via the `check_mode` option, which can be added to tasks.

There are two options:

1. Force a task to **run in check mode**, even when the playbook is called **without** `--check`. This is called `check_mode: yes`.
2. Force a task to **run in normal mode** and make changes to the system, even when the playbook is called **with** `--check`. This is called `check_mode: no`.

---

: Prior to version 2.2 only the equivalent of `check_mode: no` existed. The notation for that was `always_run: yes`.

---

Instead of `yes/no` you can use a Jinja2 expression, just like the `when` clause.

Example:



```
tasks:
- name: this task will make changes to the system even in check mode
  command: /something/to/run --even-in-check-mode
  check_mode: no

- name: this task will always run under checkmode and not change the system
  lineinfile:
    line: "important config"
    dest: /path/to/myconfig.conf
    state: present
  check_mode: yes
```

Running single tasks with `check_mode: yes` can be useful to write tests for ansible modules, either to test the module itself or to the conditions under which a module would make changes. With `register` (see [Conditionals](#)) you can check the potential changes.

### Information about check mode in variables

#### 2.1 .

If you want to skip, or ignore errors on some tasks in check mode you can use a boolean magic variable `ansible_check_mode` which will be set to `True` during check mode.

Example:

```
tasks:

- name: this task will be skipped in check mode
  git:
    repo: ssh://git@github.com/mylogin/hello.git
    dest: /home/mylogin/hello
    when: not ansible_check_mode

- name: this task will ignore errors in check mode
  git:
    repo: ssh://git@github.com/mylogin/hello.git
    dest: /home/mylogin/hello
    ignore_errors: "{{ ansible_check_mode }}"
```

### Showing Differences with `--diff`

#### 1.1 .

The `--diff` option to `ansible-playbook` works great with `--check` (detailed above) but can also be used by itself. When this flag is supplied and the module supports this, Ansible will report back the changes made or, if used with `--check`, the changes that would have been made. This is mostly used in modules that manipulate files (i.e. `template`) but other modules might also show 'before and after' information (i.e. `user`). Since the diff feature produces a large amount of output, it is best used when checking a single host at a time. For example:

```
ansible-playbook foo.yml --check --diff --limit foo.example.com
```

#### 2.4 .

The `--diff` option can reveal sensitive information. This option can be disabled for tasks by specifying `diff: no`.

Example:

```
tasks:
- name: this task will not report a diff when the file changes
  template:
    src: secret.conf.j2
    dest: /etc/secret.conf
    owner: root
    group: root
    mode: '0600'
  diff: no
```

## Playbook Debugger

### Topics

- *Playbook Debugger*
  - *Using the debugger keyword*
    - \* *On a task*
    - \* *On a play*
  - *Configuration or environment variable*
  - *As a Strategy*
  - *Examples*
  - *Available Commands*
    - \* *p(print) task/task\_vars/host/result*
    - \* *task.args[key] = value*
    - \* *task\_vars[key] = value*
    - \* *r(edo)*
    - \* *c(ontinue)*
    - \* *q(uit)*
  - *Use with the free strategy*

Ansible includes a debugger as part of the strategy plugins. This debugger enables you to debug as task. You have access to all of the features of the debugger in the context of the task. You can then, for example, check or set the value of variables, update module arguments, and re-run the task with the new variables and arguments to help resolve the cause of the failure.

There are multiple ways to invoke the debugger.

### Using the debugger keyword

2.5 .

The `debugger` keyword can be used on any block where you provide a `name` attribute, such as a play, role, block or task.

The `debugger` keyword accepts several values:

**always** Always invoke the debugger, regardless of the outcome

**never** Never invoke the debugger, regardless of the outcome

**on\_failed** Only invoke the debugger if a task fails

**on\_unreachable** Only invoke the debugger if the a host was unreachable

**on\_skipped** Only invoke the debugger if the task is skipped

These options override any global configuration to enable or disable the debugger.

### On a task

```
- name: Execute a command
  command: false
  debugger: on_failed
```

### On a play

```
- name: Play
  hosts: all
  debugger: on_skipped
  tasks:
    - name: Execute a command
      command: true
      when: False
```

When provided at a generic level and a more specific level, the more specific wins:

```
- name: Play
  hosts: all
  debugger: never
  tasks:
    - name: Execute a command
      command: false
      debugger: on_failed
```

## Configuration or environment variable

2.5 .

In ansible.cfg:

```
[defaults]
enable_task_debugger = True
```

As an environment variable:

```
ANSIBLE_ENABLE_TASK_DEBUGGER=True ansible-playbook -i hosts site.yml
```

When using this method, any failed or unreachable task will invoke the debugger, unless otherwise explicitly disabled.

## As a Strategy

: This is a backwards compatible method, to match Ansible versions before 2.5, and may be removed in a future release

---

To use the debug strategy, change the `strategy` attribute like this:

```
- hosts: test
  strategy: debug
  tasks:
  ...
```

If you don't want change the code, you can define `ANSIBLE_STRATEGY=debug` environment variable in order to enable the debugger, or modify `ansible.cfg` such as:

```
[defaults]
strategy = debug
```

## Examples

For example, run the playbook below:

```
- hosts: test
  debugger: on_failed
  gather_facts: no
  vars:
    var1: value1
  tasks:
    - name: wrong variable
      ping: data={{ wrong_var }}
```

The debugger is invoked since the `wrong_var` variable is undefined.

Let's change the module's arguments and run the task again

```
PLAY *****

TASK [wrong variable] *****
fatal: [192.0.2.10]: FAILED! => {"failed": true, "msg": "ERROR! 'wrong_var' is_
↪undefined"}
Debugger invoked
[192.0.2.10] TASK: wrong variable (debug)> p result._result
{'failed': True,
 'msg': 'The task includes an option with an undefined variable. The error '
       'was: 'wrong_var' is undefined\n'
       '\n'
       'The error appears to have been in '
       '"playbooks/debugger.yml": line 7, '
       'column 7, but may\n'
       'be elsewhere in the file depending on the exact syntax problem.\n'
       '\n'
       'The offending line appears to be:\n'
       '\n'
       '  tasks:\n'
```

(continues on next page)

()

```

    '    - name: wrong variable\n'
    '    ^ here\n'}
[192.0.2.10] TASK: wrong variable (debug)> p task.args
{'u'data': u'{{ wrong_var }}'}
[192.0.2.10] TASK: wrong variable (debug)> task.args['data'] = '{{ var1 }}'
[192.0.2.10] TASK: wrong variable (debug)> p task.args
{'u'data': '{{ var1 }}'}
[192.0.2.10] TASK: wrong variable (debug)> redo
ok: [192.0.2.10]

PLAY RECAP *****
192.0.2.10          : ok=1    changed=0    unreachable=0    failed=0

```

This time, the task runs successfully!

## Available Commands

### **p(print) *task/task\_vars/host/result***

Print values used to execute a module:

```

[192.0.2.10] TASK: install package (debug)> p task
TASK: install package
[192.0.2.10] TASK: install package (debug)> p task.args
{'u'name': u'{{ pkg_name }}'}
[192.0.2.10] TASK: install package (debug)> p task_vars
{'u'ansible_all_ipv4_addresses': [u'192.0.2.10'],
 u'ansible_architecture': u'x86_64',
 ...
}
[192.0.2.10] TASK: install package (debug)> p task_vars['pkg_name']
u'bash'
[192.0.2.10] TASK: install package (debug)> p host
192.0.2.10
[192.0.2.10] TASK: install package (debug)> p result._result
{'_ansible_no_log': False,
 'changed': False,
 u'failed': True,
 ...
 u'msg': u"No package matching 'not_exist' is available"}

```

### **task.args[key] = value**

Update module's argument.

If you run a playbook like this:

```

- hosts: test
  strategy: debug
  gather_facts: yes
  vars:
    pkg_name: not_exist
  tasks:

```

(continues on next page)

()

```
- name: install package
  apt: name={{ pkg_name }}
```

Debugger is invoked due to wrong package name, so let's fix the module's args:

```
[192.0.2.10] TASK: install package (debug)> p task.args
{'u'name': u'{{ pkg_name }}'}
[192.0.2.10] TASK: install package (debug)> task.args['name'] = 'bash'
[192.0.2.10] TASK: install package (debug)> p task.args
{'u'name': 'bash'}
[192.0.2.10] TASK: install package (debug)> redo
```

Then the task runs again with new args.

### **task\_vars[key] = value**

Update task\_vars.

Let's use the same playbook above, but fix task\_vars instead of args:

```
[192.0.2.10] TASK: install package (debug)> p task_vars['pkg_name']
u'not_exist'
[192.0.2.10] TASK: install package (debug)> task_vars['pkg_name'] = 'bash'
[192.0.2.10] TASK: install package (debug)> p task_vars['pkg_name']
'bash'
[192.0.2.10] TASK: install package (debug)> redo
```

Then the task runs again with new task\_vars.

---

: In 2.5 this was updated from vars to task\_vars to not conflict with the vars() python function.

---

### **r(edo)**

Run the task again.

### **c(ontinue)**

Just continue.

### **q(uit)**

Quit from the debugger. The playbook execution is aborted.

### **Use with the free strategy**

Using the debugger on the `free` strategy will cause no further tasks to be queued or executed while the debugger is active. Additionally, using `redo` on a task to schedule it for re-execution may cause the rescheduled task to execute after subsequent tasks listed in your playbook.

:

**Working With Playbooks** An introduction to playbooks

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Delegation, Rolling Updates, and Local Actions

### Topics

- *Delegation, Rolling Updates, and Local Actions*
  - *Rolling Update Batch Size*
  - *Maximum Failure Percentage*
  - *Delegation*
  - *Delegated facts*
  - *Run Once*
  - *Local Playbooks*
  - *Interrupt execution on any error*

Being designed for multi-tier deployments since the beginning, Ansible is great at doing things on one host on behalf of another, or doing local steps with reference to some remote hosts.

This in particular is very applicable when setting up continuous deployment infrastructure or zero downtime rolling updates, where you might be talking with load balancers or monitoring systems.

Additional features allow for tuning the orders in which things complete, and assigning a batch window size for how many machines to process at once during a rolling update.

This section covers all of these features. For examples of these items in use, [please see the ansible-examples repository](#). There are quite a few examples of zero-downtime update procedures for different kinds of applications.

You should also consult the [Working With Modules](#) section, various modules like 'ec2\_elb', 'nagios', and 'bigip\_pool', and 'netScaler' dovetail neatly with the concepts mentioned here.

You'll also want to read up on [Roles](#), as the 'pre\_task' and 'post\_task' concepts are the places where you would typically call these modules.

Be aware that certain tasks are impossible to delegate, i.e. *include*, *add\_host*, *debug*, etc as they always execute on the controller.

### Rolling Update Batch Size

By default, Ansible will try to manage all of the machines referenced in a play in parallel. For a rolling update use case, you can define how many hosts Ansible should manage at a single time by using the `serial` keyword:

```
- name: test play
  hosts: webservers
  serial: 3
```

In the above example, if we had 100 hosts, 3 hosts in the group 'webservers' would complete the play completely before moving on to the next 3 hosts.

The `serial` keyword can also be specified as a percentage, which will be applied to the total number of hosts in a play, in order to determine the number of hosts per pass:

```
- name: test play
  hosts: webservers
  serial: "30%"
```

If the number of hosts does not divide equally into the number of passes, the final pass will contain the remainder.

As of Ansible 2.2, the batch sizes can be specified as a list, as follows:

```
- name: test play
  hosts: webservers
  serial:
    - 1
    - 5
    - 10
```

In the above example, the first batch would contain a single host, the next would contain 5 hosts, and (if there are any hosts left), every following batch would contain 10 hosts until all available hosts are used.

It is also possible to list multiple batch sizes as percentages:

```
- name: test play
  hosts: webservers
  serial:
    - "10%"
    - "20%"
    - "100%"
```

You can also mix and match the values:

```
- name: test play
  hosts: webservers
  serial:
    - 1
    - 5
    - "20%"
```

---

: No matter how small the percentage, the number of hosts per pass will always be 1 or greater.

---

### Maximum Failure Percentage

By default, Ansible will continue executing actions as long as there are hosts in the batch that have not yet failed. The batch size for a play is determined by the `serial` parameter. If `serial` is not set, then batch size is all the hosts specified in the `hosts:` field. In some situations, such as with the rolling updates described above, it may be desirable to abort the play when a certain threshold of failures have been reached. To achieve this, you can set a maximum failure percentage on a play as follows:

```
- hosts: webservers
  max_fail_percentage: 30
  serial: 10
```



In the above example, if more than 3 of the 10 servers in the group were to fail, the rest of the play would be aborted.

: The percentage set must be exceeded, not equaled. For example, if serial were set to 4 and you wanted the task to abort when 2 of the systems failed, the percentage should be set at 49 rather than 50.

## Delegation

This isn't actually rolling update specific but comes up frequently in those cases.

If you want to perform a task on one host with reference to other hosts, use the 'delegate\_to' keyword on a task. This is ideal for placing nodes in a load balanced pool, or removing them. It is also very useful for controlling outage windows. Be aware that it does not make sense to delegate all tasks, debug, add\_host, include, etc always get executed on the controller. Using this with the 'serial' keyword to control the number of hosts executing at one time is also a good idea:

```
---
- hosts: webserver
  serial: 5

  tasks:

  - name: take out of load balancer pool
    command: /usr/bin/take_out_of_pool {{ inventory_hostname }}
    delegate_to: 127.0.0.1

  - name: actual steps would go here
    yum:
      name: acme-web-stack
      state: latest

  - name: add back to load balancer pool
    command: /usr/bin/add_back_to_pool {{ inventory_hostname }}
    delegate_to: 127.0.0.1
```

These commands will run on 127.0.0.1, which is the machine running Ansible. There is also a shorthand syntax that you can use on a per-task basis: 'local\_action'. Here is the same playbook as above, but using the shorthand syntax for delegating to 127.0.0.1:

```
---
# ...

tasks:

- name: take out of load balancer pool
  local_action: command /usr/bin/take_out_of_pool {{ inventory_hostname }}

# ...

- name: add back to load balancer pool
  local_action: command /usr/bin/add_back_to_pool {{ inventory_hostname }}
```

A common pattern is to use a local action to call 'rsync' to recursively copy files to the managed servers. Here is an example:

```
---
# ...
tasks:

- name: recursively copy files from management server to target
  local_action: command rsync -a /path/to/files {{ inventory_hostname }}:/path/to/
  ↪target/
```

Note that you must have passphrase-less SSH keys or an ssh-agent configured for this to work, otherwise rsync will need to ask for a passphrase.

In case you have to specify more arguments you can use the following syntax:

```
---
# ...
tasks:

- name: Send summary mail
  local_action:
    module: mail
    subject: "Summary Mail"
    to: "{{ mail_recipient }}"
    body: "{{ mail_body }}"
  run_once: True
```

The *ansible\_host* variable (*ansible\_ssh\_host* in 1.x or specific to ssh/paramiko plugins) reflects the host a task is delegated to.

### Delegated facts

By default, any fact gathered by a delegated task are assigned to the *inventory\_hostname* (the current host) instead of the host which actually produced the facts (the delegated to host). The directive *delegate\_facts* may be set to *True* to assign the task's gathered facts to the delegated host instead of the current one.:

```
- hosts: app_servers
tasks:
  - name: gather facts from db servers
    setup:
      delegate_to: "{{ item }}"
      delegate_facts: True
    loop: "{{ groups['dbservers'] }}"
```

The above will gather facts for the machines in the dbservers group and assign the facts to those machines and not to app\_servers. This way you can lookup *hostvars['dbhost1']['default\_ipv4']['address']* even though dbservers were not part of the play, or left out by using *-limit*.

### Run Once

In some cases there may be a need to only run a task one time for a batch of hosts. This can be achieved by configuring "run\_once" on a task:

```
---
# ...
```

(continues on next page)

```
tasks:

    # ...

    - command: /opt/application/upgrade_db.py
      run_once: true

    # ...
```

This directive forces the task to attempt execution on the first host in the current batch and then applies all results and facts to all the hosts in the same batch.

This approach is similar to applying a conditional to a task such as:

```
- command: /opt/application/upgrade_db.py
  when: inventory_hostname == webservers[0]
```

But the results are applied to all the hosts.

Like most tasks, this can be optionally paired with "delegate\_to" to specify an individual host to execute on:

```
- command: /opt/application/upgrade_db.py
  run_once: true
  delegate_to: web01.example.org
```

As always with delegation, the action will be executed on the delegated host, but the information is still that of the original host in the task.

---

: When used together with "serial", tasks marked as "run\_once" will be run on one host in *each* serial batch. If it's crucial that the task is run only once regardless of "serial" mode, use `when: inventory_hostname == ansible_play_hosts[0]` construct.

---



---

: Any conditional (i.e *when:*) will use the variables of the 'first host' to decide if the task runs or not, no other hosts will be tested.

---

## Local Playbooks

It may be useful to use a playbook locally, rather than by connecting over SSH. This can be useful for assuring the configuration of a system by putting a playbook in a crontab. This may also be used to run a playbook inside an OS installer, such as an Anaconda kickstart.

To run an entire playbook locally, just set the "hosts:" line to "hosts: 127.0.0.1" and then run the playbook like so:

```
ansible-playbook playbook.yml --connection=local
```

Alternatively, a local connection can be used in a single playbook play, even if other plays in the playbook use the default remote connection type:

```
- hosts: 127.0.0.1
  connection: local
```

---

: If you set the connection to local and there is no `ansible_python_interpreter` set, modules will run under `/usr/bin/python` and not under `{{ ansible_playbook_python }}`. Be sure to set `ansible_python_interpreter: "{{ ansible_playbook_python }}"` in `host_vars/localhost.yml`, for example. You can avoid this issue by using `local_action` or `delegate_to: localhost` instead.

---

### Interrupt execution on any error

With the `"any_errors_fatal"` option, any failure on any host in a multi-host play will be treated as fatal and Ansible will exit immediately without waiting for the other hosts.

Sometimes `"serial"` execution is unsuitable; the number of hosts is unpredictable (because of dynamic inventory) and speed is crucial (simultaneous execution is required), but all tasks must be 100% successful to continue playbook execution.

For example, consider a service located in many datacenters with some load balancers to pass traffic from users to the service. There is a deploy playbook to upgrade service deb-packages. The playbook has the stages:

- disable traffic on load balancers (must be turned off simultaneously)
- gracefully stop the service
- upgrade software (this step includes tests and starting the service)
- enable traffic on the load balancers (which should be turned on simultaneously)

The service can't be stopped with "alive" load balancers; they must be disabled first. Because of this, the second stage can't be played if any server failed in the first stage.

For datacenter "A", the playbook can be written this way:

```
---
- hosts: load_balancers_dc_a
  any_errors_fatal: True
  tasks:
    - name: 'shutting down datacenter [ A ]'
      command: /usr/bin/disable-dc

- hosts: frontends_dc_a
  tasks:
    - name: 'stopping service'
      command: /usr/bin/stop-software
    - name: 'updating software'
      command: /usr/bin/upgrade-software

- hosts: load_balancers_dc_a
  tasks:
    - name: 'Starting datacenter [ A ]'
      command: /usr/bin/enable-dc
```

In this example Ansible will start the software upgrade on the front ends only if all of the load balancers are successfully disabled.

:

**Working With Playbooks** An introduction to playbooks

**Ansible Examples on GitHub** Many examples of full-stack deployments

**User Mailing List** Have a question? Stop by the google group!

[irc.freenode.net](#) #ansible IRC chat channel

## Setting the Environment (and Working With Proxies)

### 1.1 .

It is quite possible that you may need to get package updates through a proxy, or even get some package updates through a proxy and access other packages not through a proxy. Or maybe a script you might wish to call may also need certain environment variables set to run properly.

Ansible makes it easy for you to configure your environment by using the 'environment' keyword. Here is an example:

```
- hosts: all
  remote_user: root

  tasks:
    - apt: name=cobbler state=installed
      environment:
        http_proxy: http://proxy.example.com:8080
```

The environment can also be stored in a variable, and accessed like so:

```
- hosts: all
  remote_user: root

  # here we make a variable named "proxy_env" that is a dictionary
  vars:
    proxy_env:
      http_proxy: http://proxy.example.com:8080

  tasks:
    - apt: name=cobbler state=installed
      environment: "{{proxy_env}}"
```

You can also use it at a play level:

```
- hosts: testhost

  roles:
    - php
    - nginx

  environment:
    http_proxy: http://proxy.example.com:8080
```

While just proxy settings were shown above, any number of settings can be supplied. The most logical place to define an environment hash might be a group\_vars file, like so:

```
---
# file: group_vars/boston

ntp_server: ntp.bos.example.com
backup: bak.bos.example.com
proxy_env:
```

(continues on next page)

```
http_proxy: http://proxy.bos.example.com:8080
https_proxy: http://proxy.bos.example.com:8080
```

## Working With Language-Specific Version Managers

Some language-specific version managers (such as `rvm` and `nvm`) require environment variables be set while these tools are in use. When using these tools manually, they usually require sourcing some environment variables via a script or lines added to your shell configuration file. In Ansible, you can instead use the environment directive:

```
---
### A playbook demonstrating a common npm workflow:
# - Check for package.json in the application directory
# - If package.json exists:
#   * Run npm prune
#   * Run npm install

- hosts: application
  become: false

  vars:
    node_app_dir: /var/local/my_node_app

  environment:
    NVM_DIR: /var/local/nvm
    PATH: /var/local/nvm/versions/node/v4.2.1/bin:{{ ansible_env.PATH }}

  tasks:
    - name: check for package.json
      stat:
        path: '{{ node_app_dir }}/package.json'
      register: packagejson

    - name: npm prune
      command: npm prune
      args:
        chdir: '{{ node_app_dir }}'
      when: packagejson.stat.exists

    - name: npm install
      npm:
        path: '{{ node_app_dir }}'
      when: packagejson.stat.exists
```

You might also want to simply specify the environment for a single task:

```
---
- name: install ruby 2.3.1
  command: rvm install {{ rvm_ruby_version }}
  args:
    creates: '{{ rvm_root }}/versions/{{ rvm_ruby_version }}/bin/ruby'
  vars:
    rvm_root: /usr/local/rvm
    rvm_ruby_version: 2.3.1
  environment:
    CONFIGURE_OPTS: '--disable-install-doc'
```

(continues on next page)

()

```
RBENV_ROOT: '{{ rbenv_root }}'
PATH: '{{ rbenv_root }}/bin:{{ rbenv_root }}/shims:{{ rbenv_plugins }}/ruby-build/
↳bin:{{ ansible_env.PATH }}'
```

: environment: is not currently supported for Windows targets

:

**Working With Playbooks** An introduction to playbooks

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Error Handling In Playbooks

### Topics

- *Error Handling In Playbooks*
  - *Ignoring Failed Commands*
  - *Resetting Unreachable Hosts*
  - *Handlers and Failure*
  - *Controlling What Defines Failure*
  - *Overriding The Changed Result*
  - *Aborting the play*

Ansible normally has defaults that make sure to check the return codes of commands and modules and it fails fast – forcing an error to be dealt with unless you decide otherwise.

Sometimes a command that returns different than 0 isn't an error. Sometimes a command might not always need to report that it 'changed' the remote system. This section describes how to change the default behavior of Ansible for certain tasks so output and error handling behavior is as desired.

### Ignoring Failed Commands

Generally playbooks will stop executing any more steps on a host that has a task fail. Sometimes, though, you want to continue on. To do so, write a task that looks like this:

```
- name: this will not be counted as a failure
  command: /bin/false
  ignore_errors: yes
```

Note that the above system only governs the return value of failure of the particular task, so if you have an undefined variable used or a syntax error, it will still raise an error that users will need to address. Note that this will not prevent failures on connection or execution issues. This feature only works when the task must be able to run and return a value of 'failed'.

## Resetting Unreachable Hosts

### 2.2 .

Connection failures set hosts as 'UNREACHABLE', which will remove them from the list of active hosts for the run. To recover from these issues you can use *meta: clear\_host\_errors* to have all currently flagged hosts reactivated, so subsequent tasks can try to use them again.

## Handlers and Failure

When a task fails on a host, handlers which were previously notified will *not* be run on that host. This can lead to cases where an unrelated failure can leave a host in an unexpected state. For example, a task could update a configuration file and notify a handler to restart some service. If a task later on in the same play fails, the service will not be restarted despite the configuration change.

You can change this behavior with the `--force-handlers` command-line option, or by including `force_handlers: True` in a play, or `force_handlers = True` in `ansible.cfg`. When handlers are forced, they will run when notified even if a task fails on that host. (Note that certain errors could still prevent the handler from running, such as a host becoming unreachable.)

## Controlling What Defines Failure

Suppose the error code of a command is meaningless and to tell if there is a failure what really matters is the output of the command, for instance if the string "FAILED" is in the output.

Ansible provides a way to specify this behavior as follows:

```
- name: Fail task when the command error output prints FAILED
  command: /usr/bin/example-command -x -y -z
  register: command_result
  failed_when: "'FAILED' in command_result.stderr"
```

or based on the return code:

```
- name: Fail task when both files are identical
  raw: diff foo/file1 bar/file2
  register: diff_cmd
  failed_when: diff_cmd.rc == 0 or diff_cmd.rc >= 2
```

In previous version of Ansible, this can still be accomplished as follows:

```
- name: this command prints FAILED when it fails
  command: /usr/bin/example-command -x -y -z
  register: command_result
  ignore_errors: True

- name: fail the play if the previous command did not succeed
  fail:
    msg: "the command failed"
  when: "'FAILED' in command_result.stderr"
```



## Overriding The Changed Result

When a shell/command or other module runs it will typically report "changed" status based on whether it thinks it affected machine state.

Sometimes you will know, based on the return code or output that it did not make any changes, and wish to override the "changed" result such that it does not appear in report output or does not cause handlers to fire:

```
tasks:

- shell: /usr/bin/billybass --mode="take me to the river"
  register: bass_result
  changed_when: "bass_result.rc != 2"

# this will never report 'changed' status
- shell: wall 'beep'
  changed_when: False
```

## Aborting the play

Sometimes it's desirable to abort the entire play on failure, not just skip remaining tasks for a host.

The `any_errors_fatal` play option will mark all hosts as failed if any fails, causing an immediate abort:

```
- hosts: somehosts
  any_errors_fatal: true
  roles:
    - myrole
```

for finer-grained control `max_fail_percentage` can be used to abort the run after a given percentage of hosts has failed.

:

**Working With Playbooks** An introduction to playbooks

**Best Practices** Best practices in playbooks

**Conditionals** Conditional statements in playbooks

**Variables** All about variables

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Advanced Syntax

### Topics

- *Advanced Syntax*
  - *YAML tags and Python types*
  - \* *Unsafe or Raw Strings*

This page describes advanced YAML syntax that enables you to have more control over the data placed in YAML files used by Ansible.

### YAML tags and Python types

The documentation covered here is an extension of the documentation that can be found in the [PyYAML Documentation](#)

### Unsafe or Raw Strings

Ansible provides an internal data type for declaring variable values as "unsafe". This means that the data held within the variables value should be treated as unsafe preventing unsafe character substitution and information disclosure.

Jinja2 contains functionality for escaping, or telling Jinja2 to not template data by means of functionality such as `{% raw %} ... {% endraw %}`, however this uses a more comprehensive implementation to ensure that the value is never templated.

Using YAML tags, you can also mark a value as "unsafe" by using the `!unsafe` tag such as:

```
---
my_unsafe_variable: !unsafe 'this variable has {{ characters that should not be_
↳treated as a jinja2 template'
```

In a playbook, this may look like:

```
---
hosts: all
vars:
  my_unsafe_variable: !unsafe 'unsafe value'
tasks:
  ...
```

For complex variables such as hashes or arrays, `!unsafe` should be used on the individual elements such as:

```
---
my_unsafe_array:
  - !unsafe 'unsafe element'
  - 'safe element'

my_unsafe_hash:
  unsafe_key: !unsafe 'unsafe value'
```

:

[Variables](#) All about variables

[User Mailing List](#) Have a question? Stop by the google group!

[irc.freenode.net](#) #ansible IRC chat channel

### Working With Plugins

Plugins are pieces of code that augment Ansible's core functionality. Ansible uses a plugin architecture to enable a rich, flexible and expandable feature set.

Ansible ships with a number of handy plugins, and you can easily write your own.

This section covers the various types of plugins that are included with Ansible:

## Action Plugins

- *Enabling Action Plugins*
- *Using Action Plugins*
- *Plugin List*

Action plugins act in conjunction with *modules* to execute the actions required by playbook tasks. They usually execute automatically in the background doing prerequisite work before modules execute.

The 'normal' action plugin is used for modules that do not already have an action plugin.

## Enabling Action Plugins

You can enable a custom action plugin by either dropping it into the `action_plugins` directory adjacent to your play, inside a role, or by putting it in one of the action plugin directory sources configured in `ansible.cfg`.

## Using Action Plugins

Action plugin are executed by default when an associated module is used; no action is required.

## Plugin List

You can use `ansible-doc -t action -l` to see the list of available plugins. Use `ansible-doc -t action <plugin name>` to see specific documentation and examples.

:

*Cache Plugins* Ansible Cache plugins

*Callback Plugins* Ansible callback plugins

*Connection Plugins* Ansible connection plugins

*Inventory Plugins* Ansible inventory plugins

*Shell Plugins* Ansible Shell plugins

*Strategy Plugins* Ansible Strategy plugins

*Vars Plugins* Ansible Vars plugins

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

### Topics

- *Cache Plugins*
  - *Enabling Cache Plugins*

- *Using Cache Plugins*
- *Plugin List*

### Cache Plugins

Cache plugin implement a backend caching mechanism that allows Ansible to store gathered facts or inventory source data without the performance hit of retrieving them from source.

The default cache plugin is the memory plugin, which only caches the data for the current execution of Ansible. Other plugins with persistent storage are available to allow caching the data across runs.

### Enabling Cache Plugins

Only one cache plugin can be active at a time. You can enable a cache plugin in the Ansible configuration, either via environment variable:

```
export ANSIBLE_CACHE_PLUGIN=jsonfile
```

or in the `ansible.cfg` file:

```
[defaults]  
fact_caching=redis
```

You will also need to configure other settings specific to each plugin. Consult the individual plugin documentation or the Ansible configuration for more details.

A custom cache plugin is enabled by dropping it into a `cache_plugins` directory adjacent to your play, inside a role, or by putting it in one of the directory sources configured in `ansible.cfg`.

### Using Cache Plugins

Cache plugins are used automatically once they are enabled.

### Plugin List

You can use `ansible-doc -t cache -l` to see the list of available plugins. Use `ansible-doc -t cache <plugin name>` to see specific documentation and examples.

:

*Action Plugins* Ansible Action plugins

*Callback Plugins* Ansible callback plugins

*Connection Plugins* Ansible connection plugins

*Inventory Plugins* Ansible inventory plugins

*Shell Plugins* Ansible Shell plugins

*Strategy Plugins* Ansible Strategy plugins

*Vars Plugins* Ansible Vars plugins

**User Mailing List** Have a question? Stop by the google group!

**webchat.freenode.net** #ansible IRC chat channel

### Topics

- *Callback Plugins*
  - *Example Callback Plugins*
  - *Enabling Callback Plugins*
    - \* *Managing stdout*
    - \* *Managing AdHoc*
  - *Plugin List*

## Callback Plugins

Callback plugins enable adding new behaviors to Ansible when responding to events. By default, callback plugins control most of the output you see when running the command line programs, but can also be used to add additional output, integrate with other tools and marshal the events to a storage backend.

### Example Callback Plugins

The `_plays` callback is an example of how to record playbook events to a log file, and the `mail` callback sends email on playbook failures.

The `osx_say` callback responds with computer synthesized speech on macOS in relation to playbook events.

### Enabling Callback Plugins

You can activate a custom callback by either dropping it into a `callback_plugins` directory adjacent to your play, inside a role, or by putting it in one of the callback directory sources configured in `ansible.cfg`.

Plugins are loaded in alphanumeric order. For example, a plugin implemented in a file named `1_first.py` would run before a plugin file named `2_second.py`.

Most callbacks shipped with Ansible are disabled by default and need to be whitelisted in your `ansible.cfg` file in order to function. For example:

```
#callback_whitelist = timer, mail, profile_roles
```

### Managing stdout

You can only have one plugin be the main manager of your console output. If you want to replace the default, you should define `CALLBACK_TYPE = stdout` in the subclass and then configure the `stdout` plugin in `ansible.cfg`. For example:

```
stdout_callback = dense
```

or for my custom callback:

```
stdout_callback = mycallback
```

This only affects ansible-playbook by default.

### Managing AdHoc

The ansible ad hoc command specifically uses a different callback plugin for stdout, so there is an extra setting in ansible\_configuration\_settings you need to add to use the stdout callback defined above:

```
[defaults]
bin_ansible_callbacks=True
```

You can also set this as an environment variable:

```
export ANSIBLE_LOAD_CALLBACK_PLUGINS=1
```

### Plugin List

You can use `ansible-doc -t callback -l` to see the list of available plugins. Use `ansible-doc -t callback <plugin name>` to see specific documents and examples.

:

**Action Plugins** Ansible Action plugins

**Cache Plugins** Ansible cache plugins

**Connection Plugins** Ansible connection plugins

**Inventory Plugins** Ansible inventory plugins

**Shell Plugins** Ansible Shell plugins

**Strategy Plugins** Ansible Strategy plugins

**Vars Plugins** Ansible Vars plugins

**User Mailing List** Have a question? Stop by the google group!

**webchat.freenode.net** #ansible IRC chat channel

#### Topics

- *Connection Plugins*
  - *ssh Plugins*
  - *Enabling Connection Plugins*
  - *Using Connection Plugins*
  - *Plugin List*

## Connection Plugins

Connection plugins allow Ansible to connect to the target hosts so it can execute tasks on them. Ansible ships with many connection plugins, but only one can be used per host at a time.

By default, Ansible ships with several plugins. The most commonly used are the paramiko SSH, native ssh (just called ssh), and local connection types. All of these can be used in playbooks and with `/usr/bin/ansible` to decide how you want to talk to remote machines.

The basics of these connection types are covered in the *getting started* section.

## ssh Plugins

Because ssh is the default protocol used in system administration and the protocol most used in Ansible, ssh options are included in the command line tools. See `ansible-playbook` for more details.

## Enabling Connection Plugins

You can extend Ansible to support other transports (such as SNMP or message bus) by dropping a custom plugin into the `connection_plugins` directory.

## Using Connection Plugins

The transport can be changed via configuration, at the command line (`-c`, `--connection`), as a keyword in your play, or by setting a *variable*, most often in your inventory. For example, for Windows machines you might want to use the winrm plugin.

Most connection plugins can operate with a minimum configuration. By default they use the inventory hostname and defaults to find the target host.

Plugins are self-documenting. Each plugin should document its configuration options. The following are connection variables common to most connection plugins:

***ansible\_host*** The name of the host to connect to, if different from the *inventory* hostname.

***ansible\_port*** The ssh port number, for ssh and paramiko\_ssh it defaults to 22.

***ansible\_user*** The default user name to use for log in. Most plugins default to the 'current user running Ansible'.

Each plugin might also have a specific version of a variable that overrides the general version. For example, `ansible_ssh_host` for the ssh plugin.

## Plugin List

You can use `ansible-doc -t connection -l` to see the list of available plugins. Use `ansible-doc -t connection <plugin name>` to see detailed documentation and examples.

:

***Working with Playbooks*** An introduction to playbooks

***Callback Plugins*** Ansible callback plugins

***Filters*** Jinja2 filter plugins

***Tests*** Jinja2 test plugins

*Lookups* Jinja2 lookup plugins

*Vars Plugins* Ansible vars plugins

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

### Topics

- *Inventory Plugins*
  - *Enabling Inventory Plugins*
  - *Using Inventory Plugins*
  - *Plugin List*

## Inventory Plugins

Inventory plugins allow users to point at data sources to compile the inventory of hosts that Ansible uses to target tasks, either via the `-i /path/to/file` and/or `-i 'host1, host2'` command line parameters or from other configuration sources.

### Enabling Inventory Plugins

Most inventory plugins shipped with Ansible are disabled by default and need to be whitelisted in your `ansible.cfg` file in order to function. This is how the default whitelist looks in the config file that ships with Ansible:

```
[inventory]
enable_plugins = host_list, script, yaml, ini
```

This list also establishes the order in which each plugin tries to parse an inventory source. Any plugins left out of the list will not be considered, so you can 'optimize' your inventory loading by minimizing it to what you actually use. For example:

```
[inventory]
enable_plugins = advanced_host_list, constructed, yaml
```

### Using Inventory Plugins

The only requirement for using an inventory plugin after it is enabled is to provide an inventory source to parse. Ansible will try to use the list of enabled inventory plugins, in order, against each inventory source provided. Once an inventory plugin succeeds at parsing a source, the any remaining inventory plugins will be skipped for that source.

### Plugin List

You can use `ansible-doc -t inventory -l` to see the list of available plugins. Use `ansible-doc -t inventory <plugin name>` to see plugin-specific documentation and examples.

:

*About Playbooks* An introduction to playbooks



*Callback Plugins* Ansible callback plugins

*Connection Plugins* Ansible connection plugins

*Filters* Jinja2 filter plugins

*Tests* Jinja2 test plugins

*Lookups* Jinja2 lookup plugins

*Vars Plugins* Ansible vars plugins

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Topics

- *Lookup Plugins*
  - *Enabling Lookup Plugins*
  - *Using Lookup Plugins*
  - *query*
  - *Plugin List*

## Lookup Plugins

Lookup plugins allow Ansible to access data from outside sources. This can include reading the filesystem in addition to contacting external datastores and services. Like all templating, these plugins are evaluated on the Ansible control machine, not on the target/remote.

The data returned by a lookup plugin is made available using the standard templating system in Ansible, and are typically used to load variables or templates with information from those systems.

Lookups are an Ansible-specific extension to the Jinja2 templating language.

---

:

- Lookups are executed with a working directory relative to the role or play, as opposed to local tasks, which are executed relative the executed script.
  - Since Ansible version 1.9, you can pass `wantlist=True` to lookups to use in Jinja2 template "for" loops.
  - Lookup plugins are an advanced feature; to best leverage them you should have a good working knowledge of how to use Ansible plays.
- 

:

- Some lookups pass arguments to a shell. When using variables from a remote/untrusted source, use the `lquote` filter to ensure safe usage.

## Enabling Lookup Plugins

You can activate a custom lookup by either dropping it into a `lookup_plugins` directory adjacent to your play, inside a role, or by putting it in one of the lookup directory sources configured in `ansible.cfg`.

## Using Lookup Plugins

Lookup plugins can be used anywhere you can use templating in Ansible: in a play, in variables file, or in a Jinja2 template for the template module.

```
vars:
  file_contents: "{{lookup('file', 'path/to/file.txt')}}"
```

Lookups are an integral part of loops. Wherever you see `with_`, the part after the underscore is the name of a lookup. This is also the reason most lookups output lists and take lists as input; for example, `with_items` uses the items lookup:

```
tasks:
  - name: count to 3
    debug: msg={{item}}
    with_items: [1, 2, 3]
```

You can combine lookups with *Filters*, *Tests* and even each other to do some complex data generation and manipulation. For example:

```
tasks:
  - name: valid but useless and over complicated chained lookups and filters
    debug: msg="find the answer here:\n{{ lookup('url', 'http://google.com/search?q=
    ↪' + item|urlencode)|join(' ') }}"
    with_nested:
      - "{{lookup('consul_kv', 'bcs/' + lookup('file', '/the/question') + ',_
    ↪host=localhost, port=2000')|shuffle}}"
      - "{{lookup('sequence', 'end=42 start=2 step=2')|map('log', 4)|list)}}"
      - ['a', 'c', 'd', 'c']
```

### 2.6 .

You can now control how errors behave in all lookup plugins by setting `errors` to `ignore`, `warn`, or `strict`. The default setting is `strict`, which causes the task to fail. For example:

To ignore errors:

```
- name: file doesnt exist, but i dont care .. file plugin itself warns anyways ...
  debug: msg="{{ lookup('file', '/idontexist', errors='ignore') }}"

[WARNING]: Unable to find '/idontexist' in expected paths (use -vvvvv to see paths)

ok: [localhost] => {
  "msg": ""
}
```

To get a warning instead of a failure:

```
- name: file doesnt exist, let me know, but continue
  debug: msg="{{ lookup('file', '/idontexist', errors='warn') }}"
```

(continues on next page)

()

```
[WARNING]: Unable to find '/idontexist' in expected paths (use -vvvvv to see paths)

[WARNING]: An unhandled exception occurred while running the lookup plugin 'file'.
↳Error was a <class 'ansible.errors.AnsibleError'>, original message: could not
↳locate file in lookup: /idontexist

ok: [localhost] => {
    "msg": ""
}
```

**Fatal error (the default):**

```
- name: file doesnt exist, FAIL (this is the default)
  debug: msg="{{ lookup('file', '/idontexist', errors='strict') }}"

[WARNING]: Unable to find '/idontexist' in expected paths (use -vvvvv to see paths)

fatal: [localhost]: FAILED! => {"msg": "An unhandled exception occurred while running
↳the lookup plugin 'file'. Error was a <class 'ansible.errors.AnsibleError'>,
↳original message: could not locate file in lookup: /idontexist"}
```

**query****2.5 .**

In Ansible 2.5, a new jinja2 function called `query` was added for invoking lookup plugins. The difference between `lookup` and `query` is largely that `query` will always return a list. The default behavior of `lookup` is to return a string of comma separated values. `lookup` can be explicitly configured to return a list using `wantlist=True`.

This was done primarily to provide an easier and more consistent interface for interacting with the new `loop` keyword, while maintaining backwards compatibility with other uses of `lookup`.

The following examples are equivalent:

```
lookup('dict', dict_variable, wantlist=True)

query('dict', dict_variable)
```

As demonstrated above the behavior of `wantlist=True` is implicit when using `query`.

Additionally, `q` was introduced as a shortform of `query`:

```
q('dict', dict_variable)
```

**Plugin List**

You can use `ansible-doc -t lookup -l` to see the list of available plugins. Use `ansible-doc -t lookup <plugin name>` to see specific documents and examples.

:

*About Playbooks* An introduction to playbooks

*Inventory Plugins* Ansible inventory plugins

*Callback Plugins* Ansible callback plugins

*Filters* [Jinja2 filter plugins](#)

*Tests* [Jinja2 test plugins](#)

*Lookups* [Jinja2 lookup plugins](#)

**User Mailing List** [Have a question? Stop by the google group!](#)

**irc.freenode.net** [#ansible IRC chat channel](#)

### Topics

- *Shell Plugins*
  - *Enabling Shell Plugins*
  - *Using Shell Plugins*

## Shell Plugins

Shell plugins work to ensure that the basic commands Ansible runs are properly formatted to work with the target machine and allow the user to configure certain behaviors related to how Ansible executes tasks.

### Enabling Shell Plugins

You can add a custom shell plugin by dropping it into a `shell_plugins` directory adjacent to your play, inside a role, or by putting it in one of the shell plugin directory sources configured in `ansible.cfg`.

: You should not alter which plugin is used unless you have a setup in which the default `/bin/sh` is not a POSIX compatible shell or is not available for execution.

### Using Shell Plugins

In addition to the default configuration settings in `ansible_configuration_settings`, you can use the connection variable `ansible_shell_type` to select the plugin to use. In this case, you will also want to update the `ansible_shell_executable` to match.

You can further control the settings for each plugin via other configuration options detailed in the plugin themselves (linked below).

:

*Working With Playbooks* [An introduction to playbooks](#)

*Inventory Plugins* [Ansible inventory plugins](#)

*Callback Plugins* [Ansible callback plugins](#)

*Filters* [Jinja2 filter plugins](#)

*Tests* [Jinja2 test plugins](#)

*Lookups* [Jinja2 lookup plugins](#)

**User Mailing List** [Have a question? Stop by the google group!](#)

[irc.freenode.net](#) #ansible IRC chat channel

### Topics

- *Strategy Plugins*
  - *Enabling Strategy Plugins*
  - *Using Strategy Plugins*
  - *Plugin List*

## Strategy Plugins

Strategy plugins control the flow of play execution by handling task and host scheduling.

### Enabling Strategy Plugins

Strategy plugins shipped with Ansible are enabled by default. You can enable a custom strategy plugin by putting it in one of the lookup directory sources configured in `ansible.cfg`.

### Using Strategy Plugins

Only one strategy plugin can be used in a play, but you can use different ones for each play in a playbook or ansible run. The default is the linear plugin. You can change this default in Ansible configuration using an environment variable:

```
export ANSIBLE_STRATEGY=free
```

or in the `ansible.cfg` file:

```
[defaults]
strategy=linear
```

You can also specify the strategy plugin in the play via the strategy keyword in a play:

```
- hosts: all
  strategy: debug
  tasks:
    - copy: src=myhosts dest=/etc/hosts
      notify: restart_tomcat

    - package: name=tomcat state=present

  handlers:
    - name: restart_tomcat
      service: name=tomcat state=restarted
```

## Plugin List

You can use `ansible-doc -t strategy -l` to see the list of available plugins. Use `ansible-doc -t strategy <plugin name>` to see plugin-specific documentation and examples.

:

*About Playbooks* An introduction to playbooks

*Inventory Plugins* Ansible inventory plugins

*Callback Plugins* Ansible callback plugins

*Filters* Jinja2 filter plugins

*Tests* Jinja2 test plugins

*Lookups* Jinja2 lookup plugins

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

### Topics

- *Vars Plugins*
  - *Enabling Vars Plugins*
  - *Using Vars Plugins*
  - *Plugin Lists*

## Vars Plugins

Vars plugins inject additional variable data into Ansible runs that did not come from an inventory source, playbook, or command line. Playbook constructs like 'host\_vars' and 'group\_vars' work using vars plugins.

Vars plugins were partially implemented in Ansible 2.0 and rewritten to be fully implemented starting with Ansible 2.4.

The host\_group\_vars plugin shipped with Ansible enables reading variables from *Host Variables* and *Group Variables*.

## Enabling Vars Plugins

You can activate a custom vars plugin by either dropping it into a `vars_plugins` directory adjacent to your play, inside a role, or by putting it in one of the directory sources configured in `ansible.cfg`.

## Using Vars Plugins

Vars plugins are used automatically after they are enabled.

## Plugin Lists

You can use `ansible-doc -t vars -l` to see the list of available plugins. Use `ansible-doc -t vars <plugin name>` to see specific plugin-specific documentation and examples.

:

*Action Plugins* Ansible Action plugins

*Cache Plugins* Ansible Cache plugins

*Callback Plugins* Ansible callback plugins

*Connection Plugins* Ansible connection plugins

*Inventory Plugins* Ansible inventory plugins

*Shell Plugins* Ansible Shell plugins

*Strategy Plugins* Ansible Strategy plugins

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Plugin Filter Configuration

Ansible 2.5 adds the ability for a site administrator to blacklist modules that they do not want to be available to Ansible. This is configured via a yaml configuration file (by default, `/etc/ansible/plugin_filters.yml`). The format of the file is:

```
---
filter_version: '1.0'
module_blacklist:
  # Deprecated
  - docker
  # We only allow pip, not easy_install
  - easy_install
```

The file contains two fields:

- a version so that it will be possible to update the format while keeping backwards compatibility in the future. The present version should be the string, "1.0"
- a list of modules to blacklist. Any module listed here will not be found by Ansible when it searches for a module to invoke for a task.

:

*About Playbooks* An introduction to playbooks

**ansible\_configuration\_settings** Ansible configuration documentation and settings

*Working with Command Line Tools* Ansible tools, description and options

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Prompts

When running a playbook, you may wish to prompt the user for certain input, and can do so with the 'vars\_prompt' section.

A common use for this might be for asking for sensitive data that you do not want to record.

This has uses beyond security, for instance, you may use the same playbook for all software releases and would prompt for a particular release version in a push-script.

Here is a most basic example:

```
---
- hosts: all
  remote_user: root

  vars:
    from: "camelot"

  vars_prompt:
    - name: "name"
      prompt: "what is your name?"
    - name: "quest"
      prompt: "what is your quest?"
    - name: "favcolor"
      prompt: "what is your favorite color?"
```

---

: Prompts for individual `vars_prompt` variables will be skipped for any variable that is already defined through the command line `--extra-vars` option, or when running from a non-interactive session (such as cron or Ansible Tower). See [Passing Variables On The Command Line](#) in the `/Variables/` chapter.

---

If you have a variable that changes infrequently, it might make sense to provide a default value that can be overridden. This can be accomplished using the `default` argument:

```
vars_prompt:

- name: "release_version"
  prompt: "Product release version"
  default: "1.0"
```

An alternative form of `vars_prompt` allows for hiding input from the user, and may later support some other options, but otherwise works equivalently:

```
vars_prompt:

- name: "some_password"
  prompt: "Enter password"
  private: yes

- name: "release_version"
  prompt: "Product release version"
  private: no
```

If `Passlib` is installed, `vars_prompt` can also encrypt the entered value so you can use it, for instance, with the `user` module to define a password:

```
vars_prompt:

- name: "my_password2"
  prompt: "Enter password2"
  private: yes
  encrypt: "sha512_crypt"
  confirm: yes
  salt_size: 7
```

You can use any crypt scheme supported by 'Passlib':

- `des_crypt` - DES Crypt



- *bsdi\_crypt* - BSDi Crypt
- *bigcrypt* - BigCrypt
- *crypt16* - Crypt16
- *md5\_crypt* - MD5 Crypt
- *bcrypt* - BCrypt
- *sha1\_crypt* - SHA-1 Crypt
- *sun\_md5\_crypt* - Sun MD5 Crypt
- *sha256\_crypt* - SHA-256 Crypt
- *sha512\_crypt* - SHA-512 Crypt
- *apr\_md5\_crypt* - Apache's MD5-Crypt variant
- *phpass* - PHPass' Portable Hash
- *pbkdf2\_digest* - Generic PBKDF2 Hashes
- *cta\_pbkdf2\_sha1* - Cryptacular's PBKDF2 hash
- *dlitz\_pbkdf2\_sha1* - Dwayne Litzenberger's PBKDF2 hash
- *scram* - SCRAM Hash
- *bsd\_nthash* - FreeBSD's MCF-compatible nthash encoding

However, the only parameters accepted are 'salt' or 'salt\_size'. You can use your own salt using 'salt', or have one generated automatically using 'salt\_size'. If nothing is specified, a salt of size 8 will be generated.

:

**Working With Playbooks** An introduction to playbooks

**Conditionals** Conditional statements in playbooks

**Variables** All about variables

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Tags

If you have a large playbook it may become useful to be able to run a specific part of the configuration without running the whole playbook.

Both plays and tasks support a "tags:" attribute for this reason. You can **ONLY** filter tasks based on tags from the command line with `--tags` or `--skip-tags`. Adding "tags:" in any part of a play (including roles) adds those tags to the contained tasks.

Example:

```
tasks:
  - yum:
      name: "{{ item }}"
      state: installed
    loop:
      - httpd
```

(continues on next page)

```
- memcached
tags:
  - packages

- template:
  src: templates/src.j2
  dest: /etc/foo.conf
tags:
  - configuration
```

If you wanted to just run the "configuration" and "packages" part of a very long playbook, you could do this:

```
ansible-playbook example.yml --tags "configuration,packages"
```

On the other hand, if you want to run a playbook *without* certain tasks, you could do this:

```
ansible-playbook example.yml --skip-tags "notification"
```

## Tag Reuse

You can apply the same tag name to more than one task, in the same file or included files. This will run all tasks with that tag.

Example:

```
---
# file: roles/common/tasks/main.yml

- name: be sure ntp is installed
  yum:
    name: ntp
    state: installed
  tags: ntp

- name: be sure ntp is configured
  template:
    src: ntp.conf.j2
    dest: /etc/ntp.conf
  notify:
    - restart ntpd
  tags: ntp

- name: be sure ntpd is running and enabled
  service:
    name: ntpd
    state: started
    enabled: yes
  tags: ntp
```

## Tag Inheritance

You can apply tags to more than tasks, but they **ONLY** affect the tasks themselves. Applying tags anywhere else is just a convenience so you don't have to write it on every task:

```
- hosts: all
  tags:
    - bar
  tasks:
    ...

- hosts: all
  tags: ['foo']
  tasks:
    ...
```

You may also apply tags to the tasks imported by roles:

```
roles:
  - role: webserver
    vars:
      port: 5000
      tags: [ 'web', 'foo' ]
```

And import statements:

```
- import_tasks: foo.yml
  tags: [web,foo]
```

All of these apply the specified tags to EACH task inside the play, imported file, or role, so that these tasks can be selectively run when the playbook is invoked with the corresponding tags.

There is no way to 'import only these tags'; you probably want to split into smaller roles/includes if you find yourself looking for such a feature.

The above information does not apply to *include\_tasks* or other dynamic includes, as the attributes applied to an include, only affect the include itself.

Tags are inherited *down* the dependency chain. In order for tags to be applied to a role and all its dependencies, the tag should be applied to the role, not to all the tasks within a role.

You can see which tags are applied to tasks by running `ansible-playbook` with the `--list-tasks` option. You can display all tags using the `--list-tags` option.

## Special Tags

There is a special `always` tag that will always run a task, unless specifically skipped (`--skip-tags always`)

Example:

```
tasks:

  - debug:
      msg: "Always runs"
      tags:
        - always

  - debug:
      msg: "runs when you use tag1"
      tags:
        - tag1
```

2.5 .

Another special tag is `never`, which will prevent a task from running unless a tag is specifically requested.

Example:

```
tasks:
- debug: msg='{{ showmevar }}'
  tags: [ 'never', 'debug' ]
```

In this example, the task will only run when the `debug` or `never` tag is explicitly requested.

There are another 3 special keywords for tags: `tagged`, `untagged` and `all`, which run only tagged, only untagged and all tasks respectively.

By default, Ansible runs as if `--tags all` had been specified.

:

**Working With Playbooks** An introduction to playbooks

**Roles** Playbook organization by roles

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Using Vault in playbooks

### Topics

- *Using Vault in playbooks*
  - *Running a Playbook With Vault*
  - *Single Encrypted Variable*
  - *Using encrypt\_string*

The "Vault" is a feature of Ansible that allows you to keep sensitive data such as passwords or keys in encrypted files, rather than as plaintext in playbooks or roles. These vault files can then be distributed or placed in source control.

To enable this feature, a command line tool, `ansible-vault` is used to edit files, and a command line flag `--ask-vault-pass` or `--vault-password-file` is used. You can also modify your `ansible.cfg` file to specify the location of a password file or configure Ansible to always prompt for the password. These options require no command line flag usage.

For best practices advice, refer to *Variables and Vaults*.

## Running a Playbook With Vault

To run a playbook that contains vault-encrypted data files, you must pass one of two flags. To specify the vault-password interactively:

```
ansible-playbook site.yml --ask-vault-pass
```

This prompt will then be used to decrypt (in memory only) any vault encrypted files that are accessed. Currently this requires that all files be encrypted with the same password.

Alternatively, passwords can be specified with a file or a script (the script version will require Ansible 1.7 or later). When using this flag, ensure permissions on the file are such that no one else can access your key and do not add your key to source control:

```
ansible-playbook site.yml --vault-password-file ~/.vault_pass.txt
ansible-playbook site.yml --vault-password-file ~/.vault_pass.py
```

The password should be a string stored as a single line in the file.

---

: You can also set `ANSIBLE_VAULT_PASSWORD_FILE` environment variable, e.g. `ANSIBLE_VAULT_PASSWORD_FILE=~/.vault_pass.txt` and Ansible will automatically search for the password in that file.

---

If you are using a script instead of a flat file, ensure that it is marked as executable, and that the password is printed to standard output. If your script needs to prompt for data, prompts can be sent to standard error.

This is something you may wish to do if using Ansible from a continuous integration system like Jenkins.

The `--vault-password-file` option can also be used with the `ansible-pull` command if you wish, though this would require distributing the keys to your nodes, so understand the implications – vault is more intended for push mode.

## Single Encrypted Variable

As of version 2.3, Ansible can now use a vaulted variable that lives in an otherwise 'clear text' YAML file:

```
notsecret: myvalue
mysecret: !vault |
    $ANSIBLE_VAULT;1.1;AES256
    ↵
    ↪66386439653236336462626566653063336164663966303231363934653561363964363833313662
    ↵
    ↪6431626536303530376336343832656537303632313433360a626438346336353331386135323734
    ↵
    ↪62656361653630373231613662633962316233633936396165386439616533353965373339616234
    ↵
    ↪3430613539666330390a313736323265656432366236633330313963326365653937323833366536
    34623731376664623134383463316265643436343438623266623965636363326136
other_plain_text: othervalue
```

To create a vaulted variable, use the `ansible-vault encrypt_string` command. See *Using encrypt\_string* for details.

This vaulted variable will be decrypted with the supplied vault secret and used as a normal variable. The `ansible-vault` command line supports stdin and stdout for encrypting data on the fly, which can be used from your favorite editor to create these vaulted variables; you just have to be sure to add the `!vault` tag so both Ansible and YAML are aware of the need to decrypt. The `|` is also required, as vault encryption results in a multi-line string.

## Using encrypt\_string

This command will output a string in the above format ready to be included in a YAML file. The string to encrypt can be provided via stdin, command line arguments, or via an interactive prompt.

See *Use encrypt\_string to create encrypted variables to embed in yaml*.

### Start and Step

This shows a few alternative ways to run playbooks. These modes are very useful for testing new plays or debugging.

#### Start-at-task

If you want to start executing your playbook at a particular task, you can do so with the `--start-at-task` option:

```
ansible-playbook playbook.yml --start-at-task="install packages"
```

The above will start executing your playbook at a task named "install packages".

#### Step

Playbooks can also be executed interactively with `--step`:

```
ansible-playbook playbook.yml --step
```

This will cause ansible to stop on each task, and ask if it should execute that task. Say you had a task called "configure ssh", the playbook run will stop and ask:

```
Perform task: configure ssh (y/n/c):
```

Answering "y" will execute the task, answering "n" will skip the task, and answering "c" will continue executing all the remaining tasks without asking.

#### Module defaults

If you find yourself calling the same module repeatedly with the same arguments, it can be useful to define default arguments for that particular module using the `module_defaults` attribute.

Here is a basic example:

```
- hosts: localhost
  module_defaults:
    file:
      owner: root
      group: root
      mode: 0755
  tasks:
    - file:
        state: touch
        path: /tmp/file1
    - file:
        state: touch
        path: /tmp/file2
    - file:
        state: touch
        path: /tmp/file3
```

The `module_defaults` attribute can be used at the play, block, and task level. Any module arguments explicitly specified in a task will override any established default for that module argument:

```
- block:
  - debug:
      msg: "a different message"
  module_defaults:
    debug:
      msg: "a default message"
```

It's also possible to remove any previously established defaults for a module by specifying an empty dict:

```
- file:
  state: touch
  path: /tmp/file1
  module_defaults:
    file: {}
```

: Any module defaults set at the play level (and block/task level when using `include_role` or `import_role`) will apply to any roles used, which may cause unexpected behavior in the role.

Here are some more realistic use cases for this feature.

Interacting with an API that requires auth:

```
- hosts: localhost
  module_defaults:
    uri:
      force_basic_auth: true
      user: some_user
      password: some_password
  tasks:
    - uri:
        url: http://some.api.host/v1/whatever1
    - uri:
        url: http://some.api.host/v1/whatever2
    - uri:
        url: http://some.api.host/v1/whatever3
```

Setting a default AWS region for specific EC2-related modules:

```
- hosts: localhost
  vars:
    my_region: us-west-2
  module_defaults:
    ec2:
      region: '{{ my_region }}'
    ec2_instance_facts:
      region: '{{ my_region }}'
    ec2_vpc_net_facts:
      region: '{{ my_region }}'
```

## Strategies

Strategies are a way to control play execution. By default, plays run with a `linear` strategy, in which all hosts will run each task before any host starts the next task, using the number of forks (default 5) to parallelize.

The `serial` directive can 'batch' this behaviour to a subset of the hosts, which then run to completion of the play before the next 'batch' starts.

A second strategy ships with Ansible - `free` - which allows each host to run until the end of the play as fast as it can.:

```
- hosts: all
  strategy: free
  tasks:
  ...
```

### Strategy Plugins

The strategies are implemented as plugins. In the future, new execution strategies can be added, either locally by users or to Ansible itself by a code contribution.

One example is debug strategy. See [Playbook Debugger](#) for details.

:

[Working With Playbooks](#) An introduction to playbooks

[Roles](#) Playbook organization by roles

[User Mailing List](#) Have a question? Stop by the google group!

[irc.freenode.net](#) #ansible IRC chat channel

### Best Practices

Here are some tips for making the most of Ansible and Ansible playbooks.

You can find some example playbooks illustrating these best practices in our [ansible-examples repository](#). (NOTE: These may not use all of the features in the latest release, but are still an excellent reference!).

#### Topics

- *Best Practices*
  - *Content Organization*
    - \* *Directory Layout*
    - \* *Alternative Directory Layout*
    - \* *Use Dynamic Inventory With Clouds*
    - \* *How to Differentiate Staging vs Production*
    - \* *Group And Host Variables*
    - \* *Top Level Playbooks Are Separated By Role*
    - \* *Task And Handler Organization For A Role*
    - \* *What This Organization Enables (Examples)*
    - \* *Deployment vs Configuration Organization*
  - *Staging vs Production*



- *Rolling Updates*
- *Always Mention The State*
- *Group By Roles*
- *Operating System and Distribution Variance*
- *Bundling Ansible Modules With Playbooks*
- *Whitespace and Comments*
- *Always Name Tasks*
- *Keep It Simple*
- *Version Control*
- *Variables and Vaults*

## Content Organization

The following section shows one of many possible ways to organize playbook content.

Your usage of Ansible should fit your needs, however, not ours, so feel free to modify this approach and organize as you see fit.

One crucial way to organize your playbook content is Ansible's "roles" organization feature, which is documented as part of the main playbooks page. You should take the time to read and understand the roles documentation which is available here: [Roles](#).

## Directory Layout

The top level of the directory would contain files and directories like so:

```
production          # inventory file for production servers
staging             # inventory file for staging environment

group_vars/
  group1.yml         # here we assign variables to particular groups
  group2.yml
host_vars/
  hostname1.yml      # here we assign variables to particular systems
  hostname2.yml

library/            # if any custom modules, put them here (optional)
module_utils/       # if any custom module_utils to support modules, put them
↳ here (optional)
filter_plugins/     # if any custom filter plugins, put them here (optional)

site.yml            # master playbook
webservers.yml      # playbook for webserver tier
dbservers.yml       # playbook for dbserver tier

roles/
  common/           # this hierarchy represents a "role"
    tasks/         #
      main.yml      # <-- tasks file can include smaller files if warranted
```

(continues on next page)

```

handlers/      #
  main.yml     # <-- handlers file
templates/     # <-- files for use with the template resource
  ntp.conf.j2  # <----- templates end in .j2
files/         #
  bar.txt      # <-- files for use with the copy resource
  foo.sh       # <-- script files for use with the script resource
vars/         #
  main.yml     # <-- variables associated with this role
defaults/     #
  main.yml     # <-- default lower priority variables for this role
meta/         #
  main.yml     # <-- role dependencies
library/      # roles can also include custom modules
module_utils/ # roles can also include custom module_utils
lookup_plugins/ # or other types of plugins, like lookup in this case

webtier/      # same kind of structure as "common" was above, done for_
→the webtier role
monitoring/   # ""
fooapp/       # ""

```

## Alternative Directory Layout

Alternatively you can put each inventory file with its group\_vars/host\_vars in a separate directory. This is particularly useful if your group\_vars/host\_vars don't have that much in common in different environments. The layout could look something like this:

```

inventories/
  production/
    hosts          # inventory file for production servers
    group_vars/
      group1.yml   # here we assign variables to particular groups
      group2.yml
    host_vars/
      hostname1.yml # here we assign variables to particular systems
      hostname2.yml

  staging/
    hosts          # inventory file for staging environment
    group_vars/
      group1.yml   # here we assign variables to particular groups
      group2.yml
    host_vars/
      stagehost1.yml # here we assign variables to particular systems
      stagehost2.yml

library/
module_utils/
filter_plugins/

site.yml
webservers.yml
dbservers.yml

```

(continues on next page)

()

```
roles/
  common/
  webtier/
  monitoring/
  fooapp/
```

This layout gives you more flexibility for larger environments, as well as a total separation of inventory variables between different environments. The downside is that it is harder to maintain, because there are more files.

## Use Dynamic Inventory With Clouds

If you are using a cloud provider, you should not be managing your inventory in a static file. See [Working With Dynamic Inventory](#).

This does not just apply to clouds – If you have another system maintaining a canonical list of systems in your infrastructure, usage of dynamic inventory is a great idea in general.

## How to Differentiate Staging vs Production

If managing static inventory, it is frequently asked how to differentiate different types of environments. The following example shows a good way to do this. Similar methods of grouping could be adapted to dynamic inventory (for instance, consider applying the AWS tag "environment:production", and you'll get a group of systems automatically discovered named "ec2\_tag\_environment\_production").

Let's show a static inventory example though. Below, the *production* file contains the inventory of all of your production hosts.

It is suggested that you define groups based on purpose of the host (roles) and also geography or datacenter location (if applicable):

```
# file: production

[atlanta-webservers]
www-atl-1.example.com
www-atl-2.example.com

[boston-webservers]
www-bos-1.example.com
www-bos-2.example.com

[atlanta-dbbservers]
db-atl-1.example.com
db-atl-2.example.com

[boston-dbbservers]
db-bos-1.example.com

# webbservers in all geos
[webbservers:children]
atlanta-webservers
boston-webservers

# dbbservers in all geos
[dbbservers:children]
```

(continues on next page)

```
atlanta-dbserver
boston-dbserver

# everything in the atlanta geo
[atlanta:children]
atlanta-webserver
atlanta-dbserver

# everything in the boston geo
[boston:children]
boston-webserver
boston-dbserver
```

## Group And Host Variables

This section extends on the previous example.

Groups are nice for organization, but that's not all groups are good for. You can also assign variables to them! For instance, atlanta has its own NTP servers, so when setting up ntp.conf, we should use them. Let's set those now:

```
---
# file: group_vars/atlanta
ntp: ntp-atlanta.example.com
backup: backup-atlanta.example.com
```

Variables aren't just for geographic information either! Maybe the webserver have some configuration that doesn't make sense for the database servers:

```
---
# file: group_vars/webserver
apacheMaxRequestsPerChild: 3000
apacheMaxClients: 900
```

If we had any default values, or values that were universally true, we would put them in a file called group\_vars/all:

```
---
# file: group_vars/all
ntp: ntp-boston.example.com
backup: backup-boston.example.com
```

We can define specific hardware variance in systems in a host\_vars file, but avoid doing this unless you need to:

```
---
# file: host_vars/db-bos-1.example.com
foo_agent_port: 86
bar_agent_port: 99
```

Again, if we are using dynamic inventory sources, many dynamic groups are automatically created. So a tag like "class:webserver" would load in variables from the file "group\_vars/ec2\_tag\_class\_webserver" automatically.

## Top Level Playbooks Are Separated By Role

In site.yml, we import a playbook that defines our entire infrastructure. This is a very short example, because it's just importing some other playbooks:

```
---
# file: site.yml
- import_playbook: webservers.yml
- import_playbook: dbservers.yml
```

In a file like `webservers.yml` (also at the top level), we map the configuration of the `webservers` group to the roles performed by the `webservers` group:

```
---
# file: webservers.yml
- hosts: webservers
  roles:
    - common
    - webtier
```

The idea here is that we can choose to configure our whole infrastructure by "running" `site.yml` or we could just choose to run a subset by running `webservers.yml`. This is analogous to the `--limit` parameter to `ansible` but a little more explicit:

```
ansible-playbook site.yml --limit webservers
ansible-playbook webservers.yml
```

## Task And Handler Organization For A Role

Below is an example tasks file that explains how a role works. Our common role here just sets up NTP, but it could do more if we wanted:

```
---
# file: roles/common/tasks/main.yml

- name: be sure ntp is installed
  yum:
    name: ntp
    state: installed
  tags: ntp

- name: be sure ntp is configured
  template:
    src: ntp.conf.j2
    dest: /etc/ntp.conf
  notify:
    - restart ntpd
  tags: ntp

- name: be sure ntpd is running and enabled
  service:
    name: ntpd
    state: started
    enabled: yes
  tags: ntp
```

Here is an example handlers file. As a review, handlers are only fired when certain tasks report changes, and are run at the end of each play:

```
---
# file: roles/common/handlers/main.yml
- name: restart ntpd
  service:
    name: ntpd
    state: restarted
```

See [Roles](#) for more information.

### What This Organization Enables (Examples)

Above we've shared our basic organizational structure.

Now what sort of use cases does this layout enable? Lots! If I want to reconfigure my whole infrastructure, it's just:

```
ansible-playbook -i production site.yml
```

To reconfigure NTP on everything:

```
ansible-playbook -i production site.yml --tags ntp
```

To reconfigure just my webservers:

```
ansible-playbook -i production webservers.yml
```

For just my webservers in Boston:

```
ansible-playbook -i production webservers.yml --limit boston
```

For just the first 10, and then the next 10:

```
ansible-playbook -i production webservers.yml --limit boston[0:9]
ansible-playbook -i production webservers.yml --limit boston[10:19]
```

And of course just basic ad-hoc stuff is also possible:

```
ansible boston -i production -m ping
ansible boston -i production -m command -a '/sbin/reboot'
```

And there are some useful commands to know:

```
# confirm what task names would be run if I ran this command and said "just ntp tasks"
ansible-playbook -i production webservers.yml --tags ntp --list-tasks

# confirm what hostnames might be communicated with if I said "limit to boston"
ansible-playbook -i production webservers.yml --limit boston --list-hosts
```

### Deployment vs Configuration Organization

The above setup models a typical configuration topology. When doing multi-tier deployments, there are going to be some additional playbooks that hop between tiers to roll out an application. In this case, 'site.yml' may be augmented by playbooks like 'deploy\_example.com.yml' but the general concepts can still apply.

Consider "playbooks" as a sports metaphor – you don't have to just have one set of plays to use against your infrastructure all the time – you can have situational plays that you use at different times and for different purposes.

Ansible allows you to deploy and configure using the same tool, so you would likely reuse groups and just keep the OS configuration in separate playbooks from the app deployment.

## Staging vs Production

As also mentioned above, a good way to keep your staging (or testing) and production environments separate is to use a separate inventory file for staging and production. This way you pick with `-i` what you are targeting. Keeping them all in one file can lead to surprises!

Testing things in a staging environment before trying in production is always a great idea. Your environments need not be the same size and you can use group variables to control the differences between those environments.

## Rolling Updates

Understand the `'serial'` keyword. If updating a webserver farm you really want to use it to control how many machines you are updating at once in the batch.

See *Delegation, Rolling Updates, and Local Actions*.

## Always Mention The State

The `'state'` parameter is optional to a lot of modules. Whether `'state=present'` or `'state=absent'`, it's always best to leave that parameter in your playbooks to make it clear, especially as some modules support additional states.

## Group By Roles

We're somewhat repeating ourselves with this tip, but it's worth repeating. A system can be in multiple groups. See *Working with Inventory* and *Working with Patterns*. Having groups named after things like *webserver*s and *dbserver*s is repeated in the examples because it's a very powerful concept.

This allows playbooks to target machines based on role, as well as to assign role specific variables using the group variable system.

See *Roles*.

## Operating System and Distribution Variance

When dealing with a parameter that is different between two different operating systems, a great way to handle this is by using the `group_by` module.

This makes a dynamic group of hosts matching certain criteria, even if that group is not defined in the inventory file:

```
---
# talk to all hosts just so we can learn about them
- hosts: all
  tasks:
    - group_by:
        key: os_{{ ansible_distribution }}

# now just on the CentOS hosts...
```

(continues on next page)

()

```
- hosts: os_CentOS
  gather_facts: False
  tasks:
    - # tasks that only happen on CentOS go here
```

This will throw all systems into a dynamic group based on the operating system name.

If group-specific settings are needed, this can also be done. For example:

```
---
# file: group_vars/all
asdf: 10

---
# file: group_vars/os_CentOS
asdf: 42
```

In the above example, CentOS machines get the value of '42' for asdf, but other machines get '10'. This can be used not only to set variables, but also to apply certain roles to only certain systems.

Alternatively, if only variables are needed:

```
- hosts: all
  tasks:
    - include_vars: "os_{{ ansible_distribution }}.yaml"
    - debug:
        var: asdf
```

This will pull in variables based on the OS name.

## Bundling Ansible Modules With Playbooks

If a playbook has a `./library` directory relative to its YAML file, this directory can be used to add ansible modules that will automatically be in the ansible module path. This is a great way to keep modules that go with a playbook together. This is shown in the directory structure example at the start of this section.

## Whitespace and Comments

Generous use of whitespace to break things up, and use of comments (which start with '#'), is encouraged.

## Always Name Tasks

It is possible to leave off the 'name' for a given task, though it is recommended to provide a description about why something is being done instead. This name is shown when the playbook is run.

## Keep It Simple

When you can do something simply, do something simply. Do not reach to use every feature of Ansible together, all at once. Use what works for you. For example, you will probably not need `vars`, `vars_files`, `vars_prompt` and `--extra-vars` all at once, while also using an external inventory file.

If something feels complicated, it probably is, and may be a good opportunity to simplify things.



## Version Control

Use version control. Keep your playbooks and inventory file in git (or another version control system), and commit when you make changes to them. This way you have an audit trail describing when and why you changed the rules that are automating your infrastructure.

## Variables and Vaults

For general maintenance, it is often easier to use `grep`, or similar tools, to find variables in your Ansible setup. Since vaults obscure these variables, it is best to work with a layer of indirection. When running a playbook, Ansible finds the variables in the unencrypted file and all sensitive variables come from the encrypted file.

A best practice approach for this is to start with a `group_vars/` subdirectory named after the group. Inside of this subdirectory, create two files named `vars` and `vault`. Inside of the `vars` file, define all of the variables needed, including any sensitive ones. Next, copy all of the sensitive variables over to the `vault` file and prefix these variables with `vault_`. You should adjust the variables in the `vars` file to point to the matching `vault_` variables using `jinja2` syntax, and ensure that the `vault` file is vault encrypted.

This best practice has no limit on the amount of variable and vault files or their names.

:

**YAML Syntax** Learn about YAML syntax

**Working With Playbooks** Review the basic playbook features

**all\_modules** Learn about available modules

**Developing Modules** Learn how to extend Ansible by writing your own modules

**Working with Patterns** Learn about how to select hosts

**GitHub examples directory** Complete playbook files from the github project source

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

### 1.4.8 Ansible Vault

#### Topics

- *Ansible Vault*
  - *What Can Be Encrypted With Vault*
  - *Creating Encrypted Files*
  - *Editing Encrypted Files*
  - *Rekeying Encrypted Files*
  - *Encrypting Unencrypted Files*
  - *Decrypting Encrypted Files*
  - *Viewing Encrypted Files*
  - *Use `encrypt_string` to create encrypted variables to embed in `yaml`*
  - *Vault Ids and Multiple Vault Passwords*

- *Providing Vault Passwords*
  - \* *Multiple vault passwords*
- *Speeding Up Vault Operations*
- *Vault Format*
- *Vault Payload Format 1.1*

Ansible Vault is a feature of ansible that allows you to keep sensitive data such as passwords or keys in encrypted files, rather than as plaintext in playbooks or roles. These vault files can then be distributed or placed in source control.

To enable this feature, a command line tool - `ansible-vault` - is used to edit files, and a command line flag (`--ask-vault-pass` or `--vault-password-file`) is used. Alternately, you may specify the location of a password file or command Ansible to always prompt for the password in your `ansible.cfg` file. These options require no command line flag usage.

For best practices advice, refer to [Variables and Vaults](#).

### What Can Be Encrypted With Vault

Ansible Vault can encrypt any structured data file used by Ansible. This can include "group\_vars/" or "host\_vars/" inventory variables, variables loaded by "include\_vars" or "vars\_files", or variable files passed on the ansible-playbook command line with `-e @file.yml` or `-e @file.json`. Role variables and defaults are also included.

Ansible tasks, handlers, and so on are also data so these can be encrypted with vault as well. To hide the names of variables that you're using, you can encrypt the task files in their entirety.

Ansible Vault can also encrypt arbitrary files, even binary files. If a vault-encrypted file is given as the `src` argument to the copy, template, unarchive, script or assemble modules, the file will be placed at the destination on the target host decrypted (assuming a valid vault password is supplied when running the play).

As of version 2.3, Ansible supports encrypting single values inside a YAML file, using the `!vault` tag to let YAML and Ansible know it uses special processing. This feature is covered in more details below.

### Creating Encrypted Files

To create a new encrypted data file, run the following command:

```
ansible-vault create foo.yml
```

First you will be prompted for a password. The password used with vault currently must be the same for all files you wish to use together at the same time.

After providing a password, the tool will launch whatever editor you have defined with `$EDITOR`, and defaults to vi (before 2.1 the default was vim). Once you are done with the editor session, the file will be saved as encrypted data.

The default cipher is AES (which is shared-secret based).

### Editing Encrypted Files

To edit an encrypted file in place, use the `ansible-vault edit` command. This command will decrypt the file to a temporary file and allow you to edit the file, saving it back when done and removing the temporary file:

```
ansible-vault edit foo.yml
```

## Rekeying Encrypted Files

Should you wish to change your password on a vault-encrypted file or files, you can do so with the rekey command:

```
ansible-vault rekey foo.yml bar.yml baz.yml
```

This command can rekey multiple data files at once and will ask for the original password and also the new password.

## Encrypting Unencrypted Files

If you have existing files that you wish to encrypt, use the ansible-vault encrypt command. This command can operate on multiple files at once:

```
ansible-vault encrypt foo.yml bar.yml baz.yml
```

## Decrypting Encrypted Files

If you have existing files that you no longer want to keep encrypted, you can permanently decrypt them by running the ansible-vault decrypt command. This command will save them unencrypted to the disk, so be sure you do not want ansible-vault edit instead:

```
ansible-vault decrypt foo.yml bar.yml baz.yml
```

## Viewing Encrypted Files

If you want to view the contents of an encrypted file without editing it, you can use the ansible-vault view command:

```
ansible-vault view foo.yml bar.yml baz.yml
```

## Use encrypt\_string to create encrypted variables to embed in yaml

The ansible-vault encrypt\_string command will encrypt and format a provided string into a format that can be included in ansible-playbook YAML files.

To encrypt a string provided as a cli arg:

```
ansible-vault encrypt_string --vault-id a_password_file 'foobar' --name 'the_secret'
```

Result:

```
the_secret: !vault |
  $ANSIBLE_VAULT;1.1;AES256
  62313365396662343061393464336163383764373764613633653634306231386433626436623361
  6134333665353966363534333632666535333761666131620a663537646436643839616531643561
  63396265333966386166373632626539326166353965363262633030333630313338646335303630
  3438626666666137650a353638643435666633633964366338633066623234616432373231333331
  6564
```

To use a vault-id label for 'dev' vault-id:

```
ansible-vault encrypt_string --vault-id dev@password 'foooodev' --name 'the_dev_secret'
↪ '
```

Result:

```
the_dev_secret: !vault |
    $ANSIBLE_VAULT;1.2;AES256;dev
↪ 30613233633461343837653833666333643061636561303338373661313838333565653635353162
↪ 3263363434623733343538653462613064333634333464660a663633623939393439316636633863
↪ 61636237636537333938306331383339353265363239643939666639386530626330633337633833
↪ 6664656334373166630a363736393262666465663432613932613036303963343263623137386239
    6330
```

To encrypt a string read from stdin and name it 'db\_password':

```
echo -n 'letmein' | ansible-vault encrypt_string --vault-id dev@password --stdin-name
↪ 'db_password'
```

Result:

```
Reading plaintext input from stdin. (ctrl-d to end input)
db_password: !vault |
    $ANSIBLE_VAULT;1.2;AES256;dev
↪ 61323931353866666336306139373937316366366138656131323863373866376666353364373761
↪ 3539633234313836346435323766306164626134376564330a373530313635343535343133316133
↪ 36643666306434616266376434363239346433643238336464643566386135356334303736353136
↪ 6565633133366366360a326566323363363936613664616364623437336130623133343530333739
    3039
```

To be prompted for a string to encrypt, encrypt it, and give it the name 'new\_user\_password':

```
ansible-vault encrypt_string --vault-id dev@./password --stdin-name 'new_user_password'
↪ '
```

Output:

```
Reading plaintext input from stdin. (ctrl-d to end input)
```

User enters 'hunter2' and hits ctrl-d.

Result:

```
new_user_password: !vault |
    $ANSIBLE_VAULT;1.2;AES256;dev
↪ 37636561366636643464376336303466613062633537323632306566653533383833366462366662
↪ 6565353063303065303831323539656138653863353230620a653638643639333133306331336365
↪ 62373737623337616130386137373461306535383538373162316263386165376131623631323434
↪ 3866363862363335620a376466656164383032633338306162326639643635663936623939666238
    3161
```

See also *Single Encrypted Variable*

## Vault Ids and Multiple Vault Passwords

*Available since Ansible 2.4*

A vault id is an identifier for one or more vault secrets. Since Ansible 2.4, Ansible supports multiple vault passwords. Vault ids is a way to provide a label for a particular vault password.

Vault encrypted content can specify which vault id it was encrypted with.

Prior to Ansible 2.4, only one vault password could be used at a time. Post Ansible 2.4, multiple vault passwords can be used each time Ansible runs, so any vault files or vars that needed to be decrypted all had to use the same password.

Since Ansible 2.4, vault files or vars that are encrypted with different passwords can be used at the same time.

For example, a playbook can now include a vars file encrypted with a 'dev' vault id and a 'prod' vault id.

## Providing Vault Passwords

Since Ansible 2.4, the recommended way to provide a vault password from the cli is to use the `--vault-id` cli option.

For example, to use a password store in the text file `/path/to/my/vault-password-file`:

```
ansible-playbook --vault-id /path/to/my/vault-password-file site.yml
```

To prompt for a password:

```
ansible-playbook --vault-id @prompt site.yml
```

To get the password from a vault password executable script `my-vault-password.py`:

```
ansible-playbook --vault-id my-vault-password.py
```

The value for `--vault-id` can specify the type of vault id (prompt, a file path, etc) and a label for the vault id ('dev', 'prod', 'cloud', etc)

For example, to use a password file `dev-password` for the vault-id 'dev':

```
ansible-playbook --vault-id dev@dev-password site.yml
```

To prompt for the 'dev' vault id:

```
ansible-playbook --vault-id dev@prompt site.yml
```

*Prior to Ansible 2.4*

To be prompted for a vault password, use the `--ask-vault-pass` cli option:

```
ansible-playbook --ask-vault-pass site.yml
```

To specify a vault password in a text file 'dev-password', use the `--vault-password-file` option:

```
ansible-playbook --vault-password-file dev-password site.yml
```

There is a config option (`DEFAULT_VAULT_PASSWORD_FILE`) to specify a vault password file to use without requiring the `--vault-password-file` cli option.

## Multiple vault passwords

Since Ansible 2.4 and later support using multiple vault passwords, `--vault-id` can be provided multiple times.

If multiple vault passwords are provided, by default Ansible will attempt to decrypt vault content by trying each vault secret in the order they were provided on the command line.

For example, to use a 'dev' password read from a file and to be prompted for the 'prod' password:

```
ansible-playbook --vault-id dev@dev-password --vault-id prod@prompt site.yml
```

In the above case, the 'dev' password will be tried first, then the 'prod' password for cases where Ansible doesn't know which vault id is used to encrypt something.

If the vault content was encrypted using a `--vault-id` option, then the label of the vault id is stored with the vault content. When Ansible knows the right vault-id, it will try the matching vault id's secret first before trying the rest of the vault-ids.

There is a config option (`DEFAULT_VAULT_ID_MATCH`) to force the vault content's vault id label to match with one of the provided vault ids. But the default is to try the matching id first, then try the other vault ids in order.

There is also a config option (`DEFAULT_VAULT_IDENTITY_LIST`) to specify a default list of vault ids to use. For example, instead of requiring the cli option on every use, the (`DEFAULT_VAULT_IDENTITY_LIST`) config option can be used:

```
ansible-playbook --vault-id dev@dev-password --vault-id prod@prompt site.yml
```

The `--vault-id` can be used in lieu of the `--vault-password-file` or `--ask-vault-pass` options, or it can be used in combination with them.

When using `ansible-vault` commands that encrypt content (`ansible-vault encrypt`, `ansible-vault encrypt_string`, etc) only one vault-id can be used.

---

: Prior to Ansible 2.4, only one vault password could be used in each Ansible run. The `--vault-id` option is not support prior to Ansible 2.4.

---

## Speeding Up Vault Operations

By default, Ansible uses PyCrypto to encrypt and decrypt vault files. If you have many encrypted files, decrypting them at startup may cause a perceptible delay. To speed this up, install the cryptography package:

```
pip install cryptography
```

## Vault Format

A vault encrypted file is a UTF-8 encoded txt file.

The file format includes a new line terminated header.

For example:

```
$ANSIBLE_VAULT;1.1;AES256
```

The header contains the vault format id, the vault format version, and a cipher id, separated by semi-colons ';':

The first field `$ANSIBLE_VAULT` is the format id. Currently `$ANSIBLE_VAULT` is the only valid file format id. This is used to identify files that are vault encrypted (via `vault.is_encrypted_file()`).

The second field (`1.1`) is the vault format version. All supported versions of ansible will currently default to '1.1'.

The '1.0' format is supported for reading only (and will be converted automatically to the '1.1' format on write). The format version is currently used as an exact string compare only (version numbers are not currently 'compared').

The third field (`AES256`) identifies the cipher algorithm used to encrypt the data. Currently, the only supported cipher is 'AES256'. [vault format 1.0 used 'AES', but current code always uses 'AES256']

Note: In the future, the header could change. Anything after the vault id and version can be considered to depend on the vault format version. This includes the cipher id, and any additional fields that could be after that.

The rest of the content of the file is the 'vaulttext'. The vaulttext is a text armored version of the encrypted ciphertext. Each line will be 80 characters wide, except for the last line which may be shorter.

### Vault Payload Format 1.1

The vaulttext is a concatenation of the ciphertext and a SHA256 digest with the result 'hexlified'.

'hexlify' refers to the `hexlify()` method of python's `binascii` module.

`hexlify()`'ed result of:

- `hexlify()`'ed string of the salt, followed by a newline ('\n')
- `hexlify()`'ed string of the crypted HMAC, followed by a newline. The HMAC is:
  - a [RFC2104](#) style HMAC
    - \* inputs are:
      - The AES256 encrypted ciphertext
      - A PBKDF2 key. This key, the cipher key, and the cipher iv are generated from:
        - the salt, in bytes
        - 10000 iterations
        - SHA256() algorithm
        - the first 32 bytes are the cipher key
        - the second 32 bytes are the HMAC key
        - remaining 16 bytes are the cipher iv
  - `hexlify()`'ed string of the ciphertext. The ciphertext is:
  - AES256 encrypted data. The data is encrypted using:
    - \* AES-CTR stream cipher
    - \* `b_pkey1`
    - \* `iv`
    - \* a 128 bit counter block seeded from an integer `iv`
    - \* the plaintext
      - the original plaintext
      - padding up to the AES256 blocksize. (The data used for padding is based on [RFC5652](#))

## 1.4.9 Working with Patterns

### Topics

- *Working with Patterns*

Patterns in Ansible are how we decide which hosts to manage. This can mean what hosts to communicate with, but in terms of *Working With Playbooks* it actually means what hosts to apply a particular configuration or IT process to.

We'll go over how to use the command line in *Introduction To Ad-Hoc Commands* section, however, basically it looks like this:

```
ansible <pattern_goes_here> -m <module_name> -a <arguments>
```

Such as:

```
ansible webserver -m service -a "name=httpd state=restarted"
```

A pattern usually refers to a set of groups (which are sets of hosts) – in the above case, machines in the "webserver" group.

Anyway, to use Ansible, you'll first need to know how to tell Ansible which hosts in your inventory to talk to. This is done by designating particular host names or groups of hosts.

The following patterns are equivalent and target all hosts in the inventory:

```
all
*
```

It is also possible to address a specific host or set of hosts by name:

```
one.example.com
one.example.com:two.example.com
192.0.2.50
192.0.2.*
```

The following patterns address one or more groups. Groups separated by a colon indicate an "OR" configuration. This means the host may be in either one group or the other:

```
webserver
webserver:dbserver
```

You can exclude groups as well, for instance, all machines must be in the group webserver but not in the group phoenix:

```
webserver:!phoenix
```

You can also specify the intersection of two groups. This would mean the hosts must be in the group webserver and the host must also be in the group staging:

```
webserver:&staging
```

You can do combinations:

```
webserver:dbserver:&staging:!phoenix
```



The above configuration means "all machines in the groups 'webservers' and 'dbservers' are to be managed if they are in the group 'staging' also, but the machines are not to be managed if they are in the group 'phoenix' ... whew!

You can also use variables if you want to pass some group specifiers via the "-e" argument to ansible-playbook, but this is uncommonly used:

```
webservers: !{{excluded}}: &{{required}}
```

You also don't have to manage by strictly defined groups. Individual host names, IPs and groups, can also be referenced using wildcards

```
*.example.com
*.com
```

It's also ok to mix wildcard patterns and groups at the same time:

```
one*.com:dbservers
```

You can select a host or subset of hosts from a group by their position. For example, given the following group:

```
[webservers]
cobweb
webbing
weber
```

You can refer to hosts within the group by adding a subscript to the group name:

```
webservers[0]      # == cobweb
webservers[-1]     # == weber
webservers[0:2]    # == webservers[0],webservers[1]
                  # == cobweb,webbing
webservers[1:]     # == webbing,weber
webservers[:3]     # == cobweb,webbing,weber
```

Most people don't specify patterns as regular expressions, but you can. Just start the pattern with a '~':

```
~(web|db).*\.example\.com
```

While we're jumping a bit ahead, additionally, you can add an exclusion criteria just by supplying the `--limit` flag to `/usr/bin/ansible` or `/usr/bin/ansible-playbook`:

```
ansible-playbook site.yml --limit datacenter2
```

And if you want to read the list of hosts from a file, prefix the file name with '@':

```
ansible-playbook site.yml --limit @retry_hosts.txt
```

Easy enough. See [Introduction To Ad-Hoc Commands](#) and then [Working With Playbooks](#) for how to apply this knowledge.

---

: You can use ',' instead of ':' as a host list separator. The ',' is preferred specially when dealing with ranges and ipv6.

---

:

[Introduction To Ad-Hoc Commands](#) Examples of basic commands

[Working With Playbooks](#) Learning ansible's configuration management language

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**irc.freenode.net** #ansible IRC chat channel

### 1.4.10 Working With Modules

#### Introduction

Modules (also referred to as "task plugins" or "library plugins") are discrete units of code that can be used from the command line or in a playbook task.

Let's review how we execute three different modules from the command line:

```
ansible webserver -m service -a "name=httpd state=started"
ansible webserver -m ping
ansible webserver -m command -a "/sbin/reboot -t now"
```

Each module supports taking arguments. Nearly all modules take `key=value` arguments, space delimited. Some modules take no arguments, and the `command/shell` modules simply take the string of the command you want to run.

From playbooks, Ansible modules are executed in a very similar way:

```
- name: reboot the servers
  action: command /sbin/reboot -t now
```

Which can be abbreviated to:

```
- name: reboot the servers
  command: /sbin/reboot -t now
```

Another way to pass arguments to a module is using `yml` syntax also called 'complex args'

```
- name: restart webserver
  service:
    name: httpd
    state: restarted
```

All modules technically return JSON format data, though if you are using the command line or playbooks, you don't really need to know much about that. If you're writing your own module, you care, and this means you do not have to write modules in any particular language – you get to choose.

Modules should be idempotent, and should avoid making any changes if they detect that the current state matches the desired final state. When using Ansible playbooks, these modules can trigger 'change events' in the form of notifying 'handlers' to run additional tasks.

Documentation for each module can be accessed from the command line with the `ansible-doc` tool:

```
ansible-doc yum
```

For a list of all available modules, see `./modules/modules_by_category`, or run the following at a command prompt:

```
ansible-doc -l
```

:

**Introduction To Ad-Hoc Commands** Examples of using modules in `/usr/bin/ansible`

**Working With Playbooks** Examples of using modules with `/usr/bin/ansible-playbook`

*Developing Modules* How to write your own modules

*Python API* Examples of using modules with the Python API

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**irc.freenode.net** #ansible IRC chat channel

## Return Values

### Topics

- *Return Values*
  - *Common*
    - \* *backup\_file*
    - \* *changed*
    - \* *failed*
    - \* *invocation*
    - \* *msg*
    - \* *rc*
    - \* *results*
    - \* *skipped*
    - \* *stderr*
    - \* *stderr\_lines*
    - \* *stdout*
    - \* *stdout\_lines*
  - *Internal use*
    - \* *ansible\_facts*
    - \* *exception*
    - \* *warnings*
    - \* *deprecations*

Ansible modules normally return a data structure that can be registered into a variable, or seen directly when output by the *ansible* program. Each module can optionally document its own unique return values (visible through *ansible-doc* and on the *main docsite*).

This document covers return values common to all modules.

---

: Some of these keys might be set by Ansible itself once it processes the module's return information.

---

### Common

#### **backup\_file**

For those modules that implement *backup=no*yes when manipulating files, a path to the backup file created.

#### **changed**

A boolean indicating if the task had to make changes.

#### **failed**

A boolean that indicates if the task was failed or not.

#### **invocation**

Information on how the module was invoked.

#### **msg**

A string with a generic message relayed to the user.

#### **rc**

Some modules execute command line utilities or are geared for executing commands directly (raw, shell, command, etc), this field contains 'return code' of these utilities.

#### **results**

If this key exists, it indicates that a loop was present for the task and that it contains a list of the normal module 'result' per item.

#### **skipped**

A boolean that indicates if the task was skipped or not

#### **stderr**

Some modules execute command line utilities or are geared for executing commands directly (raw, shell, command, etc), this field contains the error output of these utilities.

#### **stderr\_lines**

When *stderr* is returned we also always provide this field which is a list of strings, one item per line from the original.

## stdout

Some modules execute command line utilities or are geared for executing commands directly (raw, shell, command, etc). This field contains the normal output of these utilities.

## stdout\_lines

When *stdout* is returned, Ansible always provides a list of strings, each containing one item per line from the original output.

## Internal use

These keys can be added by modules but will be removed from registered variables; they are 'consumed' by Ansible itself.

## ansible\_facts

This key should contain a dictionary which will be appended to the facts assigned to the host. These will be directly accessible and don't require using a registered variable.

## exception

This key can contain traceback information caused by an exception in a module. It will only be displayed on high verbosity (-vvv).

## warnings

This key contains a list of strings that will be presented to the user.

## deprecations

This key contains a list of dictionaries that will be presented to the user. Keys of the dictionaries are *msg* and *version*, values are string, value for the *version* key can be an empty string.

:

**all\_modules** Learn about available modules

**GitHub Core modules directory** Browse source of core modules

**Github Extras modules directory** Browse source of extras modules.

**Mailing List** Development mailing list

**irc.freenode.net** #ansible IRC chat channel

## Module Maintenance & Support

To help identify maintainers and understand how the included modules are officially supported, each module now has associated metadata that provides additional clarity for maintenance and support.

### Core

Core Maintained modules are maintained by the Ansible Engineering Team. These modules are integral to the basic foundations of the Ansible distribution.

### Network

Network Maintained modules are maintained by the Ansible Network Team. Please note there are additional networking modules that are categorized as Certified or Community not maintained by Ansible.

### Certified

Certified modules are part of a future planned program currently in development.

### Community

Community Maintained modules are submitted and maintained by the Ansible community. These modules are not maintained by Ansible, and are included as a convenience.

### Issue Reporting

If you believe you have found a bug in a module and are already running the latest stable or development version of Ansible, first look at the [issue tracker in the Ansible repo](#) to see if an issue has already been filed. If not, please file one.

Should you have a question rather than a bug report, inquiries are welcome on the [ansible-project Google group](#) or on Ansible's "#ansible" channel, located on irc.freenode.net.

For development-oriented topics, use the [ansible-devel Google group](#) or Ansible's #ansible and #ansible-devel channels, located on irc.freenode.net. You should also read *Community Information & Contributing*, *Testing Ansible*, and *Developing Modules*.

The modules are hosted on GitHub in a subdirectory of the [Ansible repo](#).

NOTE: If you have a Red Hat Ansible Engine product subscription, please follow the standard issue reporting process via the Red Hat Customer Portal.

### Support

For more information on how included Ansible modules are supported by Red Hat, please refer to the following [knowledgebase article](#) as well as other resources on the [Red Hat Customer Portal](#).

:

**`./modules/modules_by_category`** A complete list of all available modules.

***Introduction To Ad-Hoc Commands*** Examples of using modules in /usr/bin/ansible

***Working With Playbooks*** Examples of using modules with /usr/bin/ansible-playbook

***Developing Modules*** How to write your own modules

***Python API*** Examples of using modules with the Python API

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**irc.freenode.net** #ansible IRC chat channel

Ansible ships with a number of modules (called the 'module library') that can be executed directly on remote hosts or through *Playbooks*.

Users can also write their own modules. These modules can control system resources, like services, packages, or files (anything really), or handle executing system commands.

:

*Introduction To Ad-Hoc Commands* Examples of using modules in /usr/bin/ansible

*Working With Playbooks* Examples of using modules with /usr/bin/ansible-playbook

*Developing Modules* How to write your own modules

*Python API* Examples of using modules with the Python API

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**irc.freenode.net** #ansible IRC chat channel

## 1.4.11 BSD Support

### Topics

- *BSD Support*
  - *Working with BSD*
  - *Bootstrapping BSD*
  - *Setting the Python interpreter*
  - *Which modules are available?*
  - *Using BSD as the control machine*
  - *BSD Facts*
  - *BSD Efforts and Contributions*

### Working with BSD

Ansible manages Linux/Unix machines using SSH by default. BSD machines are no exception, however this document covers some of the differences you may encounter with Ansible when working with BSD variants.

Typically, Ansible will try to default to using OpenSSH as a connection method. This is suitable when using SSH keys to authenticate, but when using SSH passwords, Ansible relies on sshpass. Most versions of sshpass do not deal particularly well with BSD login prompts, so when using SSH passwords against BSD machines, it is recommended to change the transport method to paramiko. You can do this in ansible.cfg globally or you can set it as an inventory/group/host variable. For example:

```
[freebsd]
mybsdhost1 ansible_connection=paramiko
```

Ansible is agentless by default, however certain software is required on the target machines. Using Python 2.4 on the agents requires an additional `py-simplejson` package/library to be installed, however this library is already included in Python 2.5 and above. Operating without Python is possible with the `raw` module. Although this module can be used to bootstrap Ansible and install Python on BSD variants (see below), it is very limited and the use of Python is required to make full use of Ansible's features.

### Bootstrapping BSD

As mentioned above, you can bootstrap Ansible with the `raw` module and remotely install Python on targets. The following example installs Python 2.7 which includes the `json` library required for full functionality of Ansible. On your control machine you can execute the following for most versions of FreeBSD:

```
ansible -m raw -a "pkg install -y python27" mybsdhost1
```

Or for most versions of OpenBSD:

```
ansible -m raw -a "pkg_add -z python-2.7"
```

Once this is done you can now use other Ansible modules apart from the `raw` module.

---

: This example demonstrated using `pkg` on FreeBSD and `pkg_add` on OpenBSD, however you should be able to substitute the appropriate package tool for your BSD; the package name may also differ. Refer to the package list or documentation of the BSD variant you are using for the exact Python package name you intend to install.

---

### Setting the Python interpreter

To support a variety of Unix/Linux operating systems and distributions, Ansible cannot always rely on the existing environment or `env` variables to locate the correct Python binary. By default, modules point at `/usr/bin/python` as this is the most common location. On BSD variants, this path may differ, so it is advised to inform Ansible of the binary's location, through the `ansible_python_interpreter` inventory variable. For example:

```
[freebsd:vars]
ansible_python_interpreter=/usr/local/bin/python2.7
[openbsd:vars]
ansible_python_interpreter=/usr/local/bin/python2.7
```

If you use additional plugins beyond those bundled with Ansible, you can set similar variables for `bash`, `perl` or `ruby`, depending on how the plugin is written. For example:

```
[freebsd:vars]
ansible_python_interpreter=/usr/local/bin/python
ansible_perl_interpreter=/usr/bin/perl5
```

### Which modules are available?

The majority of the core Ansible modules are written for a combination of Linux/Unix machines and other generic services, so most should function well on the BSDs with the obvious exception of those that are aimed at Linux-only technologies (such as LVG).



## Using BSD as the control machine

Using BSD as the control machine is as simple as installing the Ansible package for your BSD variant or by following the `pip` or 'from source' instructions.

## BSD Facts

Ansible gathers facts from the BSDs in a similar manner to Linux machines, but since the data, names and structures can vary for network, disks and other devices, one should expect the output to be slightly different yet still familiar to a BSD administrator.

## BSD Efforts and Contributions

BSD support is important to us at Ansible. Even though the majority of our contributors use and target Linux we have an active BSD community and strive to be as BSD friendly as possible. Please feel free to report any issues or incompatibilities you discover with BSD; pull requests with an included fix are also welcome!

:

*Introduction To Ad-Hoc Commands* Examples of basic commands

*Working With Playbooks* Learning ansible's configuration management language

*Developing Modules* How to write modules

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**irc.freenode.net** #ansible IRC chat channel

## 1.4.12 Windows Guides

The following sections provide information on managing Windows hosts with Ansible.

Because Windows is a non-POSIX-compliant operating system, there are differences between how Ansible interacts with them and the way Windows works. These guides will highlight some of the differences between Linux/Unix hosts and hosts running Windows.

## Setting up a Windows Host

This document discusses the setup that is required before Ansible can communicate with a Microsoft Windows host.

### Topics

- *Setting up a Windows Host*
  - *Host Requirements*
    - \* *Upgrading PowerShell and .NET Framework*
    - \* *WinRM Memory Hotfix*
  - *WinRM Setup*
    - \* *WinRM Listener*
      - *Setup WinRM Listener*

- *Delete WinRM Listener*
- \* *WinRM Service Options*
- \* *Common WinRM Issues*
  - *HTTP 401/Credentials Rejected*
  - *HTTP 500 Error*
  - *Timeout Errors*
  - *Connection Refused Errors*

## Host Requirements

For Ansible to communicate to a Windows host and use Windows modules, the Windows host must meet the following requirements:

- Ansible's supported Windows versions generally match those under current and extended support from Microsoft. Supported desktop OSs include Windows 7, 8.1, and 10, and supported server OSs are Windows Server 2008, 2008 R2, 2012, 2012 R2, and 2016.
- Ansible requires PowerShell 3.0 or newer and at least .NET 4.0 to be installed on the Windows host.
- A WinRM listener should be created and activated. More details for this can be found below.

---

: While these are the base requirements for Ansible connectivity, some Ansible modules have additional requirements, such as a newer OS or PowerShell version. Please consult the module's documentation page to determine whether a host meets those requirements.

---

## Upgrading PowerShell and .NET Framework

Ansible requires PowerShell version 3.0 and .NET Framework 4.0 or newer to function on older operating systems like Server 2008 and Windows 7. The base image does not meet this requirement. You can use the [Upgrade-PowerShell.ps1](#) script to update these.

This is an example of how to run this script from PowerShell:

```
$url = "https://raw.githubusercontent.com/jborean93/ansible-windows/master/scripts/
↪Upgrade-PowerShell.ps1"
$file = "$env:temp\Upgrade-PowerShell.ps1"
$username = "Administrator"
$password = "Password"

(New-Object -TypeName System.Net.WebClient).DownloadFile($url, $file)
Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Force

# version can be 3.0, 4.0 or 5.1
&$file -Version 5.1 -Username $username -Password $password -Verbose
```

Once completed, you will need to remove auto logon and set the execution policy back to the default of `Restricted`. You can do this with the following PowerShell commands:

```
# this isn't needed but is a good security practice to complete
Set-ExecutionPolicy -ExecutionPolicy Restricted -Force

$reg_winlogon_path = "HKLM:\Software\Microsoft\Windows NT\CurrentVersion\Winlogon"
Set-ItemProperty -Path $reg_winlogon_path -Name AutoAdminLogon -Value 0
Remove-ItemProperty -Path $reg_winlogon_path -Name DefaultUserName -ErrorAction_
↪ SilentlyContinue
Remove-ItemProperty -Path $reg_winlogon_path -Name DefaultPassword -ErrorAction_
↪ SilentlyContinue
```

The script works by checking to see what programs need to be installed (such as .NET Framework 4.5.2) and what PowerShell version is required. If a reboot is required and the username and password parameters are set, the script will automatically reboot and logon when it comes back up from the reboot. The script will continue until no more actions are required and the PowerShell version matches the target version. If the username and password parameters are not set, the script will prompt the user to manually reboot and logon when required. When the user is next logged in, the script will continue where it left off and the process continues until no more actions are required.

---

: If running on Server 2008, then SP2 must be installed. If running on Server 2008 R2 or Windows 7, then SP1 must be installed.

---



---

: Windows Server 2008 can only install PowerShell 3.0; specifying a newer version will result in the script failing.

---



---

: The username and password parameters are stored in plain text in the registry. Make sure the cleanup commands are run after the script finishes to ensure no credentials are still stored on the host.

---

## WinRM Memory Hotfix

When running on PowerShell v3.0, there is a bug with the WinRM service that limits the amount of memory available to WinRM. Without this hotfix installed, Ansible will fail to execute certain commands on the Windows host. These hotfixes should be installed as part of the system bootstrapping or imaging process. The script [Install-WMF3Hotfix.ps1](#) can be used to install the hotfix on affected hosts.

The following PowerShell command will install the hotfix:

```
$url = "https://raw.githubusercontent.com/jborean93/ansible-windows/master/scripts/
↪ Install-WMF3Hotfix.ps1"
$file = "$env:temp\Install-WMF3Hotfix.ps1"

(New-Object -TypeName System.Net.WebClient).DownloadFile($url, $file)
powershell.exe -ExecutionPolicy Bypass -File $file -Verbose
```

## WinRM Setup

Once Powershell has been upgraded to at least version 3.0, the final step is for the WinRM service to be configured so that Ansible can connect to it. There are two main components of the WinRM service that governs how Ansible can interface with the Windows host: the listener and the service configuration settings.

Details about each component can be read below, but the script [ConfigureRemotingForAnsible.ps1](#) can be used to set up the basics. This script sets up both HTTP and HTTPS listeners with a self-signed certificate and enables the `Basic`

authentication option on the service.

To use this script, run the following in PowerShell:

```
$url = "https://raw.githubusercontent.com/ansible/ansible/devel/examples/scripts/
↪ConfigureRemotingForAnsible.ps1"
$file = "$env:temp\ConfigureRemotingForAnsible.ps1"

(New-Object -TypeName System.Net.WebClient).DownloadFile($url, $file)

powershell.exe -ExecutionPolicy Bypass -File $file
```

There are different switches and parameters (like `-EnableCredSSP` and `-ForceNewSSLCert`) that can be set alongside this script. The documentation for these options are located at the top of the script itself.

---

: The `ConfigureRemotingForAnsible.ps1` script is intended for training and development purposes only and should not be used in a production environment, since it enables settings (like `Basic` authentication) that can be inherently insecure.

---

### WinRM Listener

The WinRM services listens for requests on one or more ports. Each of these ports must have a listener created and configured.

To view the current listeners that are running on the WinRM service, run the following command:

```
winrm enumerate winrm/config/Listener
```

This will output something like the following:

```
Listener
  Address = *
  Transport = HTTP
  Port = 5985
  Hostname
  Enabled = true
  URLPrefix = wsman
  CertificateThumbprint
  ListeningOn = 10.0.2.15, 127.0.0.1, 192.168.56.155, ::1, fe80::5efe:10.0.2.15%6, ↪
↪fe80::5efe:192.168.56.155%8, fe80::
ffff:ffff:ffff%2, fe80::203d:7d97:c2ed:ec78%3, fe80::e8ea:d765:2c69:7756%7

Listener
  Address = *
  Transport = HTTPS
  Port = 5986
  Hostname = SERVER2016
  Enabled = true
  URLPrefix = wsman
  CertificateThumbprint = E6CDAA82EEAF2ECE8546E05DB7F3E01AA47D76CE
  ListeningOn = 10.0.2.15, 127.0.0.1, 192.168.56.155, ::1, fe80::5efe:10.0.2.15%6, ↪
↪fe80::5efe:192.168.56.155%8, fe80::
ffff:ffff:ffff%2, fe80::203d:7d97:c2ed:ec78%3, fe80::e8ea:d765:2c69:7756%7
```

In the example above there are two listeners activated; one is listening on port 5985 over HTTP and the other is listening on port 5986 over HTTPS. Some of the key options that are useful to understand are:

- **Transport:** Whether the listener is run over HTTP or HTTPS, it is recommended to use a listener over HTTPS as the data is encrypted without any further changes required.
- **Port:** The port the listener runs on, by default it is 5985 for HTTP and 5986 for HTTPS. This port can be changed to whatever is required and corresponds to the host var `ansible_port`.
- **URLPrefix:** The URL prefix to listen on, by default it is `wsman`. If this is changed, the host var `ansible_winrm_path` must be set to the same value.
- **CertificateThumbprint:** If running over an HTTPS listener, this is the thumbprint of the certificate in the Windows Certificate Store that is used in the connection. To get the details of the certificate itself, run this command with the relevant certificate thumbprint in PowerShell:

```
$thumbprint = "E6CDAA82EEAF2ECE8546E05DB7F3E01AA47D76CE"
Get-ChildItem -Path cert:\LocalMachine\My -Recurse | Where-Object { $_.Thumbprint_
↪-eq $thumbprint } | Select-Object *
```

## Setup WinRM Listener

There are three ways to set up a WinRM listener:

- Using `winrm quickconfig` for HTTP or `winrm quickconfig -transport:https` for HTTPS. This is the easiest option to use when running outside of a domain environment and a simple listener is required. Unlike the other options, this process also has the added benefit of opening up the Firewall for the ports required and starts the WinRM service.
- Using Group Policy Objects. This is the best way to create a listener when the host is a member of a domain because the configuration is done automatically without any user input. For more information on group policy objects, see the [Group Policy Objects documentation](#).
- Using PowerShell to create the listener with a specific configuration. This can be done by running the following PowerShell commands:

```
$selector_set = @{
    Address = "*"
    Transport = "HTTPS"
}
$value_set = @{
    CertificateThumbprint = "E6CDAA82EEAF2ECE8546E05DB7F3E01AA47D76CE"
}

New-WSManInstance -ResourceURI "winrm/config/Listener" -SelectorSet $selector_set_
↪-ValueSet $value_set
```

To see the other options with this PowerShell cmdlet, see [New-WSManInstance](#).

---

: When creating an HTTPS listener, an existing certificate needs to be created and stored in the `LocalMachine\My` certificate store. Without a certificate being present in this store, most commands will fail.

---

## Delete WinRM Listener

To remove a WinRM listener:

```
# remove all listeners
Remove-Item -Path WSMAN:\localhost\Listener\* -Recurse -Force

# only remove listeners that are run over HTTPS
Get-ChildItem -Path WSMAN:\localhost\Listener | Where-Object { $_.Keys -contains
↪ "Transport=HTTPS" } | Remove-Item -Recurse -Force
```

---

: The Keys object is an array of strings, so it can contain different values. By default it contains a key for Transport= and Address= which correspond to the values from winrm enumerate winrm/config/Listeners.

---

## WinRM Service Options

There are a number of options that can be set to control the behavior of the WinRM service component, including authentication options and memory settings.

To get an output of the current service configuration options, run the following command:

```
winrm get winrm/config/Service
winrm get winrm/config/Winrs
```

This will output something like the following:

```
Service
  RootSDDL = O:NSG:BAD:P(A;;GA;;;BA)(A;;GR;;;IU)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD)
  MaxConcurrentOperations = 4294967295
  MaxConcurrentOperationsPerUser = 1500
  EnumerationTimeoutms = 240000
  MaxConnections = 300
  MaxPacketRetrievalTimeSeconds = 120
  AllowUnencrypted = false
  Auth
    Basic = true
    Kerberos = true
    Negotiate = true
    Certificate = true
    CredSSP = true
    CbtHardeningLevel = Relaxed
  DefaultPorts
    HTTP = 5985
    HTTPS = 5986
  IPv4Filter = *
  IPv6Filter = *
  EnableCompatibilityHttpListener = false
  EnableCompatibilityHttpsListener = false
  CertificateThumbprint
  AllowRemoteAccess = true

Winrs
  AllowRemoteShellAccess = true
  IdleTimeout = 7200000
  MaxConcurrentUsers = 2147483647
  MaxShellRunTime = 2147483647
  MaxProcessesPerShell = 2147483647
```

(continues on next page)

()

```
MaxMemoryPerShellMB = 2147483647
MaxShellsPerUser = 2147483647
```

While many of these options should rarely be changed, a few can easily impact the operations over WinRM and are useful to understand. Some of the important options are:

- `Service\AllowUnencrypted`: This option defines whether WinRM will allow traffic that is run over HTTP without message encryption. Message level encryption is only supported when `ansible_winrm_transport` is `ntlm`, `kerberos` or `credssp`. By default this is `false` and should only be set to `true` when debugging WinRM messages.
- `Service\Auth\*`: These flags define what authentication options are allowed with the WinRM service. By default, `Negotiate` (NTLM) and `Kerberos` are enabled.
- `Service\Auth\CbtHardeningLevel`: Specifies whether channel binding tokens are not verified (`None`), verified but not required (`Relaxed`), or verified and required (`Strict`). CBT is only used when connecting with NTLM or Kerberos over HTTPS. The downstream libraries that Ansible currently uses only support passing the CBT with NTLM authentication. Using Kerberos with `CbtHardeningLevel = Strict` will result in a 404 error.
- `Service\CertificateThumbprint`: This is the thumbprint of the certificate used to encrypt the TLS channel used with CredSSP authentication. By default this is empty; a self-signed certificate is generated when the WinRM service starts and is used in the TLS process.
- `Winrs\MaxShellRunTime`: This is the maximum time, in milliseconds, that a remote command is allowed to execute.
- `Winrs\MaxMemoryPerShellMB`: This is the maximum amount of memory allocated per shell, including the shell's child processes.

To modify a setting under the `Service` key in PowerShell, the following command can be used:

```
# substitute {path} with the path to the option after winrm/config/Service
Set-Item -Path WSMAN:\localhost\Service\{path} -Value "value here"

# for example, to change Service\Auth\CbtHardeningLevel run
Set-Item -Path WSMAN:\localhost\Service\Auth\CbtHardeningLevel -Value Strict
```

To modify a setting under the `Winrs` key in PowerShell, the following command can be used:

```
# substitute {path} with the path to the option after winrm/config/Winrs
Set-Item -Path WSMAN:\localhost\Shell\{path} -Value "value here"

# for example, to change Winrs\MaxShellRunTime run
Set-Item -Path WSMAN:\localhost\Shell\MaxShellRunTime -Value 2147483647
```

: If running in a domain environment, some of these options are set by GPO and cannot be changed on the host itself. When a key has been configured with GPO, it contains the text `[Source="GPO"]` next to the value.

## Common WinRM Issues

Because WinRM has a wide range of configuration options, it can be difficult to setup and configure. Because of this complexity, issues that are shown by Ansible could in fact be issues with the host setup instead.

One easy way to determine whether a problem is a host issue is to run the following command from another Windows host to connect to the target Windows host:

```
# test out HTTP
winrs -r:http://server:5985/wsman -u:Username -p:Password ipconfig

# test out HTTPS (will fail if the cert is not verifiable)
winrs -r:http://server:5985/wsman -u:Username -p:Password -ssl ipconfig

# test out HTTPS, ignoring certificate verification
$Username = "Username"
$Password = ConvertTo-SecureString -String "Password" -AsPlainText -Force
$Cred = New-Object -TypeName System.Management.Automation.PSCredential -ArgumentList
↪$Username, $Password

$SessionOption = New-PSSessionOption -SkipCACheck -SkipCNCheck -SkipRevocationCheck
Invoke-Command -ComputerName server -UseSSL -ScriptBlock { ipconfig } -Credential
↪$Cred -SessionOption $SessionOption
```

If this fails, the issue is probably related to the WinRM setup. If it works, the issue may not be related to the WinRM setup; please continue reading for more troubleshooting suggestions.

## HTTP 401/Credentials Rejected

A HTTP 401 error indicates the authentication process failed during the initial connection. Some things to check for this are:

- Verify that the credentials are correct and set properly in your inventory with `ansible_user` and `ansible_password`
- Ensure that the user is a member of the local Administrators group or has been explicitly granted access (a connection test with the `winrs` command can be used to rule this out).
- Make sure that the authentication option set by `ansible_winrm_transport` is enabled under `Service\Auth\*`
- If running over HTTP and not HTTPS, use `ntlm`, `kerberos` or `credssp` with `ansible_winrm_message_encryption: auto` to enable message encryption. If using another authentication option or if the installed `pywinrm` version cannot be upgraded, the `Service\AllowUnencrypted` can be set to `true` but this is only recommended for troubleshooting
- Ensure the downstream packages `pywinrm`, `requests-ntlm`, `requests-kerberos`, and/or `requests-credssp` are up to date using `pip`.
- If using Kerberos authentication, ensure that `Service\Auth\CbtHardeningLevel` is not set to `Strict`.
- When using Basic or Certificate authentication, make sure that the user is a local account and not a domain account. Domain accounts do not work with Basic and Certificate authentication.

## HTTP 500 Error

These indicate an error has occurred with the WinRM service. Some things to check for include:

- Verify that the number of current open shells has not exceeded either `WinRsMaxShellsPerUser` or any of the other Winrs quotas haven't been exceeded.



## Timeout Errors

These usually indicate an error with the network connection where Ansible is unable to reach the host. Some things to check for include:

- Make sure the firewall is not set to block the configured WinRM listener ports
- Ensure that a WinRM listener is enabled on the port and path set by the host vars
- Ensure that the `winrm` service is running on the Windows host and configured for automatic start

## Connection Refused Errors

These usually indicate an error when trying to communicate with the WinRM service on the host. Some things to check for:

- Ensure that the WinRM service is up and running on the host. Use `(Get-Service -Name winrm).Status` to get the status of the service.
- Check that the host firewall is allowing traffic over the WinRM port. By default this is 5985 for HTTP and 5986 for HTTPS.

Sometimes an installer may restart the WinRM or HTTP service and cause this error. The best way to deal with this is to use `win_psexec` from another Windows host.

:

**User Guide** The documentation index

**Working With Playbooks** An introduction to playbooks

**Best Practices** Best practices advice

**List of Windows Modules** Windows specific module list, all implemented in PowerShell

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Windows Remote Management

Unlike Linux/Unix hosts, which use SSH by default, Windows hosts are configured with WinRM. This topic covers how to configure and use WinRM with Ansible.

### Topics

- *Windows Remote Management*
  - *What is WinRM?*
  - *Authentication Options*
    - \* *Basic*
    - \* *Certificate*
      - *Generate a Certificate*
      - *Import a Certificate to the Certificate Store*
      - *Mapping a Certificate to an Account*

- \* *NTLM*
- \* *Kerberos*
  - *Installing the Kerberos Library*
  - *Configuring Host Kerberos*
  - *Automatic Kerberos Ticket Management*
  - *Manual Kerberos Ticket Management*
  - *Troubleshooting Kerberos*
- \* *CredSSP*
  - *Installing CredSSP Library*
  - *CredSSP and TLS 1.2*
  - *Set CredSSP Certificate*
- *Non-Administrator Accounts*
- *WinRM Encryption*
- *Inventory Options*
- *IPv6 Addresses*
- *HTTPS Certificate Validation*
- *Limitations*

## What is WinRM?

WinRM is a management protocol used by Windows to remotely communicate with another server. It is a SOAP-based protocol that communicates over HTTP/HTTPS, and is included in all recent Windows operating systems. Since Windows Server 2012, WinRM has been enabled by default, but in most cases extra configuration is required to use WinRM with Ansible.

Ansible uses the `pywinrm` package to communicate with Windows servers over WinRM. It is not installed by default with the Ansible package, but can be installed by running the following:

```
pip install "pywinrm>=0.3.0"
```

---

: on distributions with multiple python versions, use `pip2` or `pip2.x`, where `x` matches the python minor version Ansible is running under.

---

## Authentication Options

When connecting to a Windows host, there are several different options that can be used when authenticating with an account. The authentication type may be set on inventory hosts or groups with the `ansible_winrm_transport` variable.

The following matrix is a high level overview of the options:

Option	Local Accounts	Active Directory Accounts	Credential Delegation	HTTP Encryption
Basic	Yes	No	No	No
Certificate	Yes	No	No	No
Kerberos	No	Yes	Yes	Yes
NTLM	Yes	Yes	No	Yes
CredSSP	Yes	Yes	Yes	Yes

## Basic

Basic authentication is one of the simplest authentication options to use, but is also the most insecure. This is because the username and password are simply base64 encoded, and if a secure channel is not in use (eg, HTTPS) then it can be decoded by anyone. Basic authentication can only be used for local accounts (not domain accounts).

The following example shows host vars configured for basic authentication:

```
ansible_user: LocalUsername
ansible_password: Password
ansible_connection: winrm
ansible_winrm_transport: basic
```

Basic authentication is not enabled by default on a Windows host but can be enabled by running the following in PowerShell:

```
Set-Item -Path WSMan:\localhost\Service\Auth\Basic -Value $true
```

## Certificate

Certificate authentication uses certificates as keys similar to SSH key pairs, but the file format and key generation process is different.

The following example shows host vars configured for certificate authentication:

```
ansible_connection: winrm
ansible_winrm_cert_pem: /path/to/certificate/public/key.pem
ansible_winrm_cert_key_pem: /path/to/certificate/private/key.pem
ansible_winrm_transport: certificate
```

Certificate authentication is not enabled by default on a Windows host but can be enabled by running the following in PowerShell:

```
Set-Item -Path WSMan:\localhost\Service\Auth\Certificate -Value $true
```

---

: Encrypted private keys cannot be used as the urllib3 library that is used by Ansible for WinRM does not support this functionality.

---

## Generate a Certificate

A certificate must be generated before it can be mapped to a local user. This can be done using one of the following methods:

- OpenSSL
- PowerShell, using the `New-SelfSignedCertificate` cmdlet
- Active Directory Certificate Services

Active Directory Certificate Services is beyond of scope in this documentation but may be the best option to use when running in a domain environment. For more information, see the [Active Directory Certificate Services documentation](#).

---

: Using the PowerShell cmdlet `New-SelfSignedCertificate` to generate a certificate for authentication only works when being generated from a Windows 10 or Windows Server 2012 R2 host or later. OpenSSL is still required to extract the private key from the PFX certificate to a PEM file for Ansible to use.

---

To generate a certificate with OpenSSL:

```
# set the name of the local user that will have the key mapped to
USERNAME="username"

cat > openssl.conf << EOL
distinguished_name = req_distinguished_name
[req_distinguished_name]
[v3_req_client]
extendedKeyUsage = clientAuth
subjectAltName = otherName:1.3.6.1.4.1.311.20.2.3;UTF8:$USERNAME@localhost
EOL

export OPENSSL_CONF=openssl.conf
openssl req -x509 -nodes -days 3650 -newkey rsa:2048 -out cert.pem -outform PEM -
↳keyout cert_key.pem -subj "/CN=$USERNAME" -extensions v3_req_client
rm openssl.conf
```

To generate a certificate with `New-SelfSignedCertificate`:

```
# set the name of the local user that will have the key mapped
$Username = "username"
$output_path = "C:\temp"

# instead of generating a file, the cert will be added to the personal
# LocalComputer folder in the certificate store
$cert = New-SelfSignedCertificate -Type Custom `
    -Subject "CN=$Username" `
    -TextExtension @("2.5.29.37={text}1.3.6.1.5.5.7.3.2","2.5.29.17={text}upn=
↳$Username@localhost") `
    -KeyUsage DigitalSignature,KeyEncipherment `
    -KeyAlgorithm RSA `
    -KeyLength 2048

# export the public key
$pem_output = @()
$pem_output += "-----BEGIN CERTIFICATE-----"
$pem_output += [System.Convert]::ToBase64String($cert.RawData) -replace ".{64}", "$&`n
↳"
$pem_output += "-----END CERTIFICATE-----"
[System.IO.File]::WriteAllLines("$output_path\cert.pem", $pem_output)

# export the private key in a PFX file
[System.IO.File]::WriteAllBytes("$output_path\cert.pfx", $cert.Export("Pfx"))
```

---

: To convert the PFX file to a private key that pywinrm can use, run the following command with OpenSSL `openssl pkcs12 -in cert.pfx -nocerts -nodes -out cert_key.pem -passin pass: -passout pass:`

---

## Import a Certificate to the Certificate Store

Once a certificate has been generated, the issuing certificate needs to be imported into the Trusted Root Certificate Authorities of the LocalMachine store, and the client certificate public key must be present in the Trusted People folder of the LocalMachine store. For this example, both the issuing certificate and public key are the same.

Following example shows how to import the issuing certificate:

```
$cert = New-Object -TypeName System.Security.Cryptography.X509Certificates.  
↪X509Certificate2  
$cert.Import("cert.pem")  
  
$store_name = [System.Security.Cryptography.X509Certificates.StoreName]::Root  
$store_location = [System.Security.Cryptography.X509Certificates.  
↪StoreLocation]::LocalMachine  
$store = New-Object -TypeName System.Security.Cryptography.X509Certificates.X509Store_  
↪-ArgumentList $store_name, $store_location  
$store.Open("MaxAllowed")  
$store.Add($cert)  
$store.Close()
```

---

: If using ADCS to generate the certificate, then the issuing certificate will already be imported and this step can be skipped.

---

The code to import the client certificate public key is:

```
$cert = New-Object -TypeName System.Security.Cryptography.X509Certificates.  
↪X509Certificate2  
$cert.Import("cert.pem")  
  
$store_name = [System.Security.Cryptography.X509Certificates.StoreName]::TrustedPeople  
$store_location = [System.Security.Cryptography.X509Certificates.  
↪StoreLocation]::LocalMachine  
$store = New-Object -TypeName System.Security.Cryptography.X509Certificates.X509Store_  
↪-ArgumentList $store_name, $store_location  
$store.Open("MaxAllowed")  
$store.Add($cert)  
$store.Close()
```

## Mapping a Certificate to an Account

Once the certificate has been imported, it needs to be mapped to the local user account.

This can be done with the following PowerShell command:

```

$username = "username"
$password = ConvertTo-SecureString -String "password" -AsPlainText -Force
$credential = New-Object -TypeName System.Management.Automation.PSCredential -
    ↪ArgumentList $username, $password

# this is the issuer thumbprint which in the case of a self generated cert
# is the public key thumbprint, additional logic may be required for other
# scenarios
$thumbprint = (Get-ChildItem -Path cert:\LocalMachine\root | Where-Object { $_.
    ↪Subject -eq "CN=$username" }).Thumbprint

New-Item -Path WSMAN:\localhost\ClientCertificate `
    -Subject "$username@localhost" `
    -URI * `
    -Issuer $thumbprint `
    -Credential $credential `
    -Force

```

Once this is complete, the hostvar `ansible_winrm_cert_pem` should be set to the path of the public key and the `ansible_winrm_cert_key_pem` variable should be set to the path of the private key.

## NTLM

NTLM is an older authentication mechanism used by Microsoft that can support both local and domain accounts. NTLM is enabled by default on the WinRM service, so no setup is required before using it.

NTLM is the easiest authentication protocol to use and is more secure than Basic authentication. If running in a domain environment, Kerberos should be used instead of NTLM.

Kerberos has several advantages over using NTLM:

- NTLM is an older protocol and does not support newer encryption protocols.
- NTLM is slower to authenticate because it requires more round trips to the host in the authentication stage.
- Unlike Kerberos, NTLM does not allow credential delegation.

This example shows host variables configured to use NTLM authentication:

```

ansible_user: LocalUsername
ansible_password: Password
ansible_connection: winrm
ansible_winrm_transport: ntlm

```

## Kerberos

Kerberos is the recommended authentication option to use when running in a domain environment. Kerberos supports features like credential delegation and message encryption over HTTP and is one of the more secure options that is available through WinRM.

Kerberos requires some additional setup work on the Ansible host before it can be used properly.

The following example shows host vars configured for Kerberos authentication:

```

ansible_user: username@MY.DOMAIN.COM
ansible_password: Password

```

(continues on next page)

()

```
ansible_connection: winrm
ansible_winrm_transport: kerberos
```

As of Ansible version 2.3, the Kerberos ticket will be created based on `ansible_user` and `ansible_password`. If running on an older version of Ansible or when `ansible_winrm_kinit_mode` is `manual`, a Kerberos ticket must already be obtained. See below for more details.

There are some extra host variables that can be set:

```
ansible_winrm_kinit_mode: managed/manual (manual means Ansible will not obtain a
↪ticket)
ansible_winrm_kinit_cmd: the kinit binary to use to obtain a Kerberos ticket (default
↪to kinit)
ansible_winrm_service: overrides the SPN prefix that is used, the default is ``HTTP``
↪and should rarely ever need changing
ansible_winrm_kerberos_delegation: allows the credentials to traverse multiple hops
ansible_winrm_kerberos_hostname_override: the hostname to be used for the kerberos
↪exchange
```

## Installing the Kerberos Library

Some system dependencies that must be installed prior to using Kerberos. The script below lists the dependencies based on the distro:

```
# Via Yum (RHEL/Centos/Fedora)
yum -y install python-devel krb5-devel krb5-libs krb5-workstation

# Via Apt (Ubuntu)
sudo apt-get install python-dev libkrb5-dev krb5-user

# Via Portage (Gentoo)
emerge -av app-crypt/mit-krb5
emerge -av dev-python/setuptools

# Via Pkg (FreeBSD)
sudo pkg install security/krb5

# Via OpenCSW (Solaris)
pkgadd -d http://get.opencsw.org/now
/opt/csw/bin/pkgutil -U
/opt/csw/bin/pkgutil -y -i libkrb5_3

# Via Pacman (Arch Linux)
pacman -S krb5
```

Once the dependencies have been installed, the `python-kerberos` wrapper can be install using `pip`:

```
pip install pywinrm[kerberos]
```

## Configuring Host Kerberos

Once the dependencies have been installed, Kerberos needs to be configured so that it can communicate with a domain. This configuration is done through the `/etc/krb5.conf` file, which is installed with the packages in the script above.

To configure Kerberos, in the section that starts with:

```
[realms]
```

Add the full domain name and the fully qualified domain names of the primary and secondary Active Directory domain controllers. It should look something like this:

```
[realms]
  MY.DOMAIN.COM = {
    kdc = domain-controller1.my.domain.com
    kdc = domain-controller2.my.domain.com
  }
```

In the section that starts with:

```
[domain_realm]
```

Add a line like the following for each domain that Ansible needs access for:

```
[domain_realm]
  .my.domain.com = MY.DOMAIN.COM
```

You can configure other settings in this file such as the default domain. See [krb5.conf](#) for more details.

### Automatic Kerberos Ticket Management

Ansible version 2.3 and later defaults to automatically managing Kerberos tickets when both `ansible_user` and `ansible_password` are specified for a host. In this process, a new ticket is created in a temporary credential cache for each host. This is done before each task executes to minimize the chance of ticket expiration. The temporary credential caches are deleted after each task completes and will not interfere with the default credential cache.

To disable automatic ticket management, set `ansible_winrm_kinit_mode=manual` via the inventory.

Automatic ticket management requires a standard `kinit` binary on the control host system path. To specify a different location or binary name, set the `ansible_winrm_kinit_cmd` hostvar to the fully qualified path to a MIT `krb5` `kinit`-compatible binary.

### Manual Kerberos Ticket Management

To manually manage Kerberos tickets, the `kinit` binary is used. To obtain a new ticket the following command is used:

```
kinit username@MY.DOMAIN.COM
```

---

: The domain must match the configured Kerberos realm exactly, and must be in upper case.

---

To see what tickets (if any) have been acquired, use the following command:

```
klist
```

To destroy all the tickets that have been acquired, use the following command:

```
kdestroy
```



## Troubleshooting Kerberos

Kerberos is reliant on a properly-configured environment to work. To troubleshoot Kerberos issues, ensure that:

- The hostname set for the Windows host is the FQDN and not an IP address.
- The forward and reverse DNS lookups are working properly in the domain. To test this, ping the windows host by name and then use the ip address returned with `nslookup`. The same name should be returned when using `nslookup` on the IP address.
- The Ansible host's clock is synchronized with the domain controller. Kerberos is time sensitive, and a little clock drift can cause the ticket generation process to fail.
- Ensure that the fully qualified domain name for the domain is configured in the `krb5.conf` file. To check this, run:

```
kinit -C username@MY.DOMAIN.COM
klist
```

If the domain name returned by `klist` is different from the one requested, an alias is being used. The `krb5.conf` file needs to be updated so that the fully qualified domain name is used and not an alias.

## CredSSP

CredSSP authentication is a newer authentication protocol that allows credential delegation. This is achieved by encrypting the username and password after authentication has succeeded and sending that to the server using the CredSSP protocol.

Because the username and password are sent to the server to be used for double hop authentication, ensure that the hosts that the Windows host communicates with are not compromised and are trusted.

CredSSP can be used for both local and domain accounts and also supports message encryption over HTTP.

To use CredSSP authentication, the host vars are configured like so:

```
ansible_user: Username
ansible_password: Password
ansible_connection: winrm
ansible_winrm_transport: credssp
```

There are some extra host variables that can be set as shown below:

```
ansible_winrm_credssp_disable_tlsv1_2: when true, will not use TLS 1.2 in the CredSSP
↳auth process
```

CredSSP authentication is not enabled by default on a Windows host, but can be enabled by running the following in PowerShell:

```
Enable-WSManCredSSP -Role Server -Force
```

## Installing CredSSP Library

The `requests-credssp` wrapper can be installed using `pip`:

```
pip install pywinrm[credssp]
```

## CredSSP and TLS 1.2

By default the `requests-credssp` library is configured to authenticate over the TLS 1.2 protocol. TLS 1.2 is installed and enabled by default for Windows Server 2012 and Windows 8 and more recent releases.

There are two ways that older hosts can be used with CredSSP:

- Install and enable a hotfix to enable TLS 1.2 support (recommended for Server 2008 R2 and Windows 7).
- Set `ansible_winrm_credssp_disable_tlsv1_2=True` in the inventory to run over TLS 1.0. This is the only option when connecting to Windows Server 2008, which has no way of supporting TLS 1.2

To enable TLS 1.2 support on Server 2008 R2 and Windows 7, the optional update [KRB3080079](#) needs to be installed.

Once the update has been applied and the Windows host rebooted, run the following PowerShell commands to enable TLS 1.2:

```
$reg_path =  
↪ "HKLM:\SYSTEM\CurrentControlSet\Control\SecurityProvider\SCHANNEL\Protocols\TLS 1.2"  
New-Item -Path $reg_path  
New-Item -Path "$reg_path\Server"  
New-Item -Path "$reg_path\Client"  
  
New-ItemProperty -Path "$reg_path\Server" -Name "Enabled" -Value 1 -PropertyType DWord  
New-ItemProperty -Path "$reg_path\Server" -Name "DisabledByDefault" -Value 0 -  
↪ PropertyType DWord  
New-ItemProperty -Path "$reg_path\Client" -Name "Enabled" -Value 1 -PropertyType DWord  
New-ItemProperty -Path "$reg_path\Client" -Name "DisabledByDefault" -Value 0 -  
↪ PropertyType DWord
```

## Set CredSSP Certificate

CredSSP works by encrypting the credentials through the TLS protocol and uses a self-signed certificate by default. The `CertificateThumbprint` option under the WinRM service configuration can be used to specify the thumbprint of another certificate.

---

: This certificate configuration is independent of the WinRM listener certificate. With CredSSP, message transport still occurs over the WinRM listener, but the TLS-encrypted messages inside the channel use the service-level certificate.

---

To explicitly set the certificate to use for CredSSP:

```
# note the value $certificate_thumbprint will be different in each  
# situation, this needs to be set based on the cert that is used.  
$certificate_thumbprint = "7C8DCBD5427AFEE6560F4AF524E325915F51172C"  
  
# set the thumbprint value  
Set-Item -Path WSMAN:\localhost\Service\CertificateThumbprint -Value $certificate_  
↪ thumbprint
```

## Non-Administrator Accounts

WinRM is configured by default to only allow connections from accounts in the local Administrators group. This can be changed by running:

```
winrm configSDDL default
```

This will display an ACL editor, where new users or groups may be added. To run commands over WinRM, users and groups must have at least the `Read` and `Execute` permissions enabled.

While non-administrative accounts can be used with WinRM, most typical server administration tasks require some level of administrative access, so the utility is usually limited.

## WinRM Encryption

By default WinRM will fail to work when running over an unencrypted channel. The WinRM protocol considers the channel to be encrypted if using TLS over HTTP (HTTPS) or using message level encryption. Using WinRM with TLS is the recommended option as it works with all authentication options, but requires a certificate to be created and used on the WinRM listener.

The `ConfigureRemotingForAnsible.ps1` creates a self-signed certificate and creates the listener with that certificate. If in a domain environment, AD CS can also create a certificate for the host that is issued by the domain itself.

If using HTTPS is not an option, then HTTP can be used when the authentication option is NTLM, Kerberos or CredSSP. These protocols will encrypt the WinRM payload with their own encryption method before sending it to the server. The message-level encryption is not used when running over HTTPS because the encryption uses the more secure TLS protocol instead. If both transport and message encryption is required, set `ansible_winrm_message_encryption=always` in the host vars.

A last resort is to disable the encryption requirement on the Windows host. This should only be used for development and debugging purposes, as anything sent from Ansible can viewed by anyone on the network. To disable the encryption requirement, run the following from PowerShell on the target host:

```
Set-Item -Path WSMan:\localhost\Service\AllowUnencrypted -Value $true
```

: Do not disable the encryption check unless it is absolutely required. Doing so could allow sensitive information like credentials and files to be intercepted by others on the network.

## Inventory Options

Ansible's Windows support relies on a few standard variables to indicate the username, password, and connection type of the remote hosts. These variables are most easily set up in the inventory, but can be set on the `host_vars/` `group_vars` level.

When setting up the inventory, the following variables are required:

```
# it is suggested that these be encrypted with ansible-vault:
# ansible-vault edit group_vars/windows.yml
ansible_connection: winrm

# may also be passed on the command-line via --user
ansible_user: Administrator

# may also be supplied at runtime with --ask-pass
ansible_password: SecretPasswordGoesHere
```

Using the variables above, Ansible will connect to the Windows host with Basic authentication through HTTPS. If `ansible_user` has a UPN value like `username@MY.DOMAIN.COM` then the authentication option will automatically attempt to use Kerberos unless `ansible_winrm_transport` has been set to something other than `kerberos`.

The following custom inventory variables are also supported for additional configuration of WinRM connections:

- `ansible_port`: The port WinRM will run over, HTTPS is 5986 which is the default while HTTP is 5985
- `ansible_winrm_scheme`: Specify the connection scheme (`http` or `https`) to use for the WinRM connection. Ansible uses `https` by default unless `ansible_port` is 5985
- `ansible_winrm_path`: Specify an alternate path to the WinRM endpoint, Ansible uses `/wsman` by default
- `ansible_winrm_realm`: Specify the realm to use for Kerberos authentication. If `ansible_user` contains `@`, Ansible will use the part of the username after `@` by default
- `ansible_winrm_transport`: Specify one or more authentication transport options as a comma-separated list. By default, Ansible will use `kerberos`, `basic` if the `kerberos` module is installed and a realm is defined, otherwise it will be `plaintext`
- `ansible_winrm_server_cert_validation`: Specify the server certificate validation mode (`ignore` or `validate`). Ansible defaults to `validate` on Python 2.7.9 and higher, which will result in certificate validation errors against the Windows self-signed certificates. Unless verifiable certificates have been configured on the WinRM listeners, this should be set to `ignore`
- `ansible_winrm_operation_timeout_sec`: Increase the default timeout for WinRM operations, Ansible uses 20 by default
- `ansible_winrm_read_timeout_sec`: Increase the WinRM read timeout, Ansible uses 30 by default. Useful if there are intermittent network issues and read timeout errors keep occurring
- `ansible_winrm_message_encryption`: Specify the message encryption operation (`auto`, `always`, `never`) to use, Ansible uses `auto` by default. `auto` means message encryption is only used when `ansible_winrm_scheme` is `http` and `ansible_winrm_transport` supports message encryption. `always` means message encryption will always be used and `never` means message encryption will never be used
- `ansible_winrm_ca_trust_path`: Used to specify a different cacert container than the one used in the `certifi` module. See the HTTPS Certificate Validation section for more details.
- `ansible_winrm_send_cbt`: When using `ntlm` or `kerberos` over HTTPS, the authentication library will try to send channel binding tokens to mitigate against man in the middle attacks. This flag controls whether these bindings will be sent or not (default: `True`).
- `ansible_winrm_*`: Any additional keyword arguments supported by `winrm.Protocol` may be provided in place of `*`

In addition, there are also specific variables that need to be set for each authentication option. See the section on authentication above for more information.

---

: Ansible 2.0 has deprecated the "ssh" from `ansible_ssh_user`, `ansible_ssh_pass`, `ansible_ssh_host`, and `ansible_ssh_port` to become `ansible_user`, `ansible_password`, `ansible_host`, and `ansible_port`. If using a version of Ansible prior to 2.0, the older style (`ansible_ssh_*`) should be used instead. The shorter variables are ignored, without warning, in older versions of Ansible.

---

---

: `ansible_winrm_message_encryption` is different from transport encryption done over TLS. The WinRM payload is still encrypted with TLS when run over HTTPS, even if

---

```
ansible_winrm_message_encryption=never.
```

---

## IPv6 Addresses

IPv6 addresses can be used instead of IPv4 addresses or hostnames. This option is normally set in an inventory. Ansible will attempt to parse the address using the `ipaddress` package and pass to `pywinrm` correctly.

When defining a host using an IPv6 address, just add the IPv6 address as you would an IPv4 address or hostname:

```
[windows-server]
2001:db8::1

[windows-server:vars]
ansible_user=username
ansible_password=password
ansible_connection=winrm
```

---

: The `ipaddress` library is only included by default in Python 3.x. To use IPv6 addresses in Python 2.6 and 2.7, make sure to run `pip install ipaddress` which installs a backported package.

---

## HTTPS Certificate Validation

As part of the TLS protocol, the certificate is validated to ensure the host matches the subject and the client trusts the issuer of the server certificate. When using a self-signed certificate or setting `ansible_winrm_server_cert_validation: ignore` these security mechanisms are bypassed. While self signed certificates will always need the `ignore` flag, certificates that have been issued from a certificate authority can still be validated.

One of the more common ways of setting up a HTTPS listener in a domain environment is to use Active Directory Certificate Service (AD CS). AD CS is used to generate signed certificates from a Certificate Signing Request (CSR). If the WinRM HTTPS listener is using a certificate that has been signed by another authority, like AD CS, then Ansible can be set up to trust that issuer as part of the TLS handshake.

To get Ansible to trust a Certificate Authority (CA) like AD CS, the issuer certificate of the CA can be exported as a PEM encoded certificate. This certificate can then be copied locally to the Ansible controller and used as a source of certificate validation, otherwise known as a CA chain.

The CA chain can contain a single or multiple issuer certificates and each entry is contained on a new line. To then use the custom CA chain as part of the validation process, set `ansible_winrm_ca_trust_path` to the path of the file. If this variable is not set, the default CA chain is used instead which is located in the install path of the Python package `certifi`.

---

: Each HTTP call is done by the Python requests library which does not use the systems built-in certificate store as a trust authority. Certificate validation will fail if the server's certificate issuer is only added to the system's truststore.

---

## Limitations

Due to the design of the WinRM protocol, there are a few limitations when using WinRM that can cause issues when creating playbooks for Ansible. These include:

- Credentials are not delegated for most authentication types, which causes authentication errors when accessing network resources or installing certain programs.
- Many calls to the Windows Update API are blocked when running over WinRM.
- Some programs fail to install with WinRM due to no credential delegation or because they access forbidden Windows API like WUA over WinRM.
- Commands under WinRM are done under a non-interactive session, which can prevent certain commands or executables from running.
- You cannot run a process that interacts with DPAPI, which is used by some installers (like Microsoft SQL Server).

Some of these limitations can be mitigated by doing one of the following:

- Set `ansible_winrm_transport` to `credssp` or `kerberos` (with `ansible_winrm_kerberos_delegation=true`) to bypass the double hop issue and access network resources
- Use `become` to bypass all WinRM restrictions and run a command as it would locally. Unlike using an authentication transport like `credssp`, this will also remove the non-interactive restriction and API restrictions like WUA and DPAPI
- Use a scheduled task to run a command which can be created with the `win_scheduled_task` module. Like `become`, this bypasses all WinRM restrictions but can only run a command and not modules.

:

**User Guide** The documentation index

**Working With Playbooks** An introduction to playbooks

**Best Practices** Best practices advice

**List of Windows Modules** Windows specific module list, all implemented in PowerShell

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Using Ansible and Windows

When using Ansible to manage Windows, many of the syntax and rules that apply for Unix/Linux hosts also apply to Windows, but there are still some differences when it comes to components like path separators and OS-specific tasks. This document covers details specific to using Ansible for Windows.

### Topics

- *Using Ansible and Windows*
  - *Use Cases*
    - \* *Installing Software*
    - \* *Installing Updates*
    - \* *Set Up Users and Groups*
      - *Local*
      - *Domain*

- \* *Running Commands*
  - *Choosing Command or Shell*
  - *Argument Rules*
- \* *Creating and Running a Scheduled Task*
- *Path Formatting for Windows*
- \* *YAML Style*
- \* *Legacy key=value Style*
- *Limitations*
- *Developing Windows Modules*

## Use Cases

Ansible can be used to orchestrate a multitude of tasks on Windows servers. Below are some examples and info about common tasks.

## Installing Software

There are three main ways that Ansible can be used to install software:

- Using the `win_chocolatey` module. This sources the program data from the default public [Chocolatey](#) repository. Internal repositories can be used instead by setting the `source` option.
- Using the `win_package` module. This installs software using an MSI or .exe installer from a local/network path or URL.
- Using the `win_command` or `win_shell` module to run an installer manually.

The `win_chocolatey` module is recommended since it has the most complete logic for checking to see if a package has already been installed and is up-to-date.

Below are some examples of using all three options to install 7-Zip:

```
# install/uninstall with chocolatey
- name: ensure 7-Zip is installed via Chocolatey
  win_chocolatey:
    name: 7zip
    state: present

- name: ensure 7-Zip is not installed via Chocolatey
  win_chocolatey:
    name: 7zip
    state: absent

# install/uninstall with win_package
- name: download the 7-Zip package
  win_get_url:
    url: http://www.7-zip.org/a/7z1701-x64.msi
    dest: C:\temp\7z.msi

- name: ensure 7-Zip is installed via win_package
```

(continues on next page)

```

win_package:
  path: C:\temp\7z.msi
  state: present

- name: ensure 7-Zip is not installed via win_package
  win_package:
    path: C:\temp\7z.msi
    state: absent

# install/uninstall with win_command
- name: download the 7-Zip package
  win_get_url:
    url: http://www.7-zip.org/a/7z1701-x64.msi
    dest: C:\temp\7z.msi

- name: check if 7-Zip is already installed
  win_reg_stat:
    name: HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\{23170F69-40C1-
↪2702-1701-000001000000}
    register: 7zip_installed

- name: ensure 7-Zip is installed via win_command
  win_command: C:\Windows\System32\msiexec.exe /i C:\temp\7z.msi /qn /norestart
  when: 7zip_installed.exists == False

- name: ensure 7-Zip is uninstalled via win_command
  win_command: C:\Windows\System32\msiexec.exe /x {23170F69-40C1-2702-1701-
↪000001000000} /qn /norestart
  when: 7zip_installed.exists == True

```

Some installers like Microsoft Office or SQL Server require credential delegation or access to components restricted by WinRM. The best method to bypass these issues is to use `become` with the task. With `become`, Ansible will run the installer as if it were run interactively on the host.

---

: Many installers do not properly pass back error information over WinRM. In these cases, if the install has been verified to work locally the recommended method is to use `become`.

---



---

: Some installers restart the WinRM or HTTP services, or cause them to become temporarily unavailable, making Ansible assume the system is unreachable.

---

## Installing Updates

The `win_updates` and `win_hotfix` modules can be used to install updates or hotfixes on a host. The module `win_updates` is used to install multiple updates by category, while `win_hotfix` can be used to install a single update or hotfix file that has been downloaded locally.

---

: The `win_hotfix` module has a requirement that the DISM PowerShell cmdlets are present. These cmdlets were only added by default on Windows Server 2012 and newer and must be installed on older Windows hosts.

---

The following example shows how `win_updates` can be used:



```

- name: install all critical and security updates
  win_updates:
    category_names:
      - CriticalUpdates
      - SecurityUpdates
    state: installed
    register: update_result

- name: reboot host if required
  win_reboot:
    when: update_result.reboot_required

```

The following example show how `win_hotfix` can be used to install a single update or hotfix:

```

- name: download KB3172729 for Server 2012 R2
  win_get_url:
    url: http://download.windowsupdate.com/d/msdownload/update/software/secu/2016/07//
    ↪ windows8.1-kb3172729-x64_e8003822a7ef4705cbb65623b72fd3cec73fe222.msu
    dest: C:\temp\KB3172729.msu

- name: install hotfix
  win_hotfix:
    hotfix_kb: KB3172729
    source: C:\temp\KB3172729.msu
    state: present
    register: hotfix_result

- name: reboot host if required
  win_reboot:
    when: hotfix_result.reboot_required

```

## Set Up Users and Groups

Ansible can be used to create Windows users and groups both locally and on a domain.

### Local

The modules `win_user`, `win_group` and `win_group_membership` manage Windows users, groups and group memberships locally.

The following is an example of creating local accounts and groups that can access a folder on the same host:

```

- name: create local group to contain new users
  win_group:
    name: LocalGroup
    description: Allow access to C:\Development folder

- name: create local user
  win_user:
    name: '{{item.name}}'
    password: '{{item.password}}'
    groups: LocalGroup
    update_password: no
    password_never_expired: yes

```

(continues on next page)

```
with_items:
- name: User1
  password: Password1
- name: User2
  password: Password2

- name: create Development folder
  win_file:
    path: C:\Development
    state: directory

- name: set ACL of Development folder
  win_acl:
    path: C:\Development
    rights: FullControl
    state: present
    type: allow
    user: LocalGroup

- name: remove parent inheritance of Development folder
  win_acl_inheritance:
    path: C:\Development
    reorganize: yes
    state: absent
```

## Domain

The modules `win_domain_user` and `win_domain_group` manages users and groups in a domain. The below is an example of ensuring a batch of domain users are created:

```
- name: ensure each account is created
  win_domain_user:
    name: '{{item.name}}'
    upn: '{{item.name}}@MY.DOMAIN.COM'
    password: '{{item.password}}'
    password_never_expires: no
    groups:
      - Test User
      - Application
    company: Ansible
    update_password: on_create
  with_items:
    - name: Test User
      password: Password
    - name: Admin User
      password: SuperSecretPass01
    - name: Dev User
      password: '@fvr3IbFBujSRh!3hBg%wgFucD8^x8W5'
```

## Running Commands

In cases where there is no appropriate module available for a task, a command or script can be run using the `win_shell`, `win_command`, `raw`, and `script` modules.

The `raw` module simply executes a Powershell command remotely. Since `raw` has none of the wrappers that Ansible typically uses, `become`, `async` and environment variables do not work.

The `script` module executes a script from the Ansible controller on one or more Windows hosts. Like `raw`, `script` currently does not support `become`, `async`, or environment variables.

The `win_command` module is used to execute a command which is either an executable or batch file, while the `win_shell` module is used to execute commands within a shell.

## Choosing Command or Shell

The `win_shell` and `win_command` modules can both be used to execute a command or commands. The `win_shell` module is run within a shell-like process like PowerShell or `cmd`, so it has access to shell operators like `<`, `>`, `|`, `;`, `&&`, and `||`. Multi-lined commands can also be run in `win_shell`.

The `win_command` module simply runs a process outside of a shell. It can still run a shell command like `mkdir` or `New-Item` by passing the shell commands to a shell executable like `cmd.exe` or `Powershell.exe`.

Here are some examples of using `win_command` and `win_shell`:

```
- name: run a command under PowerShell
  win_shell: Get-Service -Name service | Stop-Service

- name: run a command under cmd
  win_shell: mkdir C:\temp
  args:
    executable: cmd.exe

- name: run a multiple shell commands
  win_shell: |
    New-Item -Path C:\temp -ItemType Directory
    Remove-Item -Path C:\temp -Force -Recurse
    $path_info = Get-Item -Path C:\temp
    $path_info.FullName

- name: run an executable using win_command
  win_command: whoami.exe

- name: run a cmd command
  win_command: cmd.exe /c mkdir C:\temp

- name: run a vbs script
  win_command: cscript.exe script.vbs
```

---

: Some commands like `mkdir`, `del`, and `copy` only exist in the CMD shell. To run them with `win_command` they must be prefixed with `cmd.exe /c`.

---

## Argument Rules

When running a command through `win_command`, the standard Windows argument rules apply:

- Each argument is delimited by a white space, which can either be a space or a tab.
- An argument can be surrounded by double quotes ". Anything inside these quotes is interpreted as a single argument even if it contains whitespace.

- A double quote preceded by a backslash \ is interpreted as just a double quote " and not as an argument delimiter.
- Backslashes are interpreted literally unless it immediately preceeds double quotes; for example \ == \ and \" == "
- If an even number of backslashes is followed by a double quote, one backslash is used in the argument for every pair, and the double quote is used as a string delimiter for the argument.
- If an odd number of backslashes is followed by a double quote, one backslash is used in the argument for every pair, and the double quote is escaped and made a literal double quote in the argument.

With those rules in mind, here are some examples of quoting:

```
- win_command: C:\temp\executable.exe argument1 "argument 2" "C:\path\with space"
↳ "double \"quoted\""
```

```
argv[0] = C:\temp\executable.exe
argv[1] = argument1
argv[2] = argument 2
argv[3] = C:\path\with space
argv[4] = double "quoted"
```

```
- win_command: 'C:\Program Files\Program\program.exe' "escaped \\" backslash"
↳ unquoted-end-backslash\
```

```
argv[0] = C:\Program Files\Program\program.exe
argv[1] = escaped \" backslash
argv[2] = unquoted-end-backslash\
```

*# due to YAML and Ansible parsing '\"' must be written as '{% raw %}\\{% endraw %}''*

```
- win_command: C:\temp\executable.exe C:\no\space\path "arg with end \ before end"
↳ quote{% raw %}\\{% endraw %}"
```

```
argv[0] = C:\temp\executable.exe
argv[1] = C:\no\space\path
argv[2] = arg with end \ before end quote\"
```

For more information, see [escaping arguments](#).

## Creating and Running a Scheduled Task

WinRM has some restrictions in place that cause errors when running certain commands. One way to bypass these restrictions is to run a command through a scheduled task. A scheduled task is a Windows component that provides the ability to run an executable on a schedule and under a different account.

Ansible version 2.5 added modules that make it easier to work with scheduled tasks in Windows. The following is an example of running a script as a scheduled task that deletes itself after running:

```
- name: create scheduled task to run a process
  win_scheduled_task:
    name: adhoc-task
    username: SYSTEM
    actions:
      - path: PowerShell.exe
        arguments: |
          Start-Sleep -Seconds 30 # this isn't required, just here as a demonstration
          New-Item -Path C:\temp\test -ItemType Directory
```

(continues on next page)

()

```

# remove this action if the task shouldn't be deleted on completion
- path: cmd.exe
  arguments: /c schtasks.exe /Delete /TN "adhoc-task" /F
triggers:
- type: registration

- name: wait for the scheduled task to complete
  win_scheduled_task_stat:
    name: adhoc-task
    register: task_stat
    until: (task_stat.state is defined and task_stat.state.status != "TASK_STATE_RUNNING
↪") or (task_stat.task_exists == False)
    retries: 12
    delay: 10

```

---

: The modules used in the above example were updated/added in Ansible version 2.5.

---

## Path Formatting for Windows

Windows differs from a traditional POSIX operating system in many ways. One of the major changes is the shift from / as the path separator to \. This can cause major issues with how playbooks are written, since \ is often used as an escape character on POSIX systems.

Ansible allows two different styles of syntax; each deals with path separators for Windows differently:

### YAML Style

When using the YAML syntax for tasks, the rules are well-defined by the YAML standard:

- When using a normal string (without quotes), YAML will not consider the backslash an escape character.
- When using single quotes ', YAML will not consider the backslash an escape character.
- When using double quotes ", the backslash is considered an escape character and needs to be escaped with another backslash.

---

: You should only quote strings when it is absolutely necessary or required by YAML, and then use single quotes.

---

The YAML specification considers the following [escape sequences](#):

- \0, \\\, \", \\_, \a, \b, \e, \f, \n, \r, \t, \v, \L, \N and \P – Single character escape
- <TAB>, <SPACE>, <NBSP>, <LNSP>, <PSP> – Special characters
- \x.. – 2-digit hex escape
- \u.... – 4-digit hex escape
- \U..... – 8-digit hex escape

Here are some examples on how to write Windows paths:

```
GOOD
tempdir: C:\Windows\Temp

WORKS
tempdir: 'C:\Windows\Temp'
tempdir: "C:\\Windows\\Temp"

BAD, BUT SOMETIMES WORKS
tempdir: C:\\Windows\\Temp
tempdir: 'C:\\Windows\\Temp'
tempdir: C:/Windows/Temp

FAILS
tempdir: "C:\Windows\Temp"

---
# example of single quotes when they are required
- name: copy tomcat config
  win_copy:
    src: log4j.xml
    dest: '{{tc_home}}\lib\log4j.xml'
```

## Legacy key=value Style

The legacy key=value syntax is used on the command line for adhoc commands, or inside playbooks. The use of this style is discouraged within playbooks because backslash characters need to be escaped, making playbooks harder to read. The legacy syntax depends on the specific implementation in Ansible, and quoting (both single and double) does not have any effect on how it is parsed by Ansible.

The Ansible key=value parser `parse_kv()` considers the following escape sequences:

- `\, ', ", \a, \b, \f, \n, \r, \t` and `\v` – Single character escape
- `\x..` – 2-digit hex escape
- `\u....` – 4-digit hex escape
- `\U.....` – 8-digit hex escape
- `\N{...}` – Unicode character by name

This means that the backslash is an escape character for some sequences, and it is usually safer to escape a backslash when in this form.

Here are some examples of using Windows paths with the key=value style:

```
GOOD
tempdir=C:\\Windows\\Temp

WORKS
tempdir='C:\\Windows\\Temp'
tempdir="C:\\Windows\\Temp"

BAD, BUT SOMETIMES WORKS
tempdir=C:\Windows\Temp
tempdir='C:\Windows\Temp'
tempdir="C:\Windows\Temp"
tempdir=C:/Windows/Temp
```

(continues on next page)

()

```

FAILS
tempdir=C:\Windows\temp
tempdir='C:\Windows\temp'
tempdir="C:\Windows\temp"

```

The failing examples don't fail outright but will substitute \t with the <TAB> character resulting in tempdir being C:\Windows<TAB>emp.

## Limitations

Some things you cannot do with Ansible and Windows are:

- Upgrade PowerShell
- Interact with the WinRM listeners

Because WinRM is reliant on the services being online and running during normal operations, you cannot upgrade PowerShell or interact with WinRM listeners with Ansible. Both of these actions will cause the connection to fail. This can technically be avoided by using `async` or a scheduled task, but those methods are fragile if the process it runs breaks the underlying connection Ansible uses, and are best left to the bootstrapping process or before an image is created.

## Developing Windows Modules

Because Ansible modules for Windows are written in PowerShell, the development guides for Windows modules differ substantially from those for standard standard modules. Please see `developing_modules_general_windows` for more information.

:

**User Guide** The documentation index

**Working With Playbooks** An introduction to playbooks

**Best Practices** Best practices advice

**List of Windows Modules** Windows specific module list, all implemented in PowerShell

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Desired State Configuration

### Topics

- *Desired State Configuration*
  - *What is Desired State Configuration?*
  - *Host Requirements*
  - *Why Use DSC?*
  - *How to Use DSC?*

- \* *Property Types*
  - *PSCredential*
  - *CimInstance Type*
  - *Arrays*
- \* *Run As Another User*
- *Custom DSC Resources*
  - \* *Finding Custom DSC Resources*
  - \* *Installing a Custom Resource*
- *Examples*
  - \* *Extract a zip file*
  - \* *Create a directory*
  - \* *Interact with Azure*
  - \* *Setup IIS Website*

## What is Desired State Configuration?

Desired State Configuration, or DSC, is a tool built into PowerShell that can be used to define a Windows host setup through code. The overall purpose of DSC is the same as Ansible, it is just executed in a different manner. Since Ansible 2.4, the `win_dsc` module has been added and can be used to leverage existing DSC resources when interacting with a Windows host.

More details on DSC can be viewed at [DSC Overview](#).

## Host Requirements

To use the `win_dsc` module, a Windows host must have PowerShell v5.0 or newer installed. All supported hosts, except for Windows Server 2008 (non R2) can be upgraded to PowerShell v5.

Once the PowerShell requirements have been met, using DSC is as simple as creating a task with the `win_dsc` module.

## Why Use DSC?

DSC and Ansible modules have a common goal which is to define and ensure the state of a resource. Because of this, resources like the DSC [File resource](#) and Ansible `win_file` can be used to achieve the same result. Deciding which to use depends on the scenario.

Reasons for using an Ansible module over a DSC resource:

- The host does not support PowerShell v5.0, or it cannot easily be upgraded
- The DSC resource does not offer a feature present in an Ansible module. For example `win_regedit` can manage the `REG_NONE` property type, while the DSC `Registry` resource cannot
- DSC resources have limited check mode support, while some Ansible modules have better checks
- DSC resources do not support diff mode, while some Ansible modules do



- Custom resources require further installation steps to be run on the host beforehand, while Ansible modules are in built-in to Ansible
- There are bugs in a DSC resource where an Ansible module works

Reasons for using a DSC resource over an Ansible module:

- The Ansible module does not support a feature present in a DSC resource
- There is no Ansible module available
- There are bugs in an existing Ansible module

In the end, it doesn't matter whether the task is performed with DSC or an Ansible module; what matters is that the task is performed correctly and the playbooks are still readable. If you have more experience with DSC over Ansible and it does the job, just use DSC for that task.

## How to Use DSC?

The `win_dsc` module takes in a free-form of options so that it changes according to the resource it is managing. A list of built in resources can be found at [resources](#).

Using the the [Registry](#) resource as an example, this is the DSC definition as documented by Microsoft:

```
Registry [string] #ResourceName
{
    Key = [string]
    ValueName = [string]
    [ Ensure = [string] { Enable | Disable } ]
    [ Force = [bool] ]
    [ Hex = [bool] ]
    [ DependsOn = [string[]] ]
    [ ValueData = [string[]] ]
    [ ValueType = [string] { Binary | Dword | ExpandString | MultiString | Qword | _
↳String } ]
}
```

When defining the task, `resource_name` must be set to the DSC resource being used - in this case the `resource_name` should be set to `Registry`. The `module_version` can refer to a specific version of the DSC resource installed; if left blank it will default to the latest version. The other options are parameters that are used to define the resource, such as `Key` and `ValueName`. While the options in the task are not case sensitive, keeping the case as-is is recommended because it makes it easier to distinguish DSC resource options from Ansible's `win_dsc` options.

This is what the Ansible task version of the above DSC Registry resource would look like:

```
- name: use win_dsc module with the Registry DSC resource
  win_dsc:
    resource_name: Registry
    Ensure: Present
    Key: HKEY_LOCAL_MACHINE\SOFTWARE\ExampleKey
    ValueName: TestValue
    ValueData: TestData
```

## Property Types

Each DSC resource property has a type that is associated with it. Ansible will try to convert the defined options to the correct type during execution. For simple types like `[string]` and `[bool]` this is a simple operation, but complex

types like [PSCredential] or arrays (like [string[]]) this require certain rules.

## PSCredential

A [PSCredential] object is used to store credentials in a secure way, but Ansible has no way to serialize this over JSON. To set a DSC PSCredential property, the definition of that parameter should have two entries that are suffixed with `_username` and `_password` for the username and password respectively. For example:

```
PsDscRunAsCredential_username: '{{ansible_user}}'
PsDscRunAsCredential_password: '{{ansible_password}}'

SourceCredential_username: AdminUser
SourceCredential_password: PasswordForAdminUser
```

---

: You should set `no_log: true` on the task definition in Ansible to ensure any credentials used are not stored in any log file or console output.

---

## CimInstance Type

A [CimInstance] object is used by DSC to store a dictionary object based on a custom class defined by that resource. Defining a value that takes in a [CimInstance] in YAML is the same as defining a dictionary in YAML. For example, to define a [CimInstance] value in Ansible:

```
# [CimInstance]AuthenticationInfo == MSFT_xWebAuthenticationInformation
AuthenticationInfo:
  Anonymous: no
  Basic: yes
  Digest: no
  Windows: yes
```

In the above example, the CIM instance is a representation of the class `MSFT_xWebAuthenticationInformation` <[https://github.com/PowerShell/xWebAdministration/blob/dev/DSCResources/MSFT\\_xWebsite/MSFT\\_xWebsite.schema.mof](https://github.com/PowerShell/xWebAdministration/blob/dev/DSCResources/MSFT_xWebsite/MSFT_xWebsite.schema.mof)>`. This class accepts four boolean variables, ``Anonymous, Basic, Digest, and Windows. The keys to use in a [CimInstance] depend on the class it represents. Please read through the documentation of the resource to determine the keys that can be used and the types of each key value. The class definition is typically located in the `<resource name>.schema.mof`.

## Arrays

Simple type arrays like [string[]] or [UInt32[]] are defined as a list or as a comma separated string which are then cast to their type. Using a list is recommended because the values are not manually parsed by the `win_dsc` module before being passed to the DSC engine. For example, to define a simple type array in Ansible:

```
# [string[]]
ValueData: entry1, entry2, entry3
ValueData:
- entry1
- entry2
- entry3
```

(continues on next page)

()

```
# [UInt32[]]
ReturnCode: 0,3010
ReturnCode:
- 0
- 3010
```

Complex type arrays like `[CimInstance[]]` (array of dicts), can be defined like this example:

```
# [CimInstance[]]BindingInfo == MSFT_xWebBindingInformation
BindingInfo:
- Protocol: https
  Port: 443
  CertificateStoreName: My
  CertificateThumbprint: C676A89018C4D5902353545343634F35E6B3A659
  HostName: DSCTest
  IPAddress: '*'
  SSLFlags: 1
- Protocol: http
  Port: 80
  IPAddress: '*'
```

The above example, is an array with two values of the class `MSFT_xWebBindingInformation` <[https://github.com/PowerShell/xWebAdministration/blob/dev/DSCResources/MSFT\\_xWebsite/MSFT\\_xWebsite.schema.mof](https://github.com/PowerShell/xWebAdministration/blob/dev/DSCResources/MSFT_xWebsite/MSFT_xWebsite.schema.mof)>`. When defining a ``[CimInstance[]], be sure to read the resource documentation to find out what keys to use in the definition.

## Run As Another User

By default, DSC runs each resource as the `SYSTEM` account and not the account that Ansible use to run the module. This means that resources that are dynamically loaded based on a user profile, like the `HKEY_CURRENT_USER` registry hive, will be loaded under the `SYSTEM` profile. The parameter `PsDscRunAsCredential` is a parameter that can be set for every DSC resource force the DSC engine to run under a different account. As `PsDscRunAsCredential` has a type of `PSCredential`, it is defined with the `_username` and `_password` suffix.

Using the Registry resource type as an example, this is how to define a task to access the `HKEY_CURRENT_USER` hive of the Ansible user:

```
- name: use win_dsc with PsDscRunAsCredential to run as a different user
  win_dsc:
    resource_name: Registry
    Ensure: Present
    Key: HKEY_CURRENT_USER\ExampleKey
    ValueName: TestValue
    ValueData: TestData
    PsDscRunAsCredential_username: '{{ansible_user}}'
    PsDscRunAsCredential_password: '{{ansible_password}}'
  no_log: true
```

## Custom DSC Resources

DSC resources are not limited to the built-in options from Microsoft. Custom modules can be installed to manage other resources that are not usually available.

## Finding Custom DSC Resources

You can use the [PSGallery](#) to find custom resources, along with documentation on how to install them on a Windows host.

The `Find-DscResource` cmdlet can also be used to find custom resources. For example:

```
# find all DSC resources in the configured repositories
Find-DscResource

# find all DSC resources that relate to SQL
Find-DscResource -ModuleName "*sql*"
```

---

: DSC resources developed by Microsoft that start with `x`, means the resource is experimental and comes with no support.

---

## Installing a Custom Resource

There are three ways that a DSC resource can be installed on a host:

- Manually with the `Install-Module` cmdlet
- Using the `win_psmodule` Ansible module
- Saving the module manually and copying it another host

This is an example of installing the `xWebAdministration` resources using `win_psmodule`:

```
- name: install xWebAdministration DSC resource
  win_psmodule:
    name: xWebAdministration
    state: present
```

Once installed, the `win_dsc` module will be able to use the resource by referencing it with the `resource_name` option.

The first two methods above only work when the host has access to the internet. When a host does not have internet access, the module must first be installed using the methods above on another host with internet access and then copied across. To save a module to a local filepath, the following PowerShell cmdlet can be run:

```
Save-Module -Name xWebAdministration -Path C:\temp
```

This will create a folder called `xWebAdministration` in `C:\temp` which can be copied to any host. For PowerShell to see this offline resource, it must be copied to a directory set in the `PSModulePath` environment variable. In most cases the path `C:\Program Files\WindowsPowerShell\Module` is set through this variable, but the `win_path` module can be used to add different paths.

## Examples

### Extract a zip file

```
- name: extract a zip file
  win_dsc:
    resource_name: Archive
    Destination: c:\temp\output
    Path: C:\temp\zip.zip
    Ensure: Present
```

## Create a directory

```
- name: create file with some text
  win_dsc:
    resource_name: File
    DestinationPath: C:\temp\file
    Contents: |
      Hello
      World
    Ensure: Present
    Type: File

- name: create directory that is hidden is set with the System attribute
  win_dsc:
    resource_name: File
    DestinationPath: C:\temp\hidden-directory
    Attributes: Hidden,System
    Ensure: Present
    Type: Directory
```

## Interact with Azure

```
- name: install xAzure DSC resources
  win_psmodule:
    name: xAzure
    state: present

- name: create virtual machine in Azure
  win_dsc:
    resource_name: xAzureVM
    ImageName: a699494373c04fc0bc8f2bb1389d6106__Windows-Server-2012-R2-201409.01-en.
    ↪us-127GB.vhd
    Name: DSCHOST01
    ServiceName: ServiceName
    StorageAccountName: StorageAccountName
    InstanceSize: Medium
    Windows: True
    Ensure: Present
    Credential_username: '{{ansible_user}}'
    Credential_password: '{{ansible_password}}'
```

## Setup IIS Website

```
- name: install xWebAdministration module
  win_psmodule:
    name: xWebAdministration
    state: present

- name: install IIS features that are required
  win_dsc:
    resource_name: WindowsFeature
    Name: '{{item}}'
    Ensure: Present
  with_items:
    - Web-Server
    - Web-Asp-Net45

- name: setup web content
  win_dsc:
    resource_name: File
    DestinationPath: C:\inetpub\IISSite\index.html
    Type: File
    Contents: |
      <html>
      <head><title>IIS Site</title></head>
      <body>This is the body</body>
      </html>
    Ensure: present

- name: create new website
  win_dsc:
    resource_name: xWebsite
    Name: NewIISSite
    State: Started
    PhysicalPath: C:\inetpub\IISSite\index.html
    BindingInfo:
      - Protocol: https
        Port: 8443
        CertificateStoreName: My
        CertificateThumbprint: C676A89018C4D5902353545343634F35E6B3A659
        HostName: DSCTest
        IPAddress: '*'
        SSLFlags: 1
      - Protocol: http
        Port: 8080
        IPAddress: '*'
    AuthenticationInfo:
      Anonymous: no
      Basic: yes
      Digest: no
      Windows: yes
```

:

**User Guide** The documentation index

**Working With Playbooks** An introduction to playbooks

**Best Practices** Best practices advice

**List of Windows Modules** :ref:‘<windows\_modules> Windows specific module list, all implemented in PowerShell

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## Windows Frequently Asked Questions

Here are some commonly asked questions in regards to Ansible and Windows and their answers.

---

: This document covers questions about managing Microsoft Windows servers with Ansible. For questions about Ansible Core, please see the [FAQ page](#).

---

### Does Ansible work with Windows XP or Server 2003?

Ansible does not support managing Windows XP or Server 2003 hosts. The supported operating system versions are:

- Windows Server 2008
- Windows Server 2008 R2
- Windows Server 2012
- Windows Server 2012 R2
- Windows Server 2016
- Windows 7
- Windows 8.1
- Windows 10

Ansible also has minimum PowerShell version requirements - please see [Setting up a Windows Host](#) for the latest information.

### Can I Manage Windows Nano Server?

Windows Nano Server is not currently supported by Ansible, since it does not have access to the full .NET Framework that is used by the majority of the modules and internal components.

### Can Ansible run on Windows?

No, Ansible cannot run on a Windows host and can only manage Windows hosts, but Ansible can be run under the Windows Subsystem for Linux (WSL).

---

: The Windows Subsystem for Linux is not supported by Microsoft or Ansible and should not be used for production systems.

---

To install Ansible on WSL, the following commands can be run in the bash terminal:

```
sudo apt-get update
sudo apt-get install python-pip git libffi-dev libssl-dev -y
pip install ansible pywinrm
```

To run Ansible from source instead of a release on the WSL, simply uninstall the pip installed version and then clone the git repo.

```
pip uninstall ansible -y
git clone https://github.com/ansible/ansible.git
source ansible/hacking/env-setup

# to enable Ansible on login, run the following
echo ". ~/ansible/hacking/env-setup -q" >> ~/.bashrc
```

### Can I use SSH keys to authenticate?

Windows uses WinRM as the transport protocol. WinRM supports a wide range of authentication options. The closest option to SSH keys is to use the certificate authentication option which maps an X509 certificate to a local user.

The way that these certificates are generated and mapped to a user is different from the SSH implementation; consult the [Windows Remote Management](#) documentation for more information.

### Why can I run a command locally that does not work under Ansible?

Ansible executes commands through WinRM. These processes are different from running a command locally in these ways:

- Unless using an authentication option like CredSSP or Kerberos with credential delegation, the WinRM process does not have the ability to delegate the user's credentials to a network resource, causing `Access is Denied` errors.
- All processes run under WinRM are in a non-interactive session. Applications that require an interactive session will not work.
- When running through WinRM, Windows restricts access to internal Windows APIs like the Windows Update API and DPAPI, which some installers and programs rely on.

Some ways to bypass these restrictions are to:

- Use `become`, which runs a command as it would when run locally. This will bypass most WinRM restrictions, as Windows is unaware the process is running under WinRM when `become` is used. See the [Understanding Privilege Escalation](#) documentation for more information.
- Use a scheduled task, which can be created with `win_scheduled_task`. Like `become`, it will bypass all WinRM restrictions, but it can only be used to run commands, not modules.
- Use `win_psexec` to run a command on the host. PSEXec does not use WinRM and so will bypass any of the restrictions.
- To access network resources without any of these workarounds, an authentication option that supports credential delegation can be used. Both CredSSP and Kerberos with credential delegation enabled can support this.

See [Understanding Privilege Escalation](#) more info on how to use `become`. The limitations section at [Windows Remote Management](#) has more details around WinRM limitations.



## This program won't install with Ansible

See [this question](#) for more information about WinRM limitations.

## What modules are available?

Most of the Ansible modules in Ansible Core are written for a combination of Linux/Unix machines and arbitrary web services. These modules are written in Python and most of them do not work on Windows.

Because of this, there are dedicated Windows modules that are written in PowerShell and are meant to be run on Windows hosts. A list of these modules can be found [here](#).

In addition, the following Ansible Core modules/action-plugins work with Windows:

- `add_host`
- `assert`
- `async_status`
- `debug`
- `fail`
- `fetch`
- `group_by`
- `include`
- `include_role`
- `include_vars`
- `meta`
- `pause`
- `raw`
- `script`
- `set_fact`
- `set_stats`
- `setup`
- `slurp`
- `template` (also: `win_template`)
- `wait_for_connection`

## Can I run Python modules?

No, the WinRM connection protocol is set to use PowerShell modules, so Python modules will not work. A way to bypass this issue is to use `delegate_to: localhost` to run a Python module on the Ansible controller. This is useful if during a playbook, an external service needs to be contacted and there is no equivalent Windows module available.

## Can I connect over SSH?

Microsoft has announced and is developing a fork of OpenSSH for Windows that allows remote manage of Windows servers through the SSH protocol instead of WinRM. While this can be installed and used right now for normal SSH clients, it is still in beta from Microsoft and the required functionality has not been developed within Ansible yet.

There are future plans on adding this feature and this page will be updated once more information can be shared.

## Why is connecting to the host via ssh failing?

When trying to connect to a Windows host and the output error indicates that SSH was used, then this is an indication that the connection vars are not set properly or the host is not inheriting them correctly.

Make sure `ansible_connection: winrm` is set in the inventory for the Windows host.

## Why are my credentials are being rejected?

This can be due to a myriad of reasons unrelated to incorrect credentials.

See HTTP 401/Credentials Rejected at [Setting up a Windows Host](#) for a more detailed guide of this could mean.

## Why am I getting an error SSL CERTIFICATE\_VERIFY\_FAILED?

When the Ansible controller is running on Python 2.7.9+ or an older version of Python that has backported SSLContext (like Python 2.7.5 on RHEL 7), the controller will attempt to validate the certificate WinRM is using for an HTTPS connection. If the certificate cannot be validated (such as in the case of a self signed cert), it will fail the verification process.

To ignore certificate validation, add `ansible_winrm_server_cert_validation: ignore` to inventory for the Windows host.

:

**User Guide** The documentation index

**Windows Guides** The Windows documentation index

**Working With Playbooks** An introduction to playbooks

**Best Practices** Best practices advice

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## 1.5 Ansible Community Guide

Welcome to the Ansible Community Guide!

The purpose of this guide is to teach you everything you need to know about being a contributing member of the Ansible community.

To get started, please read and understand the [Community Code of Conduct](#), and then select one of the following topics.

## 1.5.1 Community Code of Conduct

### Topics

- *Community Code of Conduct*
  - *Anti-harassment policy*
  - *Policy violations*

Every community can be strengthened by a diverse variety of viewpoints, insights, opinions, skillsets, and skill levels. However, with diversity comes the potential for disagreement and miscommunication. The purpose of this Code of Conduct is to ensure that disagreements and differences of opinion are conducted respectfully and on their own merits, without personal attacks or other behavior that might create an unsafe or unwelcoming environment.

These policies are not designed to be a comprehensive set of Things You Cannot Do. We ask that you treat your fellow community members with respect and courtesy, and in general, Don't Be A Jerk. This Code of Conduct is meant to be followed in spirit as much as in letter and is not exhaustive.

All Ansible events and participants therein are governed by this Code of Conduct and anti-harassment policy. We expect organizers to enforce these guidelines throughout all events, and we expect attendees, speakers, sponsors, and volunteers to help ensure a safe environment for our whole community. Specifically, this Code of Conduct covers participation in all Ansible-related forums and mailing lists, code and documentation contributions, public IRC channels, private correspondence, and public meetings.

Ansible community members are...

### Considerate

Contributions of every kind have far-ranging consequences. Just as your work depends on the work of others, decisions you make surrounding your contributions to the Ansible community will affect your fellow community members. You are strongly encouraged to take those consequences into account while making decisions.

### Patient

Asynchronous communication can come with its own frustrations, even in the most responsive of communities. Please remember that our community is largely built on volunteered time, and that questions, contributions, and requests for support may take some time to receive a response. Repeated "bumps" or "reminders" in rapid succession are not good displays of patience. Additionally, it is considered poor manners to ping a specific person with general questions. Pose your question to the community as a whole, and wait patiently for a response.

### Respectful

Every community inevitably has disagreements, but remember that it is possible to disagree respectfully and courteously. Disagreements are never an excuse for rudeness, hostility, threatening behavior, abuse (verbal or physical), or personal attacks.

### Kind

Everyone should feel welcome in the Ansible community, regardless of their background. Please be courteous, respectful and polite to fellow community members. Do not make or post offensive comments related to skill level, gender, gender identity or expression, sexual orientation, disability, physical appearance, body size, race, or religion. Sexualized images or imagery, real or implied violence, intimidation, oppression, stalking, sustained disruption of activities, publishing the personal information of others without explicit permission to do so, unwanted physical contact, and unwelcome sexual attention are all strictly prohibited. Additionally, you are encouraged not to make assumptions about the background or identity of your fellow community members.

### Inquisitive

The only stupid question is the one that does not get asked. We encourage our users to ask early and ask often. Rather than asking whether you can ask a question (the answer is always yes!), instead, simply ask your question. You are encouraged to provide as many specifics as possible. Code snippets in the form of Gists or other paste site links are almost always needed in order to get the most helpful answers. Refrain from pasting multiple lines of code directly into the IRC channels - instead use [gist.github.com](https://gist.github.com) or another paste site to provide code snippets.

### Helpful

The Ansible community is committed to being a welcoming environment for all users, regardless of skill level. We were all beginners once upon a time, and our community cannot grow without an environment where new users feel safe and comfortable asking questions. It can become frustrating to answer the same questions repeatedly; however, community members are expected to remain courteous and helpful to all users equally, regardless of skill or knowledge level. Avoid providing responses that prioritize snideness and snark over useful information. At the same time, everyone is expected to read the provided documentation thoroughly. We are happy to answer questions, provide strategic guidance, and suggest effective workflows, but we are not here to do your job for you.

### Anti-harassment policy

Harassment includes (but is not limited to) all of the following behaviors:

- Offensive comments related to gender (including gender expression and identity), age, sexual orientation, disability, physical appearance, body size, race, and religion
- Derogatory terminology including words commonly known to be slurs
- Posting sexualized images or imagery in public spaces
- Deliberate intimidation
- Stalking
- Posting others' personal information without explicit permission
- Sustained disruption of talks or other events
- Inappropriate physical contact
- Unwelcome sexual attention

Participants asked to stop any harassing behavior are expected to comply immediately. Sponsors are also subject to the anti-harassment policy. In particular, sponsors should not use sexualized images, activities, or other material. Meetup organizing staff and other volunteer organizers should not use sexualized attire or otherwise create a sexualized environment at community events.

In addition to the behaviors outlined above, continuing to behave a certain way after you have been asked to stop also constitutes harassment, even if that behavior is not specifically outlined in this policy. It is considerate and respectful to stop doing something after you have been asked to stop, and all community members are expected to comply with such requests immediately.

### Policy violations

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting [codeofconduct@ansible.com](mailto:codeofconduct@ansible.com), to any channel operator in the community IRC channels, or to the local organizers of an event. Meetup organizers are encouraged to prominently display points of contact for reporting unacceptable behavior at local events.

If a participant engages in harassing behavior, the meetup organizers may take any action they deem appropriate. These actions may include but are not limited to warning the offender, expelling the offender from the event, and barring the offender from future community events.

Organizers will be happy to help participants contact security or local law enforcement, provide escorts to an alternate location, or otherwise assist those experiencing harassment to feel safe for the duration of the meetup. We value the safety and well-being of our community members and want everyone to feel welcome at our events, both online and offline.

We expect all participants, organizers, speakers, and attendees to follow these policies at all of our event venues and event-related social events.

The Ansible Community Code of Conduct is licensed under the Creative Commons Attribution-Share Alike 3.0 license. Our Code of Conduct was adapted from Codes of Conduct of other open source projects, including:

- Contributor Covenant
- Elastic
- The Fedora Project
- OpenStack
- Puppet Labs
- Ubuntu

## 1.5.2 The Ansible Development Process

### Topics

- *The Ansible Development Process*
- *Road Maps*
- *Pull Requests*
  - *Backport Pull Request Process*
- *Ansibullbot*
  - *Overview*
  - *Community Maintainers*
  - *Workflow*
  - *PR Labels*
    - \* *Workflow Labels*
    - \* *Informational Labels*
    - \* *Special Labels*

This section discusses how the Ansible development and triage process works.

## 1.5.3 Road Maps

The Ansible Core team provides a road map for each upcoming release. These road maps can be found [here](#).

## 1.5.4 Pull Requests

Ansible accepts code via **pull requests** ("PRs" for short). GitHub provides a great overview of [how the pull request process works](#) in general.

Because Ansible receives many pull requests, we use an automated process to help us through the process of reviewing and merging pull requests. That process is managed by **Ansibullbot**.

### Backport Pull Request Process

After the pull request submitted to Ansible for the `devel` branch is accepted and merged, the following instructions will help you create a pull request to backport the change to a previous stable branch.

---

: These instructions assume that `stable-2.5` is the targeted release branch for the backport.

---

---

: These instructions assume that `https://github.com/ansible/ansible.git` is configured as a `git remote` named `upstream`. If you do not use a `git remote` named `upstream`, adjust the instructions accordingly.

---

---

: These instructions assume that `https://github.com/<yourgithubaccount>/ansible.git` is configured as a `git remote` named `origin`. If you do not use a `git remote` named `origin`, adjust the instructions accordingly.

---

1. Prepare your `devel`, `stable`, and feature branches:

```
git fetch upstream
git checkout -b backport/2.5/[PR_NUMBER_FROM_DEVEL] upstream/stable-2.5
```

2. Cherry pick the relevant commit SHA from the `devel` branch into your feature branch, handling merge conflicts as necessary:

```
git cherry-pick -x [SHA_FROM_DEVEL]
```

3. Add a changelog entry for the change, and commit it.

4. Push your feature branch to your fork on GitHub:

```
git push origin backport/2.5/[PR_NUMBER_FROM_DEVEL]
```

5. Submit the pull request for `backport/2.5/[PR_NUMBER_FROM_DEVEL]` against the `stable-2.5` branch

---

: The choice to use `backport/2.5/[PR_NUMBER_FROM_DEVEL]` as the name for the feature branch is somewhat arbitrary, but conveys meaning about the purpose of that branch. It is not required to use this format, but it can be helpful, especially when making multiple backport PRs for multiple stable branches.

---

### 1.5.5 Ansibullbot

#### Overview

Ansibullbot serves many functions:

- Responds quickly to PR submitters to thank them for submitting their PR
- Identifies the community maintainer responsible for reviewing PRs for any files affected
- Tracks the current status of PRs
- Pings responsible parties to remind them of any PR actions for which they may be responsible
- Provides maintainers with the ability to move PRs through the workflow
- Identifies PRs abandoned by their submitters so that we can close them
- Identifies modules abandoned by their maintainers so that we can find new maintainers

#### Community Maintainers

Each module has at least one assigned maintainer, listed in a [maintainer's file](#):

Some modules have no community maintainers assigned. In this case, the maintainer is listed as `$team_ansible`. Ultimately, it's our goal to have at least one community maintainer for every module.

The maintainer's job is to review PRs and decide whether that PR should be merged (`shipit`) or revised (`needs_revision`).

The ultimate goal of any pull request is to reach **shipit** status, where the Core team then decides whether the PR is ready to be merged. Not every PR that reaches the **shipit** label is actually ready to be merged, but the better our reviewers are, and the better our guidelines are, the more likely it will be that a PR that reaches **shipit** will be mergeable.

#### Workflow

Ansibullbot runs continuously. You can generally expect to see changes to your issue or pull request within thirty minutes. Ansibullbot examines every open pull request in the repositories, and enforces state roughly according to the following workflow:

- If a pull request has no workflow labels, it's considered **new**. Files in the pull request are identified, and the maintainers of those files are pinged by the bot, along with instructions on how to review the pull request. (Note: sometimes we strip labels from a pull request to "reboot" this process.)
- If the module maintainer is not `$team_ansible`, the pull request then goes into the **community\_review** state.
- If the module maintainer is `$team_ansible`, the pull request then goes into the **core\_review** state (and probably sits for a while).
- If the pull request is in **community\_review** and has received comments from the maintainer:
  - If the maintainer says `shipit`, the pull request is labeled **shipit**, whereupon the Core team assesses it for final merge.
  - If the maintainer says `needs_info`, the pull request is labeled **needs\_info** and the submitter is asked for more info.
  - If the maintainer says **needs\_revision**, the pull request is labeled **needs\_revision** and the submitter is asked to fix some things.

- If the submitter says `ready_for_review`, the pull request is put back into **community\_review** or **core\_review** and the maintainer is notified that the pull request is ready to be reviewed again.
- If the pull request is labeled **needs\_revision** or **needs\_info** and the submitter has not responded lately:
  - The submitter is first politely pinged after two weeks, pinged again after two more weeks and labeled **pending action**, and the issue or pull request will be closed two weeks after that.
  - If the submitter responds at all, the clock is reset.
- If the pull request is labeled **community\_review** and the reviewer has not responded lately:
  - The reviewer is first politely pinged after two weeks, pinged again after two more weeks and labeled **pending action**, and then may be reassigned to `$team_ansible` or labeled **core\_review**, or often the submitter of the pull request is asked to step up as a maintainer.
- If Shippable tests fail, or if the code is not able to be merged, the pull request is automatically put into **needs\_revision** along with a message to the submitter explaining why.

There are corner cases and frequent refinements, but this is the workflow in general.

### PR Labels

There are two types of PR Labels generally: *workflow labels* and *information labels*.

#### Workflow Labels

- **community\_review**: Pull requests for modules that are currently awaiting review by their maintainers in the Ansible community.
- **core\_review**: Pull requests for modules that are currently awaiting review by their maintainers on the Ansible Core team.
- **needs\_info**: Waiting on info from the submitter.
- **needs\_rebase**: Waiting on the submitter to rebase.
- **needs\_revision**: Waiting on the submitter to make changes.
- **shipit**: Waiting for final review by the core team for potential merge.

#### Informational Labels

- **backport**: this is applied automatically if the PR is requested against any branch that is not `devel`. The bot immediately assigns the labels `backport` and `core_review`.
- **bugfix\_pull\_request**: applied by the bot based on the templated description of the PR.
- **cloud**: applied by the bot based on the paths of the modified files.
- **docs\_pull\_request**: applied by the bot based on the templated description of the PR.
- **easyfix**: applied manually, inconsistently used but sometimes useful.
- **feature\_pull\_request**: applied by the bot based on the templated description of the PR.
- **networking**: applied by the bot based on the paths of the modified files.
- **owner\_pr**: largely deprecated. Formerly workflow, now informational. Originally, PRs submitted by the maintainer would automatically go to **shipit** based on this label. If the submitter is also a maintainer, we notify the other maintainers and still require one of the maintainers (including the submitter) to give a **shipit**.



- **pending\_action:** applied by the bot to PRs that are not moving. Reviewed every couple of weeks by the community team, who tries to figure out the appropriate action (closure, asking for new maintainers, etc).

## Special Labels

- **new\_plugin:** this is for new modules or plugins that are not yet in Ansible.

**Note:** *new\_plugin* kicks off a completely separate process, and frankly it doesn't work very well at present. We're working our best to improve this process.

## 1.5.6 Reporting Bugs And Requesting Features

### Topics

- *Reporting Bugs And Requesting Features*
  - *Reporting A Bug*
  - *Requesting a feature*

### Reporting A Bug

Ansible practices responsible disclosure - if this is a security related bug, email [security@ansible.com](mailto:security@ansible.com) instead of filing a ticket or posting to the Google Group and you will receive a prompt response.

Ansible bugs should be reported to [github.com/ansible/ansible/issues](https://github.com/ansible/ansible/issues) after signing up for a free GitHub account. Before reporting a bug, please use the bug/issue search to see if the issue has already been reported. This is listed on the bottom of the docs page for any module.

Knowing your Ansible version and the exact commands you are running, and what you expect, saves time and helps us help everyone with their issues more quickly. For that reason, we provide an issue template; please fill it out as completely and as accurately as possible.

Do not use the issue tracker for "how do I do this" type questions. These are great candidates for IRC or the mailing list instead where things are likely to be more of a discussion.

To be respectful of reviewers' time and allow us to help everyone efficiently, please provide minimal well-reduced and well-commented examples versus sharing your entire production playbook. Include playbook snippets and output where possible.

When sharing YAML in playbooks, formatting can be preserved by using [code blocks](#).

For multiple-file content, we encourage use of [gist.github.com](https://gist.github.com). Online pastebin content can expire, so it's nice to have things around for a longer term if they are referenced in a ticket.

If you are not sure if something is a bug yet, you are welcome to ask about something on the mailing list or IRC first.

As we are a very high volume project, if you determine that you do have a bug, please be sure to open the issue yourself to ensure we have a record of it. Don't rely on someone else in the community to file the bug report for you.

### Requesting a feature

The best way to get a feature into Ansible is to submit a pull request.

The next best way of getting a feature into Ansible is to submit a proposal through the [Ansible proposal process](#).

## 1.5.7 How To Help

### Topics

- *How To Help*
  - *Become a power user*
  - *Ask and answer questions online*
  - *Participate in your local meetup*
  - *File and verify issues*
  - *Review and submit pull requests*
  - *Become a module maintainer*
  - *Join a working group*
  - *Teach Ansible to others*
  - *Social media*

Thanks for being interested in helping the Ansible project!

There are many ways to help the Ansible project...but first, please read and understand the *Community Code of Conduct*.

### Become a power user

A great way to help the Ansible project is to become a power user:

- Use Ansible everywhere you can
- Take tutorials and classes
- Read the *official documentation*
- Study some of the [many excellent books](#) about Ansible
- [Get certified](#).

When you become a power user, your ability and opportunities to help the Ansible project in other ways will multiply quickly.

### Ask and answer questions online

There are many forums online where Ansible users ask and answer questions. Reach out and communicate with your fellow Ansible users.

You can find the official *Ansible communication channels*.

### Participate in your local meetup

There are Ansible meetups [all over the world](#). Join your local meetup. Attend regularly. Ask good questions. Volunteer to give a presentation about how you use Ansible.

If there isn't a meetup near you, we'll be happy to help you [start one](#).

## File and verify issues

All software has bugs, and Ansible is no exception. When you find a bug, you can help tremendously by *telling us about it*.

If you should discover that the bug you're trying to file already exists in an issue, you can help by verifying the behavior of the reported bug with a comment in that issue, or by reporting any additional information.

## Review and submit pull requests

As you become more familiar with how Ansible works, you may be able to fix issues or develop new features yourself. If you think you've got a solution to a bug you've found in Ansible, or if you've got a new feature that you've written and would like to share with millions of Ansible users, read all about the *Ansible development process* to learn how to get your code accepted into Ansible.

Another good way to help is to review pull requests that other Ansible users have submitted. The Ansible community keeps a full list of *open pull requests by file*, so if there's a particular module or plug-in that particularly interests you, you can easily keep track of all the relevant new pull requests and provide testing or feedback.

## Become a module maintainer

Once you've learned about the development process and have contributed code to a particular module, we encourage you to become a maintainer of that module. There are hundreds of different modules in Ansible, and the vast majority of them are written and maintained entirely by members of the Ansible community.

To learn more about the responsibilities of being an Ansible module maintainer, please read our *module maintainer guidelines*.

## Join a working group

Working groups are a way for Ansible community members to self-organize around particular topics of interest. We have working groups around various topics. To join or create a working group, please read the *Ansible working group guidelines*.

## Teach Ansible to others

We're working on a standardized Ansible workshop called *Lightbulb* that can provide a good hands-on introduction to Ansible usage and concepts.

## Social media

If you like Ansible and just want to spread the good word, feel free to share on your social media platform of choice, and let us know by using @ansible or #ansible. We'll be looking for you.

## 1.5.8 Module Maintainer Guidelines

### Topics

- *Module Maintainer Guidelines*

- *Maintainer Responsibilities*
- *Pull Requests, Issues, and Workflow*
  - \* *Pull Requests*
  - \* *Issues*
  - \* *PR Workflow*
  - \* *Maintainers (BOTMETA.yml)*
  - \* *Changing Maintainership*
- *Tools and other Resources*

Thank you for being a maintainer of part of Ansible’s codebase. This guide provides module maintainers an overview of their responsibilities, resources for additional information, and links to helpful tools.

In addition to the information below, module maintainers should be familiar with:

- *General Ansible community development practices*
- Documentation on *module development*

### Maintainer Responsibilities

When you contribute a new module to the [ansible/ansible](#) repository, you become the maintainer for that module once it has been merged. Maintainership empowers you with the authority to accept, reject, or request revisions to pull requests on your module – but as they say, "with great power comes great responsibility."

Maintainers of Ansible modules are expected to provide feedback, responses, or actions on pull requests or issues to the module(s) they maintain in a reasonably timely manner.

It is also recommended that you occasionally revisit the [contribution guidelines](#), as they are continually refined. Occasionally, you may be requested to update your module to move it closer to the general accepted standard requirements. We hope for this to be infrequent, and will always be a request with a fair amount of lead time (ie: not by tomorrow!).

Finally, following the [ansible-devel](#) mailing list can be a great way to participate in the broader Ansible community, and a place where you can influence the overall direction, quality, and goals of Ansible and its modules. If you’re not on this relatively low-volume list, please join us here: <https://groups.google.com/forum/#!forum/ansible-devel>

The Ansible community hopes that you will find that maintaining your module is as rewarding for you as having the module is for the wider community.

### Pull Requests, Issues, and Workflow

#### Pull Requests

Module pull requests are located in the [main Ansible repository](#).

Because of the high volume of pull requests, notification of PRs to specific modules are routed by an automated bot to the appropriate maintainer for handling. It is recommended that you set an appropriate notification process to receive notifications which mention your GitHub ID.

## Issues

Issues for modules, including bug reports, documentation bug reports, and feature requests, are tracked in the [ansible repository](#).

Issues for modules are routed to their maintainers via an automated process. This process is still being refined, and currently depends upon the issue creator to provide adequate details (specifically, providing the proper module name) in order to route it correctly. If you are a maintainer of a specific module, it is recommended that you periodically search module issues for issues which mention your module's name (or some variation on that name), as well as setting an appropriate notification process for receiving notification of mentions of your GitHub ID.

## PR Workflow

Automated routing of pull requests is handled by a tool called [Ansibot](#).

Being moderately familiar with how the workflow behind the bot operates can be helpful to you, and – should things go awry – your feedback can be helpful to the folks that continually help Ansibullbot to evolve.

A detailed explanation of the PR workflow can be seen in the *[The Ansible Development Process](#)*

## Maintainers (BOTMETA.yml)

The full list of maintainers is located in [BOTMETA.yml](#).

## Changing Maintainership

Communities change over time, and no one maintains a module forever. If you'd like to propose an additional maintainer for your module, please submit a PR to `BOTMETA.yml` with the GitHub username of the new maintainer.

If you'd like to step down as a maintainer, please submit a PR to the `BOTMETA.yml` removing your GitHub ID from the module in question. If that would leave the module with no maintainers, put "ansible" as the maintainer. This will indicate that the module is temporarily without a maintainer, and the Ansible community team will search for a new maintainer.

## Tools and other Resources

- [PRs in flight](#), organised by directory.
- Ansibullbot: <https://github.com/ansible/ansibullbot>
- *[The Ansible Development Process](#)*

## 1.5.9 Release Managers

### Topics

- *Release Managers*
  - *Pre-releases: What and Why*
  - \* *What is Beta?*

### \* *What is a Release Candidate?*

- *Release Process*

The release manager's purpose is to ensure a smooth release. To achieve that goal, they need to coordinate between:

- Developers with Commit privileges on the [Ansible github repository](#)
- Contributors without commit privileges
- The community
- Ansible documentation team
- Ansible Tower team

### Pre-releases: What and Why

Pre-releases exist to draw testers. They give people who don't feel comfortable running from source control a means to get an early version of the code to test and give us feedback. To ensure we get good feedback about a release, we need to make sure all major changes in a release are put into a pre-release. Testers must be given time to test those changes before the final release. Ideally we want there to be sufficient time between pre-releases for people to install and test one version for a span of time. Then they can spend more time using the new code than installing the latest version.

The right length of time for a tester is probably around two weeks. However, for our three-to-four month development cycle to work, we compress this down to one week; any less runs the risk of people spending more time installing the code instead of running it. However, if there's a time crunch (with a release date that cannot slip), it is better to release with new changes than to hold back those changes to give people time to test between. People cannot test what is not released, so we have to get those tarballs out there even if people feel they have to install more frequently.

### What is Beta?

In a Beta release, we know there are still bugs. We will continue to accept fixes for these. Although we review these fixes, sometimes they can be invasive or potentially destabilize other areas of the code.

During the beta, we will no longer accept feature submissions.

### What is a Release Candidate?

In a release candidate, we've fixed all known blockers. Any remaining bugfixes are ones that we are willing to leave out of the release. At this point we need user testing to determine if there are any other blocker bugs lurking.

Blocker bugs generally are those that cause significant problems for users. Regressions are more likely to be considered blockers because they will break present users' usage of Ansible.

The Release Manager will cherry-pick fixes for new release blockers. The release manager will also choose whether to accept bugfixes for isolated areas of the code or defer those to the next minor release. By themselves, non-blocker bugs will not trigger a new release; they will only make it into the next major release if blocker bugs require that a new release be made.

The last RC should be as close to the final as possible. The following things may be changed:

- Version numbers are changed automatically and will differ as the pre-release tags are removed from the versions.

- Tests and `docs/docsite/` can differ if really needed as they do not break runtime. However, the release manager may still reject them as they have the potential to cause breakage that will be visible during the release process.

---

: We want to specifically emphasize that code (in `bin/`, `lib/ansible/`, and `setup.py`) must be the same unless there are extraordinary extenuating circumstances. If there are extenuating circumstances, the Release Manager is responsible for notifying groups (like the Tower Team) which would want to test the code.

---

## 1.5.10 Release Process

The release process is kept in a [separate document](#) so that it can be easily updated during a release. If you need access to edit this, please ask one of the current release managers to add you.

## 1.5.11 Communicating

### Topics

- *Communicating*
  - *Code of Conduct*
  - *Mailing List Information*
  - *IRC Channel*
    - \* *General Channels*
    - \* *Working Group*
    - \* *Language specific channels*
    - \* *IRC Meetings*
  - *Tower Support Questions*

### Code of Conduct

Please read and understand the *Community Code of Conduct*.

### Mailing List Information

Ansible has several mailing lists. Your first post to the mailing list will be moderated (to reduce spam), so please allow up to a day or so for your first post to appear.

[Ansible Project List](#) is for sharing Ansible tips, answering questions, and general user discussion.

[Ansible Development List](#) is for learning how to develop on Ansible, asking about prospective feature design, or discussions about extending ansible or features in progress.

[Ansible Announce list](#) is a read-only list that shares information about new releases of Ansible, and also rare infrequent event information, such as announcements about an upcoming AnsibleFest, which is our official conference series.

[Ansible Container List](#) is for users and developers of the Ansible Container project.

[Ansible Lockdown List](#) is for all things related to Ansible Lockdown projects, including DISA STIG automation and CIS Benchmarks.

To subscribe to a group from a non-Google account, you can send an email to the subscription address requesting the subscription. For example: *ansible-devel+subscribe@googlegroups.com*

### IRC Channel

Ansible has several IRC channels on Freenode ([irc.freenode.net](http://irc.freenode.net)).

### General Channels

- `#ansible` - For general use questions and support.
- `#ansible-devel` - For discussions on developer topics and code related to features/bugs.
- `#ansible-meeting` - For public community meetings. We will generally announce these on one or more of the above mailing lists. See the [meeting schedule and agenda page](#)
- `#ansible-notice` - Mostly bot output from things like GitHub, etc.

### Working Group

- `#ansible-aws` - For discussions on Amazon Web Services.
- `#ansible-community` - Channel for discussing Ansible Community related things.
- `#ansible-container` - For discussions on Ansible Container.
- `#ansible-jboss` - Channel for discussing JBoss and Ansible related things.
- `#ansible-network` - Channel for discussing Network and Ansible related things.
- `#ansible-news` - Channel for discussing Ansible Communication & News related things.
- `#ansible-vmware` - For discussions on Ansible & VMware.
- `#ansible-windows` - For discussions on Ansible & Windows.

### Language specific channels

- `#ansible-es` - Channel for Spanish speaking Ansible community.
- `#ansible-fr` - Channel for French speaking Ansible community.

### IRC Meetings

The Ansible community holds regular IRC meetings on various topics, and anyone who is interested is invited to participate. For more information about Ansible meetings, consult the [meeting schedule and agenda page](#).



## Tower Support Questions

Ansible [Tower](#) is a UI, Server, and REST endpoint for Ansible.

If you have a question about Ansible Tower, visit [Red Hat support](#) rather than using the IRC channel or the general project mailing list.

## 1.5.12 Other Tools And Programs

The Ansible community provides several useful tools for working with the Ansible project. This is a list of some of the most popular of these tools.

- [PR by File](#) shows a current list of all open pull requests by individual file. An essential tool for Ansible module maintainers.
- [Ansible Lint](#) is a widely used, highly configurable best-practices linter for Ansible playbooks.
- [Ansible Review](#) is an extension of Ansible Lint designed for code review.
- [jctanner's Ansible Tools](#) is a miscellaneous collection of useful helper scripts for Ansible development.
- [Ansigenome](#) is a command line tool designed to help you manage your Ansible roles.
- [Awesome Ansible](#) is a collaboratively curated list of awesome Ansible resources.
- [Ansible Inventory Grapher](#) can be used to visually display inventory inheritance hierarchies and at what level a variable is defined in inventory.
- [Molecule](#) A testing framework for Ansible plays and roles.
- [ARA Records Ansible](#) ARA Records Ansible playbook runs and makes the recorded data available and intuitive for users and systems.

## 1.6 Developer Guide

Welcome to the Ansible Developer Guide!

The purpose of this guide is to document all of the paths available to you for interacting and shaping Ansible with code, ranging from developing modules and plugins to helping to develop the Ansible Core Engine via pull requests.

To get started, select one of the following topics.

### 1.6.1 Ansible Architecture

Ansible is a radically simple IT automation engine that automates cloud provisioning, configuration management, application deployment, intra-service orchestration, and many other IT needs.

Being designed for multi-tier deployments since day one, Ansible models your IT infrastructure by describing how all of your systems inter-relate, rather than just managing one system at a time.

It uses no agents and no additional custom security infrastructure, so it's easy to deploy - and most importantly, it uses a very simple language (YAML, in the form of Ansible Playbooks) that allow you to describe your automation jobs in a way that approaches plain English.

In this section, we'll give you a really quick overview of how Ansible works so you can see how the pieces fit together.

### Modules

Ansible works by connecting to your nodes and pushing out small programs, called "Ansible Modules" to them. These programs are written to be resource models of the desired state of the system. Ansible then executes these modules (over SSH by default), and removes them when finished.

Your library of modules can reside on any machine, and there are no servers, daemons, or databases required. Typically you'll work with your favorite terminal program, a text editor, and probably a version control system to keep track of changes to your content.

### Plugins

Plugins are pieces of code that augment Ansible's core functionality. Ansible ships with a number of handy plugins, and you can easily write your own.

### Inventory

By default, Ansible represents what machines it manages using a very simple INI file that puts all of your managed machines in groups of your own choosing.

To add new machines, there is no additional SSL signing server involved, so there's never any hassle deciding why a particular machine didn't get linked up due to obscure NTP or DNS issues.

If there's another source of truth in your infrastructure, Ansible can also plugin to that, such as drawing inventory, group, and variable information from sources like EC2, Rackspace, OpenStack, and more.

Here's what a plain text inventory file looks like:

```
---
[webservers]
www1.example.com
www2.example.com

[dbservers]
db0.example.com
db1.example.com
```

Once inventory hosts are listed, variables can be assigned to them in simple text files (in a subdirectory called 'group\_vars/' or 'host\_vars/' or directly in the inventory file.

Or, as already mentioned, use a dynamic inventory to pull your inventory from data sources like EC2, Rackspace, or OpenStack.

### Playbooks

Playbooks can finely orchestrate multiple slices of your infrastructure topology, with very detailed control over how many machines to tackle at a time. This is where Ansible starts to get most interesting.

Ansible's approach to orchestration is one of finely-tuned simplicity, as we believe your automation code should make perfect sense to you years down the road and there should be very little to remember about special syntax or features.

Here's what a simple playbook looks like:

```

---
- hosts: webservers
  serial: 5 # update 5 machines at a time
  roles:
    - common
    - webapp

- hosts: content_servers
  roles:
    - common
    - content

```

## Extending Ansible with Plug-ins and the API

Should you want to write your own, Ansible modules can be written in any language that can return JSON (Ruby, Python, bash, etc). Inventory can also plug in to any datasource by writing a program that speaks to that datasource and returns JSON. There's also various Python APIs for extending Ansible's connection types (SSH is not the only transport possible), callbacks (how Ansible logs, etc), and even for adding new server side behaviors.

### 1.6.2 Developing Modules

#### Topics

- *Developing Modules*
  - *Welcome*
  - *Should You Develop A Module?*
  - *How To Develop A Module*

#### Welcome

This section discusses how to develop, debug, review, and test modules.

Ansible modules are reusable, standalone scripts that can be used by the Ansible API, or by the **ansible** or **ansible-playbook** programs. They return information to ansible by printing a JSON string to stdout before exiting. They take arguments in one of several ways which we'll go into as we work through this tutorial.

See `all_modules` for a list of existing modules.

Modules can be written in any language and are found in the path specified by `ANSIBLE_LIBRARY` or the `--module-path` command line option or in the `library` section of the Ansible configuration file.

#### Should You Develop A Module?

Before diving into the work of creating a new module, you should think about whether you actually *should* develop a module. Ask the following questions:

1. Does a similar module already exist?

There are a lot of existing modules available, you should check out the list of existing modules at [Importing Modules](#)

### 2. Has someone already worked on a similar Pull Request?

It's possible that someone has already started developing a similar PR. There are a few ways to find open module Pull Requests:

- [GitHub new module PRs](#)
- [All updates to modules](#)
- [New module PRs listed by directory search for \*lib/ansible/modules/\*](#)

If you find an existing PR that looks like it addresses the issue you are trying to solve, please provide feedback on the PR - this will speed up getting the PR merged.

### 3. Should you use or develop an action plugin instead?

Action plugins get run on the master instead of on the target. For modules like `file/copy/template`, some of the work needs to be done on the master before the module executes on the target. Action plugins execute first on the master and can then execute the normal module on the target if necessary.

For more information about action plugins, read the [action plugins documentation](#).

### 4. Should you use a role instead?

Check out the [roles documentation](#).

### 5. Should you write multiple modules instead of one module?

The following guidelines will help you determine if your module attempts to do too much, and should likely be broken into several smaller modules.

- Modules should have a concise and well defined functionality. Basically, follow the UNIX philosophy of doing one thing well.
- Modules should not require that a user know all the underlying options of an api/tool to be used. For instance, if the legal values for a required module parameter cannot be documented, that's a sign that the module would be rejected.
- Modules should typically encompass much of the logic for interacting with a resource. A lightweight wrapper around an API that does not contain much logic would likely cause users to offload too much logic into a playbook, and for this reason the module would be rejected. Instead try creating multiple modules for interacting with smaller individual pieces of the API.

## How To Develop A Module

The following topics will discuss how to develop and work with modules:

[Ansible Module Architecture](#) A description of Ansible's module architecture.

**developing\_modules\_general** A general overview of how to develop, debug, and test modules.

**developing\_modules\_general\_windows** A general overview of how to develop, debug and test Windows modules.

**developing\_modules\_documenting** How to include in-line documentation in your module.

**developing\_modules\_best\_practices** Best practices, recommendations, and things to avoid.

**developing\_modules\_checklist** Checklist for contributing your module to Ansible.

[Testing Ansible](#) Developing unit and integration tests.

[Ansible and Python 3](#) Adding Python 3 support to modules (all new modules must be Python-2.6 and Python-3.5 compatible).

**developing\_modules\_in\_groups** A guide for partners wanting to submit multiple modules.

:

**all\_modules** Learn about available modules*Developing Plugins* Learn about developing plugins*Python API* Learn about the Python API for playbook and task execution**GitHub modules directory** Browse module source code**Mailing List** Development mailing list**irc.freenode.net** #ansible IRC chat channel

### 1.6.3 Ansible Module Architecture

This in-depth dive helps you understand Ansible's program flow to execute modules. It is written for people working on the portions of the Core Ansible Engine that execute a module. Those writing Ansible Modules may also find this in-depth dive to be of interest, but individuals simply using Ansible Modules will not likely find this to be helpful.

#### Types of Modules

Ansible supports several different types of modules in its code base. Some of these are for backwards compatibility and others are to enable flexibility.

#### Action Plugins

Action Plugins look like modules to end users who are writing *playbooks* but they're distinct entities for the purposes of this document. Action Plugins always execute on the controller and are sometimes able to do all work there (for instance, the `debug` Action Plugin which prints some text for the user to see or the `assert` Action Plugin which can test whether several values in a playbook satisfy certain criteria.)

More often, Action Plugins set up some values on the controller, then invoke an actual module on the managed node that does something with these values. An easy to understand version of this is the `template` Action Plugin. The `template` Action Plugin takes values from the user to construct a file in a temporary location on the controller using variables from the playbook environment. It then transfers the temporary file to a temporary file on the remote system. After that, it invokes the `copy` module which operates on the remote system to move the file into its final location, sets file permissions, and so on.

#### New-style Modules

All of the modules that ship with Ansible fall into this category.

New-style modules have the arguments to the module embedded inside of them in some manner. Non-new-style modules must copy a separate file over to the managed node, which is less efficient as it requires two over-the-wire connections instead of only one.

#### Python

New-style Python modules use the *Ansiballz* framework for constructing modules. All official modules (shipped with Ansible) use either this or the *powershell module framework*.

These modules use imports from `ansible.module_utils` in order to pull in boilerplate module code, such as argument parsing, formatting of return values as *JSON*, and various file operations.

: In Ansible, up to version 2.0.x, the official Python modules used the *Module Replacer* framework. For module authors, *Ansiballz* is largely a superset of *Module Replacer* functionality, so you usually do not need to know about one versus the other.

## Powershell

New-style powershell modules use the *Module Replacer* framework for constructing modules. These modules get a library of powershell code embedded in them before being sent to the managed node.

## JSONARGS

Scripts can arrange for an argument string to be placed within them by placing the string `<<INCLUDE_ANSIBLE_MODULE_JSON_ARGS>>` somewhere inside of the file. The module typically sets a variable to that value like this:

```
json_arguments = "<<INCLUDE_ANSIBLE_MODULE_JSON_ARGS>>"
```

Which is expanded as:

```
json_arguments = """{"param1": "test's quotes", "param2": "\"To be or not to be\" - ↪Hamlet"}"""
```

: Ansible outputs a *JSON* string with bare quotes. Double quotes are used to quote string values, double quotes inside of string values are backslash escaped, and single quotes may appear unescaped inside of a string value. To use JSONARGS, your scripting language must have a way to handle this type of string. The example uses Python's triple quoted strings to do this. Other scripting languages may have a similar quote character that won't be confused by any quotes in the JSON or it may allow you to define your own start-of-quote and end-of-quote characters. If the language doesn't give you any of these then you'll need to write a *non-native JSON module* or *Old-style module* instead.

The module typically parses the contents of `json_arguments` using a JSON library and then use them as native variables throughout the rest of its code.

## Non-native want JSON modules

If a module has the string `WANT_JSON` in it anywhere, Ansible treats it as a non-native module that accepts a filename as its only command line parameter. The filename is for a temporary file containing a *JSON* string containing the module's parameters. The module needs to open the file, read and parse the parameters, operate on the data, and print its return data as a JSON encoded dictionary to stdout before exiting.

These types of modules are self-contained entities. As of Ansible 2.1, Ansible only modifies them to change a shebang line if present.

$$\vdots$$

Examples of Non-native modules written in ruby are in the [Ansible for Rubyists](#) repository.

## Binary Modules

From Ansible 2.2 onwards, modules may also be small binary programs. Ansible doesn't perform any magic to make these portable to different systems so they may be specific to the system on which they were compiled or require other binary runtime dependencies. Despite these drawbacks, a site may sometimes have no choice but to compile a custom module against a specific binary library if that's the only way they have to get access to certain resources.

Binary modules take their arguments and will return data to Ansible in the same way as *want JSON modules*.

:

One example of a *binary module* written in go.

## Old-style Modules

Old-style modules are similar to *want JSON modules*, except that the file that they take contains `key=value` pairs for their parameters instead of *JSON*.

Ansible decides that a module is old-style when it doesn't have any of the markers that would show that it is one of the other types.

## How modules are executed

When a user uses **ansible** or **ansible-playbook**, they specify a task to execute. The task is usually the name of a module along with several parameters to be passed to the module. Ansible takes these values and processes them in various ways before they are finally executed on the remote machine.

### executor/task\_executor

The TaskExecutor receives the module name and parameters that were parsed from the *playbook* (or from the command line in the case of `/usr/bin/ansible`). It uses the name to decide whether it's looking at a module or an *Action Plugin*. If it's a module, it loads the *Normal Action Plugin* and passes the name, variables, and other information about the task and play to that Action Plugin for further processing.

## Normal Action Plugin

The `normal` Action Plugin executes the module on the remote host. It is the primary coordinator of much of the work to actually execute the module on the managed machine.

- It takes care of creating a connection to the managed machine by instantiating a `Connection` class according to the inventory configuration for that host.
- It adds any internal Ansible variables to the module's parameters (for instance, the ones that pass along `no_log` to the module).
- It takes care of creating any temporary files on the remote machine and cleans up afterwards.
- It does the actual work of pushing the module and module parameters to the remote host, although the *module\_common* code described in the next section does the work of deciding which format those will take.
- It handles any special cases regarding modules (for instance, various complications around Windows modules that must have the same names as Python modules, so that internal calling of modules from other Action Plugins work.)

Much of this functionality comes from the *BaseAction* class, which lives in `plugins/action/__init__.py`. It makes use of `Connection` and `Shell` objects to do its work.

---

: When *tasks* are run with the `async:` parameter, Ansible uses the `async` Action Plugin instead of the normal Action Plugin to invoke it. That program flow is currently not documented. Read the source for information on how that works.

---

## executor/module\_common.py

Code in `executor/module_common.py` takes care of assembling the module to be shipped to the managed node. The module is first read in, then examined to determine its type. *PowerShell* and *JSON-args modules* are passed through *Module Replacer*. New-style *Python modules* are assembled by *Ansiballz*. *Non-native-want-JSON*, *Binary modules*, and *Old-Style modules* aren't touched by either of these and pass through unchanged. After the assembling step, one final modification is made to all modules that have a shebang line. Ansible checks whether the interpreter in the shebang line has a specific path configured via an `ansible_${X}_interpreter` inventory variable. If it does, Ansible substitutes that path for the interpreter path given in the module. After this, Ansible returns the complete module data and the module type to the *Normal Action* which continues execution of the module.

Next we'll go into some details of the two assembler frameworks.

## Module Replacer

The Module Replacer framework is the original framework implementing new-style modules. It is essentially a preprocessor (like the C Preprocessor for those familiar with that programming language). It does straight substitutions of specific substring patterns in the module file. There are two types of substitutions:

- Replacements that only happen in the module file. These are public replacement strings that modules can utilize to get helpful boilerplate or access to arguments.
  - `from ansible.module_utils.MOD_LIB_NAME import *` is replaced with the contents of the `ansible/module_utils/MOD_LIB_NAME.py`. These should only be used with *new-style Python modules*.
  - `#<<INCLUDE_ANSIBLE_MODULE_COMMON>>` is equivalent to `from ansible.module_utils.basic import *` and should also only apply to new-style Python modules.
  - `# POWERSHELL_COMMON` substitutes the contents of `ansible/module_utils/powershell.ps1`. It should only be used with *new-style Powershell modules*.
- Replacements that are used by `ansible.module_utils` code. These are internal replacement patterns. They may be used internally, in the above public replacements, but shouldn't be used directly by modules.
  - `"<<ANSIBLE_VERSION>>"` is substituted with the Ansible version. In *new-style Python modules* under the *Ansiballz* framework the proper way is to instead instantiate an *AnsibleModule* and then access the version from `:attr:AnsibleModule.ansible_version`.
  - `"<<INCLUDE_ANSIBLE_MODULE_COMPLEX_ARGS>>"` is substituted with a string which is the Python `repr` of the *JSON* encoded module parameters. Using `repr` on the JSON string makes it safe to embed in a Python file. In new-style Python modules under the *Ansiballz* framework this is better accessed by instantiating an *AnsibleModule* and then using `AnsibleModule.params`.
  - `<<SELINUX_SPECIAL_FILESYSTEMS>>` substitutes a string which is a comma separated list of file systems which have a file system dependent security context in SELinux. In new-style Python modules, if you really need this you should instantiate an *AnsibleModule* and then use `AnsibleModule`.



`_selinux_special_fs`. The variable has also changed from a comma separated string of file system names to an actual python list of filesystem names.

- `<<INCLUDE_ANSIBLE_MODULE_JSON_ARGS>>` substitutes the module parameters as a JSON string. Care must be taken to properly quote the string as JSON data may contain quotes. This pattern is not substituted in new-style Python modules as they can get the module parameters another way.
- The string `syslog.LOG_USER` is replaced wherever it occurs with the `syslog_facility` which was named in `ansible.cfg` or any `ansible_syslog_facility` inventory variable that applies to this host. In new-style Python modules this has changed slightly. If you really need to access it, you should instantiate an *AnsibleModule* and then use `AnsibleModule._syslog_facility` to access it. It is no longer the actual syslog facility and is now the name of the syslog facility. See the [documentation on internal arguments](#) for details.

## Ansiballz

Ansible 2.1 switched from the *Module Replacer* framework to the Ansiballz framework for assembling modules. The Ansiballz framework differs from module replacer in that it uses real Python imports of things in `ansible/module_utils` instead of merely preprocessing the module. It does this by constructing a zipfile – which includes the module file, files in `ansible/module_utils` that are imported by the module, and some boilerplate to pass in the module’s parameters. The zipfile is then Base64 encoded and wrapped in a small Python script which decodes the Base64 encoding and places the zipfile into a temp directory on the managed node. It then extracts just the ansible module script from the zip file and places that in the temporary directory as well. Then it sets the `PYTHONPATH` to find python modules inside of the zip file and invokes **python** on the extracted ansible module.

---

: Ansible wraps the zipfile in the Python script for two reasons:

- for compatibility with Python-2.6 which has a less functional version of Python’s `-m` command line switch.
  - so that pipelining will function properly. Pipelining needs to pipe the Python module into the Python interpreter on the remote node. Python understands scripts on stdin but does not understand zip files.
- 

In Ansiballz, any imports of Python modules from the `ansible.module_utils` package trigger inclusion of that Python file into the zipfile. Instances of `#<<INCLUDE_ANSIBLE_MODULE_COMMON>>` in the module are turned into `from ansible.module_utils.basic import *` and `ansible/module-utils/basic.py` is then included in the zipfile. Files that are included from `module_utils` are themselves scanned for imports of other Python modules from `module_utils` to be included in the zipfile as well.

: At present, the Ansiballz Framework cannot determine whether an import should be included if it is a relative import. Always use an absolute import that has `ansible.module_utils` in it to allow Ansiballz to determine that the file should be included.

## Passing args

In *Module Replacer*, module arguments are turned into a JSON-ified string and substituted into the combined module file. In *Ansiballz*, the JSON-ified string is passed into the module via stdin. When a `ansible.module_utils.basic.AnsibleModule` is instantiated, it parses this string and places the args into `AnsibleModule.params` where it can be accessed by the module’s other code.

---

: Internally, the *AnsibleModule* uses the helper function, `ansible.module_utils.basic._load_params()`, to load the parameters from stdin and save them into an internal global variable. Very

---

dynamic custom modules which need to parse the parameters prior to instantiating an `AnsibleModule` may use `_load_params` to retrieve the parameters. Be aware that `_load_params` is an internal function and may change in breaking ways if necessary to support changes in the code. However, we'll do our best not to break it gratuitously, which is not something that can be said for either the way parameters are passed or the internal global variable.

---

### Internal arguments

Both *Module Replacer* and *Ansiballz* send additional arguments to the module beyond those which the user specified in the playbook. These additional arguments are internal parameters that help implement global Ansible features. Modules often do not need to know about these explicitly as the features are implemented in `ansible.module_utils.basic` but certain features need support from the module so it's good to know about them.

#### `_ansible_no_log`

This is a boolean. If it's `True` then the playbook specified `no_log` (in a task's parameters or as a play parameter). This automatically affects calls to `AnsibleModule.log()`. If a module implements its own logging then it needs to check this value. The best way to look at this is for the module to instantiate an *AnsibleModule* and then check the value of `AnsibleModule.no_log`.

---

: `no_log` specified in a module's `argument_spec` are handled by a different mechanism.

---

#### `_ansible_debug`

This is a boolean that turns on more verbose logging. If a module uses `AnsibleModule.debug()` rather than `AnsibleModule.log()` then the messages are only logged if this is `True`. This also turns on logging of external commands that the module executes. This can be changed via the `debug` setting in `ansible.cfg` or the environment variable `ANSIBLE_DEBUG`. If, for some reason, a module must access this, it should do so by instantiating an *AnsibleModule* and accessing `AnsibleModule._debug`.

#### `_ansible_diff`

This boolean is turned on via the `--diff` command line option. If a module supports it, it will tell the module to show a unified diff of changes to be made to templated files. The proper way for a module to access this is by instantiating an *AnsibleModule* and accessing `AnsibleModule._diff`.

#### `_ansible_verbosity`

This value could be used for finer grained control over logging. However, it is currently unused.

#### `_ansible_selinux_special_fs`

This is a list of names of filesystems which should have a special selinux context. They are used by the *AnsibleModule* methods which operate on files (changing attributes, moving, and copying). The list of names is set via a comma separated string of filesystem names from `ansible.cfg`:

```
# ansible.cfg
[selinux]
special_context_filesystems=nfs,vboxsf,fuse,ramfs
```

If a module cannot use the builtin `AnsibleModule` methods to manipulate files and needs to know about these special context filesystems, it should instantiate an `AnsibleModule` and then examine the list in `AnsibleModule._selinux_special_fs`.

This replaces `ansible.module_utils.basic.SELINUX_SPECIAL_FS` from *Module Replacer*. In module replacer it was a comma separated string of filesystem names. Under Ansiballz it's an actual list.

2.1 .

### `_ansible_syslog_facility`

This parameter controls which syslog facility ansible module logs to. It may be set by changing the `syslog_facility` value in `ansible.cfg`. Most modules should just use `AnsibleModule.log()` which will then make use of this. If a module has to use this on its own, it should instantiate an *AnsibleModule* and then retrieve the name of the syslog facility from `AnsibleModule._syslog_facility`. The code will look slightly different than it did under *Module Replacer* due to how hacky the old way was

```
# Old way
import syslog
syslog.openlog(NAME, 0, syslog.LOG_USER)

# New way
import syslog
facility_name = module._syslog_facility
facility = getattr(syslog, facility_name, syslog.LOG_USER)
syslog.openlog(NAME, 0, facility)
```

2.1 .

### `_ansible_version`

This parameter passes the version of ansible that runs the module. To access it, a module should instantiate an *AnsibleModule* and then retrieve it from `AnsibleModule.ansible_version`. This replaces `ansible.module_utils.basic.ANSIBLE_VERSION` from *Module Replacer*.

2.1 .

## Special Considerations

### Pipelining

Ansible can transfer a module to a remote machine in one of two ways:

- it can write out the module to a temporary file on the remote host and then use a second connection to the remote host to execute it with the interpreter that the module needs
- or it can use what's known as pipelining to execute the module by piping it into the remote interpreter's stdin.

Pipelining only works with modules written in Python at this time because Ansible only knows that Python supports this mode of operation. Supporting pipelining means that whatever format the module payload takes before being sent over the wire must be executable by Python via stdin.

### Why pass args over stdin?

Passing arguments via stdin was chosen for the following reasons:

- When combined with `ANSIBLE_PIPELINING`, this keeps the module's arguments from temporarily being saved onto disk on the remote machine. This makes it harder (but not impossible) for a malicious user on the remote machine to steal any sensitive information that may be present in the arguments.
- Command line arguments would be insecure as most systems allow unprivileged users to read the full commandline of a process.
- Environment variables are usually more secure than the commandline but some systems limit the total size of the environment. This could lead to truncation of the parameters if we hit that limit.

## AnsibleModule

### Argument Spec

The `argument_spec` provided to `AnsibleModule` defines the supported arguments for a module, as well as their type, defaults and more.

Example `argument_spec`:

```
module = AnsibleModule(argument_spec=dict(
    top_level=dict(
        type='dict',
        options=dict(
            second_level=dict(
                default=True,
                type='bool',
            )
        )
    )
))
```

This section will discuss the behavioral attributes for arguments

### type

`type` allows you to define the type of the value accepted for the argument. The default value for `type` is `str`. Possible values are:

- `str`
- `list`
- `dict`
- `bool`
- `int`
- `float`

- `path`
- `raw`
- `jsonarg`
- `json`
- `bytes`
- `bits`

The `raw` type, performs no type validation or type casing, and maintains the type of the passed value.

## elements

`elements` works in combination with `type` when `type='list'`. `elements` can then be defined as `elements='int'` or any other type, indicating that each element of the specified list should be of that type.

## default

The `default` option allows sets a default value for the argument for the scenario when the argument is not provided to the module. When not specified, the default value is `None`.

## fallback

`fallback` accepts a tuple where the first argument is a callable (function) that will be used to perform the lookup, based on the second argument. The second argument is a list of values to be accepted by the callable.

The most common callable used is `env_fallback` which will allow an argument to optionally use an environment variable when the argument is not supplied.

Example:

```
username=dict(fallback=(env_fallback, ['ANSIBLE_NET_USERNAME']))
```

## choices

`choices` accepts a list of choices that the argument will accept. The types of `choices` should match the `type`.

## required

`required` accepts a boolean, either `True` or `False` that indicates that the argument is required. This should not be used in combination with `default`.

## no\_log

`no_log` indicates that the value of the argument should not be logged or displayed.

## aliases

`aliases` accepts a list of alternative argument names for the argument, such as the case where the argument is `name` but the module accepts `aliases=[ 'pkg' ]` to allow `pkg` to be interchangeably with `name`

## options

`options` implements the ability to create a sub-argument\_spec, where the sub options of the top level argument are also validated using the attributes discussed in this section. The example at the top of this section demonstrates use of `options.type` or `elements` should be `dict` is this case.

## apply\_defaults

`apply_defaults` works alongside `options` and allows the default of the sub-options to be applied even when the top-level argument is not supplied.

In the example of the `argument_spec` at the top of this section, it would allow `module.params['top_level']['second_level']` to be defined, even if the user does not provide `top_level` when calling the module.

## removed\_in\_version

`removed_in_version` indicates which version of Ansible a deprecated argument will be removed in.

## 1.6.4 Appendix: Module Utilities

Ansible provides a number of module utilities that provide helper functions that you can use when developing your own modules. The *basic.py* module utility provides the main entry point for accessing the Ansible library, and all Ansible modules must, at minimum, import from *basic.py*:

```
from ansible.module_utils.basic import *
```

The following is a list of `module_utils` files and a general description. The module utility source code lives in the `./lib/module_utils` directory under your main Ansible path - for more details on any specific module utility, please see the source code.

- `api.py` - Adds shared support for generic API modules.
- `azure_rm_common.py` - Definitions and utilities for Microsoft Azure Resource Manager template deployments.
- `basic.py` - General definitions and helper utilities for Ansible modules.
- `cloudstack.py` - Utilities for CloudStack modules.
- `database.py` - Miscellaneous helper functions for PostGRES and MySQL
- `docker_common.py` - Definitions and helper utilities for modules working with Docker.
- `ec2.py` - Definitions and utilities for modules working with Amazon EC2
- `facts/` - Folder containing helper functions for modules that return facts. See <https://github.com/ansible/ansible/pull/23012> for more information.
- `gce.py` - Definitions and helper functions for modules that work with Google Compute Engine resources.
- `ismount.py` - Contains single helper function that fixes `os.path.ismount`

- `keycloak.py` - Definitions and helper functions for modules working with the Keycloak API
- `known_hosts.py` - utilities for working with `known_hosts` file
- `manageiq.py` - Functions and utilities for modules that work with ManageIQ platform and its resources.
- `memset.py` - Helper functions and utilities for interacting with Memset's API.
- `mysql.py` - Allows modules to connect to a MySQL instance
- `netapp.py` - Functions and utilities for modules that work with the NetApp storage platforms.
- `network/a10/a10.py` - Utilities used by the `a10_server` module to manage A10 Networks devices.
- `network/aci/aci.py` - Definitions and helper functions for modules that manage Cisco ACI Fabrics.
- `network/aireos/aireos.py` - Definitions and helper functions for modules that manage Cisco WLC devices.
- `network/aos/aos.py` - Module support utilities for managing Apstra AOS Server.
- `network/aruba/aruba.py` - Helper functions for modules working with Aruba networking devices.
- `network/asa/asa.py` - Module support utilities for managing Cisco ASA network devices.
- `network/avi/avi.py` - Helper functions for modules working with AVI networking devices.
- `network/bigswitch/bigswitch_utils.py` - Utilities used by the `bigswitch` module to manage Big Switch Networks devices.
- `network/cloudengine/ce.py` - Module support utilities for managing Huawei Cloudengine switch.
- `network/cnos/cnos.py` - Helper functions for modules working on devices running Lenovo CNOS.
- `network/common/config.py` - Configuration utility functions for use by networking modules
- `network/common/netconf.py` - Definitions and helper functions for modules that use Netconf transport.
- `network/common/parsing.py` - Definitions and helper functions for Network modules.
- `network/common/network.py` - Functions for running commands on networking devices
- `network/common/utils.py` - Defines commands and comparison operators and other utilises for use in networking modules
- `network/dellos6/dellos6.py` - Module support utilities for managing device running Dell OS6.
- `network/dellos9/dellos9.py` - Module support utilities for managing device running Dell OS9.
- `network/dellos10/dellos10.py` - Module support utilities for managing device running Dell OS10.
- `network/enos/enos.py` - Helper functions for modules working with Lenovo ENOS devices.
- `network/eos/eos.py` - Helper functions for modules working with EOS networking devices.
- `network/fortios/fortios.py` - Module support utilities for managing FortiOS devices.
- `network/ios/ios.py` - Definitions and helper functions for modules that manage Cisco IOS networking devices
- `network/iosxr/iosxr.py` - Definitions and helper functions for modules that manage Cisco IOS-XR networking devices.
- `network/ironware/ironware.py` - Module support utilities for managing Brocade IronWare devices.
- `network/junos/junos.py` - Definitions and helper functions for modules that manage Junos networking devices.
- `network/meraki/meraki.py` - Utilities specifically for the Meraki network modules.
- `network/netscaler/netscaler.py` - Utilities specifically for the netscaler network modules.
- `network/nso/nso.py` - Utilities for modules that work with Cisco NSO.

- `network/nxos/nxos.py` - Contains definitions and helper functions specific to Cisco NXOS networking devices.
- `network/onyx/onyx.py` - Definitions and helper functions for modules that manage Mellanox ONYX networking devices.
- `network/ordance/ordance.py` - Module support utilities for managing Ordnance devices.
- `network/sros/sros.py` - Helper functions for modules working with Open vSwitch bridges.
- `network/vyos/vyos.py` - Definitions and functions for working with VyOS networking
- `openstack.py` - Utilities for modules that work with Openstack instances.
- `openswitch.py` - Definitions and helper functions for modules that manage OpenSwitch devices
- `powershell.ps1` - Utilities for working with Microsoft Windows clients
- `pure.py` - Functions and utilities for modules that work with the Pure Storage storage platforms.
- `pycompat24.py` - Exception workaround for Python 2.4.
- `rax.py` - Definitions and helper functions for modules that work with Rackspace resources.
- `redhat.py` - Functions for modules that manage Red Hat Network registration and subscriptions
- `service.py` - Contains utilities to enable modules to work with Linux services (placeholder, not in use).
- `shell.py` - Functions to allow modules to create shells and work with shell commands
- `six/__init__.py` - Bundled copy of the [Six Python library](#) to aid in writing code compatible with both Python 2 and Python 3.
- `splitter.py` - String splitting and manipulation utilities for working with Jinja2 templates
- `urls.py` - Utilities for working with http and https requests
- `vca.py` - Contains utilities for modules that work with VMware vCloud Air
- `vmware.py` - Contains utilities for modules that work with VMware vSphere VMs

## 1.6.5 Developing Plugins

### Topics

- *Developing Plugins*
  - *General Guidelines*
    - \* *Raising Errors*
    - \* *String Encoding*
    - \* *Plugin Configuration*
  - *Callback Plugins*
  - *Connection Plugins*
  - *Inventory Plugins*
  - *Lookup Plugins*
  - *Vars Plugins*
  - *Filter Plugins*



- *Test Plugins*
- *Distributing Plugins*

Plugins are pieces of code that augment Ansible’s core functionality. Ansible ships with a number of handy plugins, and you can easily write your own. This section describes the various types of Ansible plugins and how to implement them.

## General Guidelines

This section lists some things that should apply to any type of plugin you develop.

## Raising Errors

In general, errors encountered during execution should be returned by raising `AnsibleError()` or similar class with a message describing the error. When wrapping other exceptions into error messages, you should always use the `to_text` Ansible function to ensure proper string compatibility across Python versions:

```
from ansible.module_utils._text import to_native

try:
    cause_an_exception()
except Exception as e:
    AnsibleError('Something happened, this was original exception: %s' % to_native(e))
```

Check the different `AnsibleError` objects and see which one applies the best to your situation.

## String Encoding

Any strings returned by your plugin that could ever contain non-ASCII characters must be converted into Python’s unicode type because the strings will be run through `jinja2`. To do this, you can use:

```
from ansible.module_utils._text import to_text
result_string = to_text(result_string)
```

## Plugin Configuration

Starting with Ansible version 2.4, we are unifying how each plugin type is configured and how they get those settings. Plugins will be able to declare their requirements and have Ansible provide them with a resolved configuration. Starting with Ansible 2.4 both callback and connection type plugins can use this system.

Most plugins will be able to use `self.get_option(<optionname>)` to access the settings. These are pre-populated by a `self.set_options()` call, for most plugin types this is done by the controller, but for some types you might need to do this explicitly. Of course, if you don’t have any configurable options, you can ignore this.

Plugins that support embedded documentation (see `ansible-doc` for the list) must now include well-formed doc strings to be considered for merge into the Ansible repo. This documentation also doubles as ‘configuration definition’ so they will never be out of sync.

If you inherit from a plugin, you must document the options it takes, either via a documentation fragment or as a copy.

## Callback Plugins

Callback plugins enable adding new behaviors to Ansible when responding to events. By default, callback plugins control most of the output you see when running the command line programs.

Callback plugins are created by creating a new class with the `Base(Callbacks)` class as the parent:

```
from ansible.plugins.callback import CallbackBase

class CallbackModule(CallbackBase):
    pass
```

From there, override the specific methods from the `CallbackBase` that you want to provide a callback for. For plugins intended for use with Ansible version 2.0 and later, you should only override methods that start with `v2`. For a complete list of methods that you can override, please see `__init__.py` in the `lib/ansible/plugins/callback` directory.

The following is a modified example of how Ansible's timer plugin is implemented, but with an extra option so you can see how configuration works in Ansible version 2.4 and later:

```
# Make coding more python3-ish, this is required for contributions to Ansible
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type

# not only visible to ansible-doc, it also 'declares' the options the plugin requires,
# and how to configure them.
DOCUMENTATION = '''
    callback: timer
    callback_type: aggregate
    requirements:
        - whitelist in configuration
    short_description: Adds time to play stats
    version_added: "2.0"
    description:
        - This callback just adds total play duration to the play stats.
    options:
        format_string:
            description: format of the string shown to user at play end
            ini:
                - section: callback_timer
                  key: format_string
            env:
                - name: ANSIBLE_CALLBACK_TIMER_FORMAT
            default: "Playbook run took %s days, %s hours, %s minutes, %s seconds"
'''

from datetime import datetime

from ansible.plugins.callback import CallbackBase

class CallbackModule(CallbackBase):
    """
    This callback module tells you how long your plays ran for.
    """
    CALLBACK_VERSION = 2.0
    CALLBACK_TYPE = 'aggregate'
    CALLBACK_NAME = 'timer'

    # only needed if you ship it and don't want to enable by default
```

(continues on next page)

()

```

CALLBACK_NEEDS_WHITELIST = True

def __init__(self):

    # make sure the expected objects are present, calling the base's __init__
    super(CallbackModule, self).__init__()

    # start the timer when the plugin is loaded, the first play should start a
    ↪ few milliseconds after.
    self.start_time = datetime.now()

    def _days_hours_minutes_seconds(self, runtime):
        ''' internal helper method for this callback '''
        minutes = (runtime.seconds // 60) % 60
        r_seconds = runtime.seconds - (minutes * 60)
        return runtime.days, runtime.seconds // 3600, minutes, r_seconds

    # this is only event we care about for display, when the play shows its summary
    ↪ stats; the rest are ignored by the base class
    def v2_playbook_on_stats(self, stats):
        end_time = datetime.now()
        runtime = end_time - self.start_time

        # Shows the usage of a config option declared in the DOCUMENTATION variable.
    ↪ Ansible will have set it when it loads the plugin.
        # Also note the use of the display object to print to screen. This is
    ↪ available to all callbacks, and you should use this over printing yourself
        self._display.display(self._plugin_options['format_string'] % (self._days_
    ↪ hours_minutes_seconds(runtime)))

```

Note that the `CALLBACK_VERSION` and `CALLBACK_NAME` definitions are required for properly functioning plugins for Ansible version 2.0 and later. `CALLBACK_TYPE` is mostly needed to distinguish 'stdout' plugins from the rest, since you can only load one plugin that writes to stdout.

## Connection Plugins

Connection plugins allow Ansible to connect to the target hosts so it can execute tasks on them. Ansible ships with many connection plugins, but only one can be used per host at a time.

By default, Ansible ships with several plugins. The most commonly used are the 'paramiko' SSH, native ssh (just called 'ssh'), and 'local' connection types. All of these can be used in playbooks and with `/usr/bin/ansible` to decide how you want to talk to remote machines.

The basics of these connection types are covered in the *Getting Started* section.

Should you want to extend Ansible to support other transports (SNMP, Message bus, etc) it's as simple as copying the format of one of the existing modules and dropping it into the connection plugins directory.

Ansible version 2.1 introduced the 'smart' connection plugin. The 'smart' connection type allows Ansible to automatically select either the 'paramiko' or 'openssh' connection plugin based on system capabilities, or the 'ssh' connection plugin if OpenSSH supports ControlPersist.

For examples on how to implement a connection plugin in, see the source code here: [lib/ansible/plugins/connection](#).

## Inventory Plugins

Inventory plugins were added in Ansible version 2.4. Inventory plugins parse inventory sources and form an in memory representation of the inventory.

You can see the details for inventory plugins in the *Developing Dynamic Inventory* page.

## Lookup Plugins

Lookup plugins are used to pull in data from external data stores. Lookup plugins can be used within playbooks both for looping — playbook language constructs like `with_fileglob` and `with_items` are implemented via lookup plugins — and to return values into a variable or parameter.

Lookup plugins are very flexible, allowing you to retrieve and return any type of data. When writing lookup plugins, always return data of a consistent type that can be easily consumed in a playbook. Avoid parameters that change the returned data type. If there is a need to return a single value sometimes and a complex dictionary other times, write two different lookup plugins.

Ansible includes many *filters* which can be used to manipulate the data returned by a lookup plugin. Sometimes it makes sense to do the filtering inside the lookup plugin, other times it is better to return results that can be filtered in the playbook. Keep in mind how the data will be referenced when determining the appropriate level of filtering to be done inside the lookup plugin.

Here's a simple lookup plugin implementation — this lookup returns the contents of a text file as a variable:

```
# python 3 headers, required if submitting to Ansible
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type

DOCUMENTATION = """
    lookup: file
    author: Daniel Hokka Zakrisson <daniel@hoxac.com>
    version_added: "0.9"
    short_description: read file contents
    description:
        - This lookup returns the contents from a file on the Ansible controller
        ↪'s file system.
    options:
        _terms:
            description: path(s) of files to read
            required: True
    notes:
        - if read in variable context, the file can be interpreted as YAML if the_
        ↪content is valid to the parser.
        - this lookup does not understand globing --- use the fileglob lookup_
        ↪instead.
    """
from ansible.errors import AnsibleError, AnsibleParserError
from ansible.plugins.lookup import LookupBase

try:
    from __main__ import display
except ImportError:
    from ansible.utils.display import Display
    display = Display()
```

(continues on next page)

()

```

class LookupModule(LookupBase):

    def run(self, terms, variables=None, **kwargs):

        # lookups in general are expected to both take a list as input and output a
        ↪ list
        # this is done so they work with the looping construct 'with_'.
        ret = []
        for term in terms:
            display.debug("File lookup term: %s" % term)

            # Find the file in the expected search path, using a class method
            # that implements the 'expected' search path for Ansible plugins.
            lookupfile = self.find_file_in_search_path(variables, 'files', term)

            # Don't use print or your own logging, the display class
            # takes care of it in a unified way.
            display.vvvv(u"File lookup using %s as file" % lookupfile)
            try:
                if lookupfile:
                    contents, show_data = self._loader._get_file_contents(lookupfile)
                    ret.append(contents.rstrip())
                else:
                    # Always use ansible error classes to throw 'final' exceptions,
                    # so the Ansible engine will know how to deal with them.
                    # The Parser error indicates invalid options passed
                    raise AnsibleParserError()
            except AnsibleParserError:
                raise AnsibleError("could not locate file in lookup: %s" % term)

        return ret

```

The following is an example of how this lookup is called:

```

---
- hosts: all
  vars:
    contents: "{{ lookup('file', '/etc/foo.txt') }}"

  tasks:

    - debug:
        msg: the value of foo.txt is {{ contents }} as seen today {{ lookup('pipe',
        ↪ 'date +%Y-%m-%d') }}

```

For more example lookup plugins, check out the source code for the lookup plugins that are included with Ansible here: [lib/ansible/plugins/lookup](#).

For more usage examples of lookup plugins, see [Using Lookups](#).

## Vars Plugins

Vars plugins inject additional variable data into Ansible runs that did not come from an inventory source, playbook, or command line. Playbook constructs like 'host\_vars' and 'group\_vars' work using vars plugins.

Vars plugins were partially implemented in Ansible 2.0 and rewritten to be fully implemented starting with Ansible 2.4.

Older plugins used a `run` method as their main body/work:

```
def run(self, name, vault_password=None):  
    pass # your code goes here
```

Ansible 2.0 did not pass passwords to older plugins, so vaults were unavailable. Most of the work now happens in the `get_vars` method which is called from the `VariableManager` when needed.

```
def get_vars(self, loader, path, entities):  
    pass # your code goes here
```

The parameters are:

- `loader`: Ansible's `DataLoader`. The `DataLoader` can read files, auto load JSON/YAML and decrypt vaulted data, and cache read files.
- `path`: this is 'directory data' for every inventory source and the current play's playbook directory, so they can search for data in reference to them. `get_vars` will be called at least once per available path.
- `entities`: these are host or group names that are pertinent to the variables needed. The plugin will get called once for hosts and again for groups.

This `get_vars` method just needs to return a dictionary structure with the variables.

Since Ansible version 2.4, vars plugins only execute as needed when preparing to execute a task. This avoids the costly 'always execute' behavior that occurred during inventory construction in older versions of Ansible.

For implementation examples of vars plugins, check out the source code for the vars plugins that are included with Ansible: [lib/ansible/plugins/vars](#).

## Filter Plugins

Filter plugins are used for manipulating data. They are a feature of Jinja2 and are also available in Jinja2 templates used by the `template` module. As with all plugins, they can be easily extended, but instead of having a file for each one you can have several per file. Most of the filter plugins shipped with Ansible reside in a `core.py`.

See [lib/ansible/plugins/filter](#) for details.

## Test Plugins

Test plugins are for verifying data. They are a feature of Jinja2 and are also available in Jinja2 templates used by the `template` module. As with all plugins, they can be easily extended, but instead of having a file for each one you can have several per file. Most of the test plugins shipped with Ansible reside in a `core.py`. These are specially useful in conjunction with some filter plugins like `map` and `select`; they are also available for conditional directives like `when`.

See [lib/ansible/plugins/test](#) for details.

## Distributing Plugins

Plugins are loaded from the library installed path and the configured plugins directory (check your `ansible.cfg`). The location can vary depending on how you installed Ansible (pip, rpm, deb, etc) or by the OS/Distribution/Packager. Plugins are automatically loaded when you have one of the following subfolders adjacent to your playbook or inside a role:

- `action_plugins`
- `lookup_plugins`
- `callback_plugins`
- `connection_plugins`
- `inventory_plugins`
- `filter_plugins`
- `strategy_plugins`
- `cache_plugins`
- `test_plugins`
- `shell_plugins`
- `vars_plugins`

When shipped as part of a role, the plugin will be available as soon as the role is called in the play.

:

**all\_modules** List of all modules

*Python API* Learn about the Python API for task execution

*Developing Dynamic Inventory* Learn about how to develop dynamic inventory sources

*Developing Modules* Learn about how to write Ansible modules

**Mailing List** The development mailing list

**irc.freenode.net** #ansible IRC chat channel

## 1.6.6 Developing Dynamic Inventory

### Topics

- *Inventory sources*
- *Inventory Plugins*
  - *Developing an Inventory Plugin*
    - \* *verify\_file*
    - \* *parse*
  - *inventory source common format*
  - *The 'auto' plugin*
- *Inventory Scripts*
  - *Inventory script conventions*
  - *Tuning the External Inventory Script*

As described in *Working With Dynamic Inventory*, Ansible can pull inventory information from dynamic sources, including cloud sources, using the supplied *inventory plugins*. If the source you want is not currently covered by existing plugins, you can create your own as with any other plugin type.

In previous versions you had to create a script or program that can output JSON in the correct format when invoked with the proper arguments. You can still use and write inventory scripts, as we ensured backwards compatibility via the script inventory plugin and there is no restriction on the programming language used. If you choose to write a script, however, you will need to implement some features yourself. i.e caching, configuration management, dynamic variable and group composition, etc. While with *inventory plugins* you can leverage the Ansible codebase to add these common features.

## Inventory sources

Inventory sources are strings (i.e what you pass to `-i` in the command line), they can represent a path to a file/script or just be the raw data for the plugin to use. Here are some plugins and the type of source they use:

Plugin	Source
host list	A comma separated list of hosts
yaml	Path to a YAML format data file
constructed	Path to a YAML configuration file
ini	Path to An ini formatted data file
virtualbox	Path to a YAML configuration file
script plugin	Path to an executable outputting JSON

## Inventory Plugins

Like most plugin types (except modules) they must be developed in Python, since they execute on the controller they should match the same requirements *Control Machine Requirements*.

Most of the documentation in *Developing Plugins* also applies here, so as to not repeat ourselves, you should read that document first and we'll include inventory plugin specifics next.

Inventory plugins normally only execute at the start of a run, before playbooks/plays and roles are loaded, but they can be 're-executed' via the `meta: refresh_inventory` task, which will clear out the existing inventory and rebuild it.

When using the 'persistent' cache, inventory plugins can also use the configured cache plugin to store and retrieve data to avoid costly external calls.

## Developing an Inventory Plugin

The first thing you want to do is use the base class:

```
from ansible.plugins.inventory import BaseInventoryPlugin

class InventoryModule(BaseInventoryPlugin):

    NAME = 'myplugin' # used internally by Ansible, it should match the file name_
    ↪but not required
```

This class has a couple of methods each plugin should implement and a few helpers for parsing the inventory source and updating the inventory.

After you have the basic plugin working you might want to incorporate other features by adding more base classes:



```

from ansible.plugins.inventory import BaseInventoryPlugin, Constructable, Cacheable

class InventoryModule(BaseInventoryPlugin, Constructable, Cacheable):

    NAME = 'myplugin'

```

For the bulk of the work in the plugin, We mostly want to deal with 2 methods `verify_file` and `parse`.

### verify\_file

This method is used by Ansible to make a quick determination if the inventory source is usable by the plugin. It does not need to be 100% accurate as there might be overlap in what plugins can handle and Ansible will try the enabled plugins (in order) by default.

```

def verify_file(self, path):
    ''' return true/false if this is possibly a valid file for this plugin to consume
    ↪ '''
    valid = False
    if super(InventoryModule, self).verify_file(path):
        # base class verifies that file exists and is readable by current user
        if path.endswith(('vbox.yaml', 'vbox.yml')):
            valid = True
    return valid

```

In this case, from the virtualbox inventory plugin, we screen for specific file name patterns to avoid attempting to consume any valid yaml file. You can add any type of condition here, but the most common one is 'extension matching'

Another example that actually does not use a 'file' but the inventory source string itself, from the host list plugin:

```

def verify_file(self, path):
    ''' don't call base class as we don't expect a path, but a host list '''
    host_list = path
    valid = False
    b_path = to_bytes(host_list, errors='surrogate_or_strict')
    if not os.path.exists(b_path) and ',' in host_list:
        # the path does NOT exist and there is a comma to indicate this is a 'host_
    ↪ list'
        valid = True
    return valid

```

This method is just to expedite the inventory process and avoid unnecessary parsing of sources that are easy to filter out before causing a parse error.

### parse

This method does the bulk of the work in the plugin.

It takes the following parameters:

- `inventory`: inventory object with existing data and the methods to add hosts/groups/variables to inventory
- `loader`: Ansible's `DataLoader`. The `DataLoader` can read files, auto load JSON/YAML and decrypt vaulted data, and cache read files.
- `path`: string with inventory source (this is usually a path, but is not required)
- `cache`: indicates whether the plugin should use or avoid caches (cache plugin and/or loader)

The base class does some minimal assignment for reuse in other methods.

```
def parse(self, inventory, loader, path, cache=True):

    self.loader = loader
    self.inventory = inventory
    self.templar = Templar(loader=loader)
```

It is up to the plugin now to deal with the inventory source provided and translate that into the Ansible inventory. To facilitate this there are a few of helper functions used in the example below:

```
NAME = 'myplugin'

def parse(self, inventory, loader, path, cache=True):

    # call base method to ensure properties are available for use with other helper_
    ↪methods
    super(InventoryModule, self).parse(inventory, loader, path, cache)

    # this method will parse 'common format' inventory sources and
    # update any options declared in DOCUMENTATION as needed
    config = self._read_config_data(self, path)

    # if NOT using _read_config_data you should call set_options directly,
    # to process any defined configuration for this plugin,
    # if you dont define any options you can skip
    #self.set_options()

    # example consuming options from inventory source
    mysession = apilib.session(user=self.get_option('api_user'),
                              password=self.get_option('api_pass'),
                              server=self.get_option('api_server')
    )

    # make requests to get data to feed into inventory
    mydata = myselss.getitall()

    #parse data and create inventory objects:
    for colo in mydata:
        for server in mydata[colo]['servers']:
            self.inventory.add_host(server['name'])
            self.inventory.set_variable('ansible_host', server['external_ip'])
```

The specifics will vary depending on API and structure returned. But one thing to keep in mind, if the inventory source or any other issue crops up you should raise `AnsibleParserError` to let Ansible know that the source was invalid or the process failed.

For examples on how to implement an inventory plug in, see the source code here: [lib/ansible/plugins/inventory](#).

## inventory source common format

To simplify development, most plugins use a mostly standard configuration file as the inventory source, YAML based and with just one required field `plugin` which should contain the name of the plugin that is expected to consume the file. Depending on other common features used, other fields might be needed, but each plugin can also add it's own custom options as needed. For example, if you use the integrated caching, `cache_plugin`, `cache_timeout` and other cache related fields could be present.

## The 'auto' plugin

Since Ansible 2.5, we include the auto inventory plugin enabled by default, which itself just loads other plugins if they use the common YAML configuration format that specifies a `plugin` field that matches an inventory plugin name, this makes it easier to use your plugin w/o having to update configurations.

## Inventory Scripts

Even though we now have inventory plugins, we still support inventory scripts, not only for backwards compatibility but also to allow users to leverage other programming languages.

### Inventory script conventions

Inventory scripts must accept the `--list` and `--host <hostname>` arguments, other arguments are allowed but Ansible will not use them. They might still be useful for when executing the scripts directly.

When the script is called with the single argument `--list`, the script must output to stdout a JSON-encoded hash or dictionary containing all of the groups to be managed. Each group's value should be either a hash or dictionary containing a list of each host, any child groups, and potential group variables, or simply a list of hosts:

```
{
  "group001": {
    "hosts": ["host001", "host002"],
    "vars": {
      "var1": true
    },
    "children": ["group002"]
  },
  "group002": {
    "hosts": ["host003", "host004"],
    "vars": {
      "var2": 500
    },
    "children": []
  }
}
```

If any of the elements of a group are empty they may be omitted from the output.

When called with the argument `--host <hostname>` (where `<hostname>` is a host from above), the script must print either an empty JSON hash/dictionary, or a hash/dictionary of variables to make available to templates and playbooks. For example:

```
{
  "VAR001": "VALUE",
  "VAR002": "VALUE",
}
```

Printing variables is optional. If the script does not do this, it should print an empty hash or dictionary.

## Tuning the External Inventory Script

1.3 .

The stock inventory script system detailed above works for all versions of Ansible, but calling `--host` for every host can be rather inefficient, especially if it involves API calls to a remote subsystem.

To avoid this inefficiency, if the inventory script returns a top level element called `"_meta"`, it is possible to return all of the host variables in one script execution. When this meta element contains a value for `"hostvars"`, the inventory script will not be invoked with `--host` for each host. This results in a significant performance increase for large numbers of hosts.

The data to be added to the top level JSON dictionary looks like this:

```
{

  # results of inventory script as above go here
  # ...

  "_meta": {
    "hostvars": {
      "host001": {
        "var001" : "value"
      },
      "host002": {
        "var002": "value"
      }
    }
  }
}
```

To satisfy the requirements of using `_meta`, to prevent ansible from calling your inventory with `--host` you must at least populate `_meta` with an empty `hostvars` dictionary. For example:

```
{

  # results of inventory script as above go here
  # ...

  "_meta": {
    "hostvars": {}
  }
}
```

If you intend to replace an existing static inventory file with an inventory script, it must return a JSON object which contains an `'all'` group that includes every host in the inventory as a member and every group in the inventory as a child. It should also include an `'ungrouped'` group which contains all hosts which are not members of any other group. A skeleton example of this JSON object is:

```
{

  "_meta": {
    "hostvars": {}
  },
  "all": {
    "children": [
      "ungrouped"
    ]
  },
  "ungrouped": {}
}
```

An easy way to see how this should look is using `ansible-inventory`, which also supports `--list` and `--host` parameters like an inventory script would.

:

*Python API* Python API to Playbooks and Ad Hoc Task Execution*Developing Modules* How to develop modules*Developing Plugins* How to develop plugins**Ansible Tower** REST API endpoint and GUI for Ansible, syncs with dynamic inventory**Development Mailing List** Mailing list for development topics**irc.freenode.net** #ansible IRC chat channel

## 1.6.7 Developing the Ansible Core Engine

Although many of the pieces of the Ansible Core Engine are plugins that can be swapped out via playbook directives or configuration, there are still pieces of the Engine that are not modular. The documents here give insight into how those pieces work together.

:

*Python API* Learn about the Python API for task execution*Developing Plugins* Learn about developing plugins**Mailing List** The development mailing list**irc.freenode.net** #ansible-devel IRC chat channel

## 1.6.8 Ansible and Python 3

Ansible is pursuing a strategy of having one code base that runs on both Python-2 and Python-3 because we want Ansible to be able to manage a wide variety of machines. Contributors to Ansible should be aware of the tips in this document so that they can write code that will run on the same versions of Python as the rest of Ansible.

Ansible can be divided into three overlapping pieces for the purposes of porting:

1. Controller-side code. This is the code which runs on the machine where you invoke `/usr/bin/ansible`
2. Modules. This is the code which Ansible transmits over the wire and invokes on the managed machine.
3. `module_utils` code. This is code whose primary purpose is to be used by the modules to perform tasks. However, some controller-side code might use generic functions from here.

Much of the knowledge of porting code will be usable on all three of these pieces but there are some special considerations for some of it as well. Information that is generally applicable to all three places is located in the controller-side section.

### Minimum Version of Python-3.x and Python-2.x

In both controller side and module code, we support Python-3.5 or greater and Python-2.6 or greater. Python-3.5 was chosen as a minimum because it is the earliest Python-3 version adopted as the default Python by a Long Term Support (LTS) Linux distribution (in this case, Ubuntu-16.04). Previous LTS Linux distributions shipped with a Python-2 version which users can rely upon instead of the Python-3 version.

For Python-2, the default is for modules to run on at least Python-2.6. This allows users with older distributions that are stuck on Python-2.6 to manage their machines. Modules are allowed to drop support for Python-2.6 when one of their dependent libraries requires a higher version of Python. This is not an invitation to add unnecessary dependent libraries

in order to force your module to be usable only with a newer version of Python; instead it is an acknowledgment that some libraries (for instance, boto3 and docker-py) will only function with a newer version of Python.

---

### : Python-2.4 Module-side Support:

Support for Python-2.4 and Python-2.5 was dropped in Ansible-2.4. RHEL-5 (and its rebuilds like CentOS-5) were supported until April of 2017. Ansible-2.3 was released in April of 2017 and was the last Ansible release to support Python-2.4 on the module-side.

---

## Porting Controller Code to Python 3

Most of the general tips for porting code to be used on both Python-2 and Python-3 applies to porting controller code. The best place to start learning to port code is [Lennart Regebro's book: Porting to Python 3](#).

The book describes several strategies for porting to Python 3. The one we're using is [to support Python-2 and Python-3 from a single code base](#)

## Controller String Strategy

### Background

One of the most essential things to decide upon for porting code to Python-3 is what string model to use. Strings can be an array of bytes (like in C) or they can be an array of text. Text is what we think of as letters, digits, numbers, other printable symbols, and a small number of unprintable "symbols" (control codes).

In Python-2, the two types for these (`str` for bytes and `unicode` for text) are often used interchangeably. When dealing only with ASCII characters, the strings can be combined, compared, and converted from one type to another automatically. When non-ASCII characters are introduced, Python starts throwing exceptions due to not knowing what encoding the non-ASCII characters should be in.

Python-3 changes this behavior by making the separation between bytes (`bytes`) and text (`str`) more strict. Python will throw an exception when trying to combine and compare the two types. The programmer has to explicitly convert from one type to the other to mix values from each.

This change makes it immediately apparent to the programmer when code is mixing the types inappropriately, rather than working until one of their users causes an exception by entering non-ASCII input. However, it forces the programmer to proactively define a strategy for working with strings in their program so that they don't mix text and byte strings unintentionally.

### Unicode Sandwich

In controller-side code we use a strategy known as the Unicode Sandwich (named after Python-2's `unicode` text type). For Unicode Sandwich we know that at the border of our code and the outside world (for example, file and network IO, environment variables, and some library calls) we are going to receive bytes. We need to transform these bytes into text and use that throughout the internal portions of our code. When we have to send those strings back out to the outside world we first convert the text back into bytes. To visualize this, imagine a 'sandwich' consisting of a top and bottom layer of bytes, a layer of conversion between, and all text type in the center.

## Common Borders

This is a partial list of places where we have to convert to and from bytes. It's not exhaustive but gives you an idea of where to watch for problems.

## Reading and writing to files

In Python-2, reading from files yields bytes. In Python-3, it can yield text. To make code that's portable to both we don't make use of Python-3's ability to yield text but instead do the conversion explicitly ourselves. For example:

```
from ansible.module_utils._text import to_text

with open('filename-with-utf8-data.txt', 'rb') as my_file:
    b_data = my_file.read()
    try:
        data = to_text(b_data, errors='surrogate_or_strict')
    except UnicodeError:
        # Handle the exception gracefully -- usually by displaying a good
        # user-centric error message that can be traced back to this piece
        # of code.
        pass
```

: Much of Ansible assumes that all encoded text is UTF-8. At some point, if there is demand for other encodings we may change that, but for now it is safe to assume that bytes are UTF-8.

Writing to files is the opposite process:

```
from ansible.module_utils._text import to_bytes

with open('filename.txt', 'wb') as my_file:
    my_file.write(to_bytes(some_text_string))
```

Note that we don't have to catch `UnicodeError` here because we're transforming to UTF-8 and all text strings in Python can be transformed back to UTF-8.

## Filesystem Interaction

Dealing with filenames often involves dropping back to bytes because on UNIX-like systems filenames are bytes. On Python-2, if we pass a text string to these functions, the text string will be converted to a byte string inside of the function and a traceback will occur if non-ASCII characters are present. In Python-3, a traceback will only occur if the text string can't be decoded in the current locale, but it's still good to be explicit and have code which works on both versions:

```
import os.path

from ansible.module_utils._text import to_bytes

filename = u'/var/tmp/.txt'
f = open(to_bytes(filename), 'wb')
mtime = os.path.getmtime(to_bytes(filename))
b_filename = os.path.expandvars(to_bytes(filename))
```

(continues on next page)

()

```
if os.path.exists(to_bytes(filename)):
    pass
```

When you are only manipulating a filename as a string without talking to the filesystem (or a C library which talks to the filesystem) you can often get away without converting to bytes:

```
import os.path

os.path.join(u'/var/tmp/café', u'')
os.path.split(u'/var/tmp/café/')
```

On the other hand, if the code needs to manipulate the filename and also talk to the filesystem, it can be more convenient to transform to bytes right away and manipulate in bytes.

: Make sure all variables passed to a function are the same type. If you're working with something like `os.path.join()` which takes multiple strings and uses them in combination, you need to make sure that all the types are the same (either all bytes or all text). Mixing bytes and text will cause tracebacks.

## Interacting with Other Programs

Interacting with other programs goes through the operating system and C libraries and operates on things that the UNIX kernel defines. These interfaces are all byte-oriented so the Python interface is byte oriented as well. On both Python-2 and Python-3, byte strings should be given to Python's subprocess library and byte strings should be expected back from it.

One of the main places in Ansible's controller code that we interact with other programs is the connection plugins' `exec_command` methods. These methods transform any text strings they receive in the command (and arguments to the command) to execute into bytes and return stdout and stderr as byte strings. Higher level functions (like action plugins' `_low_level_execute_command`) transform the output into text strings.

## Tips, tricks, and idioms to adopt

### Forwards Compatibility Boilerplate

Use the following boilerplate code at the top of all controller-side modules to make certain constructs act the same way on Python-2 and Python-3:

```
# Make coding more python3-ish
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type
```

`__metaclass__ = type` makes all classes defined in the file into new-style classes without explicitly inheriting from `object`.

The `__future__` imports do the following:

**absolute\_import** Makes imports look in `sys.path` for the modules being imported, skipping the directory in which the module doing the importing lives. If the code wants to use the directory in which the module doing the importing, there's a new dot notation to do so.

**division** Makes division of integers always return a float. If you need to find the quotient use `x // y` instead of `x / y`.



**print\_function** Changes `print()` from a keyword into a function.

:

- PEP 0328: Absolute Imports
- PEP 0238: Division
- PEP 3105: Print function

## Prefix byte strings with "b\_"

Since mixing text and bytes types leads to tracebacks we want to be clear about what variables hold text and what variables hold bytes. We do this by prefixing any variable holding bytes with `b_`. For instance:

```
filename = u'/var/tmp/café.txt'
b_filename = to_bytes(filename)
with open(b_filename) as f:
    data = f.read()
```

We do not prefix the text strings instead because we only operate on byte strings at the borders, so there are fewer variables that need bytes than text.

## Bundled six

The third-party `python-six` library exists to help projects create code that runs on both Python-2 and Python-3. Ansible includes a version of the library in `module_utils` so that other modules can use it without requiring that it is installed on the remote system. To make use of it, import it like this:

```
from ansible.module_utils import six
```

: Ansible can also use a system copy of `six`

Ansible will use a system copy of `six` if the system copy is a later version than the one Ansible bundles.

## Exceptions

In order for code to function on Python-2.6+ and Python-3, use the new exception-catching syntax which uses the `as` keyword:

```
try:
    a = 2/0
except ValueError as e:
    module.fail_json(msg="Tried to divide by zero: %s" % e)
```

Do **not** use the following syntax as it will fail on every version of Python-3:

```
try:
    a = 2/0
except ValueError, e:
    module.fail_json(msg="Tried to divide by zero: %s" % e)
```

## Octal numbers

In Python-2.x, octal literals could be specified as 0755. In Python-3, octals must be specified as 0o755.

## String formatting

### str.format() compatibility

Starting in Python-2.6, strings gained a method called `format()` to put strings together. However, one commonly used feature of `format()` wasn't added until Python-2.7, so you need to remember not to use it in Ansible code:

```
# Does not work in Python-2.6!
new_string = "Dear {}, Welcome to {}".format(username, location)

# Use this instead
new_string = "Dear {0}, Welcome to {1}".format(username, location)
```

Both of the format strings above map positional arguments of the `format()` method into the string. However, the first version doesn't work in Python-2.6. Always remember to put numbers into the placeholders so the code is compatible with Python-2.6.

:

Python documentation on [format strings](#)

### Use percent format with byte strings

In Python-3.x, byte strings do not have a `format()` method. However, it does have support for the older, percent-formatting.

```
b_command_line = b'ansible-playbook --become-user %s -K %s' % (user, playbook_file)
```

---

: Percent formatting added in Python-3.5

Percent formatting of byte strings was added back into Python3 in 3.5. This isn't a problem for us because Python-3.5 is our minimum version. However, if you happen to be testing Ansible code with Python-3.4 or earlier, you will find that the byte string formatting here won't work. Upgrade to Python-3.5 to test.

:

Python documentation on [percent formatting](#)

## Porting Modules to Python 3

Ansible modules are slightly harder to port than normal code from other projects. A lot of mocking has to go into unit testing an Ansible module so it's harder to test that your porting has fixed everything or to make sure that later commits haven't regressed the Python-3 support.

## Module String Strategy

There are a large number of modules in Ansible. Most of those are maintained by the Ansible community at large, not by a centralized team. To make life easier on them, it was decided not to break backwards compatibility by mandating that all strings inside of modules are text and converting between text and bytes at the borders; instead, we're using a native string strategy for now.

Native strings refer to the type that Python uses when you specify a bare string literal:

```
"This is a native string"
```

In Python-2, these are byte strings. In Python-3 these are text strings. The `module_utils` shipped with Ansible attempts to accept native strings as input to its functions and emit native strings as their output. Modules should be coded to expect bytes on Python-2 and text on Python-3.

## Tips, tricks, and idioms to adopt

### Python-2.4 Compatible Exception Syntax

Until Ansible-2.4, modules needed to be compatible with Python-2.4 as well. Python-2.4 did not understand the new exception-catching syntax so we had to write a compatibility function that could work with both Python-2 and Python-3. You may still see this used in some modules:

```
from ansible.module_utils.pycompat24 import get_exception

try:
    a = 2/0
except ValueError:
    e = get_exception()
    module.fail_json(msg="Tried to divide by zero: %s" % e)
```

Unless a change is going to be backported to Ansible-2.3, you should not have to use this in new code.

### Python 2.4 octal workaround

Before Ansible-2.4, modules had to be compatible with Python-2.4. Python-2.4 did not understand the new syntax for octal literals so we used the following workaround to specify octal values:

```
# Can't use 0755 on Python-3 and can't use 0o755 on Python-2.4
EXECUTABLE_PERMS = int('0755', 8)
```

Unless a change is going to be backported to Ansible-2.3, you should not have to use this in new code.

## Porting module\_utils code to Python 3

`module_utils` code is largely like module code. However, some pieces of it are used by the controller as well. Because of this, it needs to be usable with the controller's assumptions. This is most notable in the string strategy.

### Module\_utils String Strategy

`Module_utils` **must** use the Native String Strategy. Functions in `module_utils` receive either text strings or byte strings and may emit either the same type as they were given or the native string for the Python version they are run on

depending on which makes the most sense for that function. Functions which return strings **must** document whether they return text, byte, or native strings. Module-utils functions are therefore often very defensive in nature, converting from potential text or bytes at the beginning of a function and converting to the native string type at the end.

### 1.6.9 Python API

---

: This document is out of date; 'ansible.parsing.dataloader' and 'ansible.runner' are not available in the current version of Ansible.

---

#### Topics

- *Python API*
  - *Python API example*

---

: This API is intended for internal Ansible use. Ansible may make changes to this API at any time that could break backward compatibility with older versions of the API. Because of this, external use is not supported by Ansible.

---

There are several ways to use Ansible from an API perspective. You can use the Ansible Python API to control nodes, you can extend Ansible to respond to various Python events, you can write plugins, and you can plug in inventory data from external data sources. This document gives a basic overview and examples of the Ansible execution and playbook API.

If you would like to use Ansible programmatically from a language other than Python, trigger events asynchronously, or have access control and logging demands, please see the [Ansible Tower documentation](#).

---

: Because Ansible relies on forking processes, this API is not thread safe.

---

#### Python API example

This example is a simple demonstration that shows how to minimally run a couple of tasks:

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

import json
import shutil
from collections import namedtuple
from ansible.parsing.dataloader import DataLoader
from ansible.vars.manager import VariableManager
from ansible.inventory.manager import InventoryManager
from ansible.playbook.play import Play
from ansible.executor.task_queue_manager import TaskQueueManager
from ansible.plugins.callback import CallbackBase
import ansible.constants as C

class ResultCallback(CallbackBase):
    """A sample callback plugin used for performing an action as results come in
```

(continues on next page)

()

```

If you want to collect all results into a single object for processing at
the end of the execution, look into utilizing the ``json`` callback plugin
or writing your own custom callback plugin
"""
def v2_runner_on_ok(self, result, **kwargs):
    """Print a json representation of the result

    This method could store the result in an instance attribute for retrieval
    ↪later
    """
    host = result._host
    print(json.dumps({host.name: result._result}, indent=4))

# since API is constructed for CLI it expects certain options to always be set, named
↪tuple 'fakes' the args parsing options object
Options = namedtuple('Options', ['connection', 'module_path', 'forks', 'become',
↪'become_method', 'become_user', 'check', 'diff'])
options = Options(connection='local', module_path=['/to/mymodules'], forks=10,
↪become=None, become_method=None, become_user=None, check=False, diff=False)

# initialize needed objects
loader = DataLoader() # Takes care of finding and reading yaml, json and ini files
passwords = dict(vault_pass='secret')

# Instantiate our ResultCallback for handling results as they come in. Ansible
↪expects this to be one of its main display outlets
results_callback = ResultCallback()

# create inventory, use path to host config file as source or hosts in a comma
↪separated string
inventory = InventoryManager(loader=loader, sources='localhost,')

# variable manager takes care of merging all the different sources to give you a
↪unified view of variables available in each context
variable_manager = VariableManager(loader=loader, inventory=inventory)

# create datastructure that represents our play, including tasks, this is basically
↪what our YAML loader does internally.
play_source = dict(
    name = "Ansible Play",
    hosts = 'localhost',
    gather_facts = 'no',
    tasks = [
        dict(action=dict(module='shell', args='ls'), register='shell_out'),
        dict(action=dict(module='debug', args=dict(msg='{{shell_out.stdout}}'))
    ]
)

# Create play object, playbook objects use .load instead of init or new methods,
# this will also automatically create the task objects from the info provided in play
↪source
play = Play().load(play_source, variable_manager=variable_manager, loader=loader)

# Run it - instantiate task queue manager, which takes care of forking and setting up
↪all objects to iterate over host list and tasks
tqm = None
try:

```

(continues on next page)

```

    tqm = TaskQueueManager(
        inventory=inventory,
        variable_manager=variable_manager,
        loader=loader,
        options=options,
        passwords=passwords,
        stdout_callback=results_callback, # Use our custom callback instead of
↳ the ``default`` callback plugin, which prints to stdout
    )
    result = tqm.run(play) # most interesting data for a play is actually sent to the
↳ callback's methods
finally:
    # we always need to cleanup child procs and the structures we use to communicate
↳ with them
    if tqm is not None:
        tqm.cleanup()

    # Remove ansible tmpdir
    shutil.rmtree(C.DEFAULT_LOCAL_TMP, True)

```

: Ansible emits warnings and errors via the display object, which prints directly to stdout, stderr and the Ansible log.

The source code for the `ansible` command line tools (`lib/ansible/cli/`) is [available on Github](#).

:

**Developing Dynamic Inventory** Developing dynamic inventory integrations

**Developing Modules** How to develop modules

**Developing Plugins** How to develop plugins

**Development Mailing List** Mailing list for development topics

**irc.freenode.net** #ansible IRC chat channel

## 1.6.10 Rebasing a Pull Request

You may find that your pull request (PR) is out-of-date and needs to be rebased. This can happen for several reasons:

- Files modified in your PR are in conflict with changes which have already been merged.
- Your PR is old enough that significant changes to automated test infrastructure have occurred.

Rebasing the branch used to create your PR will resolve both of these issues.

### Configuring Your Remotes

Before you can rebase your PR, you need to make sure you have the proper remotes configured. Assuming you cloned your fork in the usual fashion, the `origin` remote will point to your fork:

```

$ git remote -v
origin  git@github.com:YOUR_GITHUB_USERNAME/ansible.git (fetch)
origin  git@github.com:YOUR_GITHUB_USERNAME/ansible.git (push)

```

However, you also need to add a remote which points to the upstream repository:

```
$ git remote add upstream https://github.com/ansible/ansible.git
```

Which should leave you with the following remotes:

```
$ git remote -v
origin  git@github.com:YOUR_GITHUB_USERNAME/ansible.git (fetch)
origin  git@github.com:YOUR_GITHUB_USERNAME/ansible.git (push)
upstream      https://github.com/ansible/ansible.git (fetch)
upstream      https://github.com/ansible/ansible.git (push)
```

Checking the status of your branch should show you're up-to-date with your fork at the `origin` remote:

```
$ git status
On branch YOUR_BRANCH
Your branch is up-to-date with 'origin/YOUR_BRANCH'.
nothing to commit, working tree clean
```

## Rebasing Your Branch

Once you have an `upstream` remote configured, you can rebase the branch for your PR:

```
$ git pull --rebase upstream devel
```

This will replay the changes in your branch on top of the changes made in the upstream `devel` branch. If there are merge conflicts, you will be prompted to resolve those before you can continue.

Once you've rebased, the status of your branch will have changed:

```
$ git status
On branch YOUR_BRANCH
Your branch and 'origin/YOUR_BRANCH' have diverged,
and have 4 and 1 different commits each, respectively.
(use "git pull" to merge the remote branch into yours)
nothing to commit, working tree clean
```

Don't worry, this is normal after a rebase. You should ignore the `git status` instructions to use `git pull`. We'll cover what to do next in the following section.

## Updating Your Pull Request

Now that you've rebased your branch, you need to push your changes to GitHub to update your PR.

Since rebasing re-writes git history, you will need to use a force push:

```
$ git push --force
```

Your PR on GitHub has now been updated. This will automatically trigger testing of your changes. You should check in on the status of your PR after tests have completed to see if further changes are required.

## Getting Help Rebasing

For help with rebasing your PR, or other development related questions, join us on our [#ansible-devel](#) IRC chat channel on [freenode.net](#).

## 1.6.11 Testing Ansible

### Topics

- *Testing Ansible*
  - *Introduction*
  - *Types of tests*
  - *Testing within GitHub & Shippable*
    - \* *Organization*
    - \* *Rerunning a failing CI job*
  - *How to test a PR*
    - \* *Setup: Checking out a Pull Request*
    - \* *Testing the Pull Request*
      - *Code Coverage Online*
  - *Want to know more about testing?*

### Introduction

This document describes:

- how Ansible is tested
- how to test Ansible locally
- how to extend the testing capabilities

### Types of tests

At a high level we have the following classifications of tests:

#### **compile**

- testing\_compile
- Test python code against a variety of Python versions.

#### **sanity**

- testing\_sanity
- Sanity tests are made up of scripts and tools used to perform static code analysis.
- The primary purpose of these tests is to enforce Ansible coding standards and requirements.

#### **integration**

- testing\_integration
- Functional tests of modules and Ansible core functionality.

#### **units**

- testing\_units



- Tests directly against individual parts of the code base.

If you're a developer, one of the most valuable things you can do is look at the GitHub issues list and help fix bugs. We almost always prioritize bug fixing over feature development.

Even for non developers, helping to test pull requests for bug fixes and features is still immensely valuable. Ansible users who understand writing playbooks and roles should be able to add integration tests and so Github pull requests with integration tests that show bugs in action will also be a great way to help.

## Testing within GitHub & Shippable

### Organization

When Pull Requests (PRs) are created they are tested using Shippable, a Continuous Integration (CI) tool. Results are shown at the end of every PR.

When Shippable detects an error and it can be linked back to a file that has been modified in the PR then the relevant lines will be added as a GitHub comment. For example:

```
The test `ansible-test sanity --test pep8` failed with the following errors:

lib/ansible/modules/network/foo/bar.py:509:17: E265 block comment should start with '
↪# '

The test `ansible-test sanity --test validate-modules` failed with the following
↪errors:
lib/ansible/modules/network/foo/bar.py:0:0: E307 version_added should be 2.4.
↪Currently 2.3
lib/ansible/modules/network/foo/bar.py:0:0: E316 ANSIBLE_METADATA.metadata_version:
↪required key not provided @ data['metadata_version']. Got None
```

From the above example we can see that `--test pep8` and `--test validate-modules` have identified issues. The commands given allow you to run the same tests locally to ensure you've fixed the issues without having to push your changes to GitHub and wait for Shippable, for example:

If you haven't already got Ansible available, use the local checkout by running:

```
source hacking/env-setup
```

Then run the tests detailed in the GitHub comment:

```
ansible-test sanity --test pep8
ansible-test sanity --test validate-modules
```

If there isn't a GitHub comment stating what's failed you can inspect the results by clicking on the "Details" button under the "checks have failed" message at the end of the PR.

### Rerunning a failing CI job

Occasionally you may find your PR fails due to a reason unrelated to your change. This could happen for several reasons, including:

- a temporary issue accessing an external resource, such as a yum or git repo
- a timeout creating a virtual machine to run the tests on

If either of these issues appear to be the case, you can rerun the Shippable test by:

- closing and re-opening the PR
- making another change to the PR and pushing to GitHub

If the issue persists, please contact us in `#ansible-devel` on Freenode IRC.

### How to test a PR

Ideally, code should add tests that prove that the code works. That's not always possible and tests are not always comprehensive, especially when a user doesn't have access to a wide variety of platforms, or is using an API or web service. In these cases, live testing against real equipment can be more valuable than automation that runs against simulated interfaces. In any case, things should always be tested manually the first time as well.

Thankfully, helping to test Ansible is pretty straightforward, assuming you are familiar with how Ansible works.

### Setup: Checking out a Pull Request

You can do this by:

- checking out Ansible
- making a test branch off the main branch
- merging a GitHub issue
- testing
- commenting on that particular issue on GitHub

Here's how:

: Testing source code from GitHub pull requests sent to us does have some inherent risk, as the source code sent may have mistakes or malicious code that could have a negative impact on your system. We recommend doing all testing on a virtual machine, whether a cloud instance, or locally. Some users like Vagrant or Docker for this, but they are optional. It is also useful to have virtual machines of different Linux or other flavors, since some features (apt vs. yum, for example) are specific to those OS versions.

Create a fresh area to work:

```
git clone https://github.com/ansible/ansible.git ansible-pr-testing
cd ansible-pr-testing
```

Next, find the pull request you'd like to test and make note of the line at the top which describes the source and destination repositories. It will look something like this:

```
Someuser wants to merge 1 commit into ansible:devel from someuser:feature_branch_name
```

---

: Only test `ansible:devel`

It is important that the PR request target be `ansible:devel`, as we do not accept pull requests into any other branch. Dot releases are cherry-picked manually by Ansible staff.

---

The username and branch at the end are the important parts, which will be turned into git commands as follows:

```
git checkout -b testing_PRXXXX devel
git pull https://github.com/someuser/ansible.git feature_branch_name
```

The first command creates and switches to a new branch named `testing_PRXXXX`, where the `XXXX` is the actual issue number associated with the pull request (for example, 1234). This branch is based on the `devel` branch. The second command pulls the new code from the users feature branch into the newly created branch.

---

: If the GitHub user interface shows that the pull request will not merge cleanly, we do not recommend proceeding if you are not somewhat familiar with git and coding, as you will have to resolve a merge conflict. This is the responsibility of the original pull request contributor.

---



---

: Some users do not create feature branches, which can cause problems when they have multiple, unrelated commits in their version of `devel`. If the source looks like `someuser:devel`, make sure there is only one commit listed on the pull request.

---

The Ansible source includes a script that allows you to use Ansible directly from source without requiring a full installation that is frequently used by developers on Ansible.

Simply source it (to use the Linux/Unix terminology) to begin using it immediately:

```
source ./hacking/env-setup
```

This script modifies the `PYTHONPATH` environment variables (along with a few other things), which will be temporarily set as long as your shell session is open.

## Testing the Pull Request

At this point, you should be ready to begin testing!

Some ideas of what to test are:

- Create a test Playbook with the examples in and check if they function correctly
- Test to see if any Python backtraces returned (that's a bug)
- Test on different operating systems, or against different library versions

Any potential issues should be added as comments on the pull request (and it's acceptable to comment if the feature works as well), remembering to include the output of `ansible --version`

Example:

```
Works for me! Tested on `Ansible 2.3.0`. I verified this on CentOS 6.5 and also_
↪Ubuntu 14.04.
```

If the PR does not resolve the issue, or if you see any failures from the unit/integration tests, just include that output instead:

This doesn't work for me.

When I ran this Ubuntu 16.04 it failed with the following:

```
""
some output
```

```
StackTrace
some other output
...

```

## Code Coverage Online

The [online code coverage reports](#) are a good way to identify areas for testing improvement in Ansible. By following red colors you can drill down through the reports to find files which have no tests at all. Adding both integration and unit tests which show clearly how code should work, verify important Ansible functions and increase testing coverage in areas where there is none is a valuable way to help improve Ansible.

The code coverage reports only cover the `devel` branch of Ansible where new feature development takes place. Pull requests and new code will be missing from the [codecov.io](#) coverage reports so local reporting is needed. Most `ansible-test` commands allow you to collect code coverage, this is particularly useful to indicate where to extend testing. See [testing\\_running\\_locally](#) for more information.

## Want to know more about testing?

If you'd like to know more about the plans for improving testing Ansible then why not join the [Testing Working Group](#).

## 1.6.12 Repo Merge

### Background

On Tuesday 6th December 2016, the Ansible Core Team re-merged the module repositories back into [ansible/ansible](#) in GitHub. The two module repos will be essentially locked, though they will be kept in place for the existing 2.1 and 2.2 dependencies. Once 2.2 moves out of official support (early 2018), these repositories will be fully read-only for all branches. Until then, any issues/PRs opened there will be auto-closed with a note to open it on [ansible/ansible](#).

### Why Are We Doing This (Again...)?

For those who've been using Ansible long enough, you know that originally we started with a single repository. The original intention of the core vs. extras split was that core would be better supported/tested/etc. Extras would have been a bit more of a "wild-west" for modules, to allow new modules to make it into the distribution more quickly. Unfortunately this never really worked out, as well as the following:

1. Many modules in the core repo were also essentially "grand-fathered" in, despite not having a good set of tests or dedicated maintainers from the community.
2. The time in queue for modules to be merged into extras was not really any different from the time to merge modules into core.
3. The split introduced a few other problems for contributors such as having to submit multiple related PRs for modules with tests, or for those which rely on action plugins.
4. git submodules are notoriously complicated, even for contributors with decent git experience. The constant need to update git submodule pointers for `devel` and each stable branch can lead to testing surprises and really buys us nothing in terms of flexibility.
5. Users can already be confused about where to open issues, especially when the problem appears to be with a module but is actually an action plugin (ie. `template`) or something more fundamental like `includes`. Having everything back in one repo makes it easier to link issues, and you're always sure to open a bug report in the right place.

## Metadata - Support/Ownership and Module Status

As part of this move, we will be introducing module metadata, which will contain a couple of pieces of information regarding modules:

1. **Support Status:** This field indicates who supports the module, whether it's the core team, the community, the person who wrote it, or if it is an abandoned module which is not receiving regular updates. The Ansible team has gone through the list of modules and we have marked about 100 of them as "Core Supported", meaning a member of the Ansible core team should be actively fixing bugs on those modules. The vast majority of the rest will be community supported. This is not really a change from the status quo, this just makes it clearer.
2. **Module Status:** This field indicates how well supported that module may be. This generally applies to the maturity of the module's parameters, however, not necessarily its bug status.

The documentation pages for modules will be updated to reflect the above information as well, so that users can evaluate the status of a module before committing to using it in playbooks and roles.

### 1.6.13 Release and maintenance

#### Topics

- [Release cycle](#)
- [Release status](#)
- [Development and stable version maintenance workflow](#)
  - [Changelogs](#)
  - [Release candidates](#)
  - [Feature freeze](#)
- [Deprecation Cycle](#)

#### Release cycle

Ansible is developed and released on a flexible 4 months release cycle. This cycle can be extended in order to allow for larger changes to be properly implemented and tested before a new release is made available.

Ansible has a graduated support structure that extends to three major releases. For more information, read about the [Development and stable version maintenance workflow](#) or see the chart in [Release status](#) for the degrees to which current releases are supported.

---

: Support for three major releases began with Ansible-2.4. Ansible-2.3 and older versions are only supported for two releases.

---

If you are using a release of Ansible that is no longer supported, we strongly encourage you to upgrade as soon as possible in order to benefit from the latest features and security fixes.

Older, unsupported versions of Ansible can contain unfixed security vulnerabilities (*CVE*).

You can refer to the [porting guides](#) for tips on updating your Ansible playbooks to run on newer versions.

## Release status

Ansible Release	Latest Version	Status
devel	2.6 (unreleased, trunk)	In development
2.5	2.5.4 (2018-05-31)	Supported (security <b>and</b> general bugfixes)
2.4	2.4.4 (2018-01-31)	Supported (security <b>and</b> critical bug fixes)
2.3	2.3.3 (2017-12-20)	Unsupported (end of life)
2.2	2.2.3 (2017-05-09)	Unsupported (end of life)
2.1	2.1.6 (2017-06-01)	Unsupported (end of life)
2.0	2.0.2 (2016-04-19)	Unsupported (end of life)
1.9	1.9.6 (2016-04-15)	Unsupported (end of life)
<1.9	n/a	Unsupported (end of life)

---

: Starting with Ansible-2.4, support lasts for 3 releases. Thus Ansible-2.4 will receive security and general bug fixes when it is first released, security and critical bug fixes when 2.5 is released, and **only** security fixes once 2.6 is released.

---

## Development and stable version maintenance workflow

The Ansible community develops and maintains Ansible on [GitHub](#).

New modules, plugins, features and bugfixes will always be integrated in what will become the next major version of Ansible. This work is tracked on the `devel` git branch.

Ansible provides bugfixes and security improvements for the most recent major release. The previous major release will only receive fixes for security issues and critical bugs. Ansible only applies security fixes to releases which are two releases old. This work is tracked on the `stable-<version>` git branches.

---

: Support for three major releases began with Ansible-2.4. Ansible-2.3 and older versions are only supported for two releases with the first stage including both security and general bug fixes while the second stage includes security and critical bug fixes

---

The fixes that land in supported stable branches will eventually be released as a new version when necessary.

Note that while there are no guarantees for providing fixes for unsupported releases of Ansible, there can sometimes be exceptions for critical issues.

## Changelogs

Since 2.5, we've logged changes to `stable-<version>` git branches at `stable-<version>/changelogs/CHANGELOG-v<version>.rst`. For example, here's the changelog for 2.5 on [GitHub](#).

Older versions logged changes to `stable-<version>/CHANGELOG.md`. For example, here's the CHANGELOG for 2.4.

## Release candidates

Before a new release or version of Ansible can be done, it will typically go through a release candidate process.

This provides the Ansible community the opportunity to test Ansible and report bugs or issues they might come across.

Ansible tags the first release candidate (RC1) which is usually scheduled to last five business days. The final release is done if no major bugs or issues are identified during this period.

If there are major problems with the first candidate, a second candidate will be tagged (RC2) once the necessary fixes have landed. This second candidate lasts for a shorter duration than the first. If no problems have been reported after two business days, the final release is done.

More release candidates can be tagged as required, so long as there are bugs that the Ansible core maintainers consider should be fixed before the final release.

## Feature freeze

While there is a pending release candidate, the focus of core developers and maintainers will on fixes towards the release candidate.

Merging new features or fixes that are not related to the release candidate may be delayed in order to allow the new release to be shipped as soon as possible.

## Deprecation Cycle

Sometimes we need to remove a feature, normally in favor of a reimplementaion that we hope does a better job. To do this we have a deprecation cycle. First we mark a feature as 'deprecated'. This is normally accompanied with warnings to the user as to why we deprecated it, what alternatives they should switch to and when (which version) we are scheduled to remove the feature permanently.

The cycle is normally across 4 feature releases (2.x.y, where the x marks a feature release and the y a bugfix release), so the feature is normally removed in the 4th release after we announce the deprecation. For example, something deprecated in 2.5 will be removed in 2.9, assuming we don't jump to 3.x before that point. The tracking is tied to the number of releases, not the release numbering.

For modules/plugins, we keep the documentation after the removal for users of older versions.

:

*Committers Guidelines (for people with commit rights to Ansible on GitHub)* Guidelines for Ansible core contributors and maintainers

*Testing Strategies* Testing strategies

*Ansible Community Guide* Community information and contributing

*Ansible release tarballs* Ansible release tarballs

*Development Mailing List* Mailing list for development topics

*irc.freenode.net* #ansible IRC chat channel

### 1.6.14 Committers Guidelines (for people with commit rights to Ansible on GitHub)

These are the guidelines for people with commit access to Ansible. Committers are essentially acting as members of the Ansible Core team, although not necessarily as an employee of Ansible and Red Hat. Please read the guidelines before you commit.

These guidelines apply to everyone. At the same time, this ISN'T a process document. So just use good judgement. You've been given commit access because we trust your judgement.

That said, use the trust wisely.

If you abuse the trust and break components and builds, etc., the trust level falls and you may be asked not to commit or you may lose access to do so.

### Features, High Level Design, and Roadmap

As a core team member, you are an integral part of the team that develops the *roadmap*. Please be engaged, and push for the features and fixes that you want to see. Also keep in mind that Red Hat, as a company, will commit to certain features, fixes, APIs, etc. for various releases. Red Hat, the company, and the Ansible team must get these committed features (etc.) completed and released as scheduled. Obligations to users, the community, and customers must come first. Because of these commitments, a feature you want to develop yourself may not get into a release if it impacts a lot of other parts within Ansible.

Any other new features and changes to high level design should go through the proposal process (TBD), to ensure the community and core team have had a chance to review the idea and approve it. The core team has sole responsibility for merging new features based on proposals.

### Our Workflow on GitHub

As a committer, you may already know this, but our workflow forms a lot of our team policies. Please ensure you're aware of the following workflow steps:

- Fork the repository upon which you want to do some work to your own personal repository
- Work on the specific branch upon which you need to commit
- Create a Pull Request back to the Ansible repository and tag the people you would like to review; assign someone as the primary "owner" of your request
- Adjust code as necessary based on the Comments provided
- Ask someone on the Core Team to do a final review and merge

### Addendum to workflow for Committers:

The Core Team is aware that this can be a difficult process at times. Sometimes, the team breaks the rules: Direct commits, merging their own PRs. This section is a set of guidelines. If you're changing a comma in a doc, or making a very minor change, you can use your best judgement. This is another trust thing. The process is critical for any major change, but for little things or getting something done quickly, use your best judgement and make sure people on the team are aware of your work.

### Roles on Core

- Core Committers: Fine to do PRs for most things, but we should have a timebox. Hanging PRs may merge on the judgement of these devs.
- Module Owners: Module Owners own specific modules and have indirect commit access via the current module PR mechanisms.

### General Rules

Individuals with direct commit access to ansible/ansible are entrusted with powers that allow them to do a broad variety of things—probably more than we can write down. Rather than rules, treat these as general *guidelines*, individuals with this power are expected to use their best judgement.



- Don't
  - Commit directly.
  - Merge your own PRs. Someone else should have a chance to review and approve the PR merge. If you are a Core Committer, you have a small amount of leeway here for very minor changes.
  - Forget about alternate environments. Consider the alternatives—yes, people have bad environments, but they are the ones who need us the most.
  - Drag your community team members down. Always discuss the technical merits, but you should never address the person's limitations (you can later go for beers and call them idiots, but not in IRC/Github/etc.).
  - Forget about the maintenance burden. Some things are really cool to have, but they might not be worth shoehorning in if the maintenance burden is too great.
  - Break playbooks. Always keep backwards compatibility in mind.
  - Forget to keep it simple. Complexity breeds all kinds of problems.
- Do
  - Squash, avoid merges whenever possible, use github's squash commits or cherry pick if needed (bisect thanks you).
  - Be active. Committers who have no activity on the project (through merges, triage, commits, etc.) will have their permissions suspended.
  - Consider backwards compatibility (goes back to "don't break existing playbooks").
  - Write tests. PRs with tests are looked at with more priority than PRs without tests that should have them included. While not all changes require tests, be sure to add them for bug fixes or functionality changes.
  - Discuss with other committers, specially when you are unsure of something.
  - Document! If your PR is a new feature or a change to behavior, make sure you've updated all associated documentation or have notified the right people to do so. It also helps to add the version of Core against which this documentation is compatible (to avoid confusion with stable versus devel docs, for backwards compatibility, etc.).
  - Consider scope, sometimes a fix can be generalized
  - Keep it simple, then things are maintainable, debuggable and intelligible.

Committers are expected to continue to follow the same community and contribution guidelines followed by the rest of the Ansible community.

## People

Individuals who've been asked to become a part of this group have generally been contributing in significant ways to the Ansible community for some time. Should they agree, they are requested to add their names and GitHub IDs to this file, in the section below, via a pull request. Doing so indicates that these individuals agree to act in the ways that their fellow committers trust that they will act.

Name	Github ID	IRC Nick	Other
James Cammarata	jimi-c	jimi	
Brian Coca	bcoca	bcoca	
Matt Davis	nitzmahone	nitzmahone	
Toshio Kuratomi	abadger	abadger1999	
Jason McKerr	mckerrj	newtMcKerr	

1 –

Name	Github ID	IRC Nick	Other
Robyn Bergeron	robynbergeron	rbergeron	
Greg DeKoenigsberg	gregdek	gregdek	
Monty Taylor	emonty	mordred	
Matt Martz	sivel	sivel	
Nate Case	qalthos	Qalthos	
James Tanner	jctanner	jtanner	
Peter Sprygada	privateip	privateip	
Abhijit Menon-Sen	amenonsen	crab	
Michael Scherer	mscherer	misc	
René Moser	resmo	resmo	
David Shrewsbury	Shrews	Shrews	
Sandra Wills	docschick	docschick	
Graham Mainwaring	ghjm		
Chris Houseknecht	chouseknecht		
Trond Hindenes	trondhinden		
Jon Hawkesworth	jhawkesworth	jhawkesworth	
Will Thames	willthames	willthames	
Ryan Brown	ryansb	ryansb	
Adrian Likins	alikins	alikins	
Dag Wieers	dagwieers	dagwieers	<a href="mailto:dag@wieers.com">dag@wieers.com</a>
Tim Rupp	caphrim007	caphrim007	
Sloane Hertel	s-hertel	shertel	
Sam Doran	samdoran	samdoran	
Scott Butler	dharmabumstead	dharmabumstead	
Matt Clay	mattclay	mattclay	
Martin Krizek	mkrizek	mkrizek	
Kedar Kekan	kedarX	kedarX	
Ganesh Nalawade	ganeshrn	ganeshrn	
Trishna Guha	trishnaguha	trishnag	
Andrew Gaffney	agaffney	agaffney	

## 1.6.15 Ansible Style Guide

### Why Use a Style Guide?

Style guides are important because they ensure consistency in the content, look, and feel of a book or a website.

Remember, a style guide is only useful if it is used, updated, and enforced. Style Guides are useful for engineering-related documentation, sales and marketing materials, support docs, community contributions, and more.

As changes are made to the overall Ansible site design, be sure to update this style guide with those changes. Or, should other resources listed below have major revisions, consider including company information here for ease of reference.

This style guide incorporates current Ansible resources and information so that overall site and documentation consistency can be met.

### Resources

- Follow the style of the *Ansible Documentation*

- Ask for advice on IRC, on the `#ansible-devel` Freenode channel
- Review these online style guides:
  - [AP Stylebook](#)
  - [Chicago Manual of Style](#)
  - [Strunk and White's Elements of Style](#)

## Basic Rules

### Use Standard American English

Ansible has customers/users all around the globe, but the headquarters is in Durham, NC, in the US. Use Standard American English rather than other variations of the English language.

### Write for a Global Audience

The idea behind global writing is that everything you say should be understandable by those of many different backgrounds and cultures. References, therefore, should be as universal as possible. Avoid idioms and regionalism and maintain a neutral tone that cannot be misinterpreted. Avoid attempts at humor.

### Follow Naming Conventions

Always follow naming conventions and trademarks. If you aren't sure how a product should be properly referred to, ask the Engineering Product Manager of that product line (`ansible-core` or `Tower`) for information.

### Important Information First

Important information stated at the beginning of a sentence makes it easier to understand.

Unclear: The unwise walking about upon the area near the cliff edge may result in a dangerous fall and therefore it is recommended that one remains a safe distance to maintain personal safety.

Clearer: Danger! Stay away from cliff.

### Sentence Structure

Good sentence structure helps convey information. Try to keep the most important information towards the beginning of the sentence.

Bad: Furthermore, large volumes of water are also required for the process of extraction.

Better: Extraction also requires large volumes of water.

### Avoid padding

When reading a piece of technical writing, the audience does not benefit from elaborate prose. They just need information on how to perform a task. Avoid using padding, or filler. Don't use phrases such as, kind of, sort of, and essentially.

### Avoid redundant prepositional phrases

Prepositional phrases, the combination of a preposition with a noun phrase, are among the worst offenders in making text long and tiresome to read. Often, it is possible to replace an entire phrase with a single word.

Use now instead of at this point in time. Use suddenly instead of all of the sudden.

### Avoid verbosity

Write short, succinct sentences. Never say, "...as has been said before," "...each and every," "...point in time," etc. Avoid "...in order to," especially at the beginning of sentences. Every word must contribute meaning to the sentence. Technical writing is information delivery.

### Avoid pomposity

While it is good to have a wide vocabulary, technical writing is not the place for showing off linguistic abilities. Technical writing is about producing clear, plain instructions for a specific audience.

### Action verbs, menus, and commands

We interact with computers in a variety of ways. You can select anything on an application user interface by selecting it using a keyboard or mouse. It is important to use action verbs and software terminology correctly.

The most frequent verbs used in software are:

- Click
- Double-click
- Select
- Type
- Press

Use of an action verb in a sentence (**bolded** words):

1. In the dialog box, click **Open**.
2. **Type** a name in the text box.
3. On the keyboard press **Enter**.

Use of menu actions and commands in a sentence:

1. On the **File** menu, click **Open**.
2. **Type** a name in the **User Name** field.
3. In the **Open** dialog box, click **Save**.
4. On the computer keyboard, press **Enter**.
5. On the toolbar, click the **Open File** icon.

Make users aware of where they are in the application. If there is more than one method to perform an action, use the most common method. Define "what, where, and how" in each step of the task or procedure. Describe menu items for the current task left to right, top-down.

## Voice Style

The essence of the Ansible writing style is short sentences that flow naturally together. Mix up sentence structures. Vary sentence subjects. Address the reader directly. Ask a question. And when the reader adjusts to the pace of shorter sentences, write a longer one.

- Write how real people speak...
- ...but try to avoid slang and colloquialisms that might not translate well into other languages.
- Say big things with small words.
- Be direct. Tell the reader exactly what you want them to do.
- Be honest.
- Short sentences show confidence.
- Grammar rules are meant to be bent, but only if the reader knows you are doing this.
- Choose words with fewer syllables for faster reading and better understanding.
- Think of copy as one-on-one conversations rather than as a speech. It's more difficult to ignore someone who is speaking to you directly.
- When possible, start task-oriented sentences (those that direct a user to do something) with action words. For example: Find software... Contact support... Install the media.... and so forth.

## Active Voice

Use the active voice ("Start Linuxconf by typing...") rather than passive ("Linuxconf can be started by typing...") whenever possible. Active voice makes for more lively, interesting reading. Also avoid future tense (or using the term "will") whenever possible. For example, future tense ("The screen will display...") does not read as well as an active voice ("The screen displays"). Remember, the users you are writing for most often refer to the documentation while they are using the system, not after or in advance of using the system.

## Trademark Usage

Why is it important to use the TM, SM, and ® for our registered marks?

Before a trademark is registered with the United States Patent and Trademark Office it is appropriate to use the TM or SM symbol depending whether the product is for goods or services. It is important to use the TM or SM as it is notification to the public that Ansible claims rights to the mark even though it has not yet been registered.

Once the trademark is registered, it is appropriate to use the symbol in place of the TM or SM. The symbol designation must be used in conjunction with the trademark if Ansible is to fully protect its rights. If we don't protect these marks, we run the risk of losing them in the way of Aspirin or Trampoline or Escalator.

## General Rules:

Trademarks should be used on 1st references on a page or within a section.

Use Ansible Tower® or Ansible®, on first reference when referring to products.

Use "Ansible" alone as the company name, as in "Ansible announced quarterly results," which is not marked.

Also add the trademark disclaimer. \* When using Ansible trademarks in the body of written text, you should use the following credit line in a prominent place, usually a footnote.

For Registered Trademarks: - [Name of Trademark] is a registered trademark of Ansible, Inc. in the United States and other countries.

For Unregistered Trademarks (TMs/SMs): - [Name of Trademark] is a trademark of Ansible, Inc. in the United States and other countries.

For registered and unregistered trademarks: - [Name of Trademark] is a registered trademark and [Name of Trademark] is a trademark of Ansible, Inc. in the United States and other countries.

### Guidelines for the proper use of trademarks:

Always distinguish trademarks from surround text with at least initial capital letters or in all capital letters.

Always use proper trademark form and spelling.

Never use a trademark as a noun. Always use a trademark as an adjective modifying the noun.

Correct: Ansible Tower® system performance is incredible.

Incorrect: Ansible's performance is incredible.

Never use a trademark as a verb. Trademarks are products or services, never actions.

Correct: "Orchestrate your entire network using Ansible Tower®."

Incorrect: "Ansible your entire network."

Never modify a trademark to a plural form. Instead, change the generic word from the singular to the plural.

Correct: "Corporate demand for Ansible Tower® configuration software is surging."

Incorrect: "Corporate demand for Ansible is surging."

Never modify a trademark from its possessive form, or make a trademark possessive. Always use it in the form it has been registered.

Never translate a trademark into another language.

Never use trademarks to coin new words or names.

Never use trademarks to create a play on words.

Never alter a trademark in any way including through unapproved fonts or visual identifiers.

Never abbreviate or use any Ansible trademarks as an acronym.

### The importance of Ansible trademarks

The Ansible trademark and the "A" logo in a shaded circle are our most valuable assets. The value of these trademarks encompass the Ansible Brand. Effective trademark use is more than just a name, it defines the level of quality the customer will receive and it ties a product or service to a corporate image. A trademark may serve as the basis for many of our everyday decisions and choices. The Ansible Brand is about how we treat customers and each other. In order to continue to build a stronger more valuable Brand we must use it in a clear and consistent manner.

The mark consists of the letter "A" in a shaded circle. As of 5/11/15, this was a pending trademark (registration in process).

## Common Ansible Trademarks

- Ansible®
- Ansible Tower®

## Other Common Trademarks and Resource Sites:

- Linux is a registered trademark of Linus Torvalds.
- UNIX® is a registered trademark of The Open Group.
- Microsoft, Windows, Vista, XP, and NT are registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. <https://www.microsoft.com/en-us/legal/intellectualproperty/trademarks/en-us.aspx>
- Apple, Mac, Mac OS, Macintosh, Pages and TrueType are either registered trademarks or trademarks of Apple Computer, Inc. in the United States and/or other countries. <https://www.apple.com/legal/intellectual-property/trademark/appletmlist.html>
- Adobe, Acrobat, GoLive, InDesign, Illustrator, PostScript , PhotoShop and the OpenType logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. <http://www.adobe.com/legal/permissions/trademarks.html>
- Macromedia and Macromedia Flash are trademarks of Macromedia, Inc. <http://www.adobe.com/legal/permissions/trademarks.html>
- IBM is a registered trademark of International Business Machines Corporation. <https://www.ibm.com/legal/us/en/copytrade.shtml>
- Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, Intel Core, Intel Inside, Intel Inside logo, Itanium, Itanium Inside, Pentium, Pentium Inside, VTune, Xeon, and Xeon Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. <https://www.intel.com/content/www/us/en/legal/trademarks.html>

## Grammar and Punctuation

### Common Styles and Usage, and Common Mistakes

#### Ansible

- Write "Ansible." Not "Ansible, Inc." or "AnsibleWorks" The only exceptions to this rule are when we're writing legal or financial statements.
- Never use the logotype by itself in body text. Always keep the same font you are using the rest of the sentence.
- A company is singular in the US. In other words, Ansible is an "it," not a "they."

#### Capitalization

If it's not a real product, service, or department at Ansible, don't capitalize it. Not even if it seems important. Capitalize only the first letter of the first word in headlines.

### Colon

A colon is generally used before a list or series: - The Triangle Area consists of three cities: Raleigh, Durham, and Chapel Hill.

But not if the list is a complement or object of an element in the sentence: - Before going on vacation, be sure to (1) set the alarm, (2) cancel the newspaper, and (3) ask a neighbor to collect your mail.

Use a colon after "as follows" and "the following" if the related list comes immediately after: wedge The steps for changing directories are as follows:

1. Open a terminal.
2. Type cd...

Use a colon to introduce a bullet list (or dash, or icon/symbol of your choice):

In the Properties dialog box, you'll find the following entries:

- Connection name
- Count
- Cost per item

### Commas

Use serial commas, the comma before the "and" in a series of three or more items:

- "Item 1, item 2, and item 3."

It's easier to read that way and helps avoid confusion. The primary exception to this you will see is in PR, where it is traditional not to use serial commas because it is often the style of journalists.

Commas are always important, considering the vast difference in meanings of the following two statements.

- Let's eat, Grandma
- Let's eat Grandma.

Correct punctuation could save Grandma's life.

If that does not convince you, maybe this will:





## Contractions

Do not use contractions in Ansible documents.

## Em dashes

When possible, use em-dashes with no space on either side. When full em-dashes aren't available, use double-dashes with no spaces on either side—like this.

A pair of em dashes can be used in place of commas to enhance readability. Note, however, that dashes are always more emphatic than commas.

A pair of em dashes can replace a pair of parentheses. Dashes are considered less formal than parentheses; they are also more intrusive. If you want to draw attention to the parenthetical content, use dashes. If you want to include the parenthetical content more subtly, use parentheses.

: When dashes are used in place of parentheses, surrounding punctuation should be omitted. Compare the following examples.

---

```
Upon discovering the errors (all 124 of them), the publisher immediately recalled the ↵  
↵books.
```

```
Upon discovering the errors--all 124 of them--the publisher immediately recalled the ↵  
↵books.
```

When used in place of parentheses at the end of a sentence, only a single dash is used.

```
After three weeks on set, the cast was fed up with his direction (or, rather, lack of ↵  
↵direction).
```

```
After three weeks on set, the cast was fed up with his direction--or, rather, lack of ↵  
↵direction.
```

### Exclamation points (!)

Do not use them at the end of sentences. An exclamation point can be used when referring to a command, such as the bang (!) command.

### Gender References

Do not use gender-specific pronouns in documentation. It is far less awkward to read a sentence that uses "they" and "their" rather than "he/she" and "his/hers."

It is fine to use "you" when giving instructions and "the user," "new users," etc. in more general explanations.

Never use "one" in place of "you" when writing technical documentation. Using "one" is far too formal.

Never use "we" when writing. "We" aren't doing anything on the user side. Ansible's products are doing the work as requested by the user.

### Hyphen

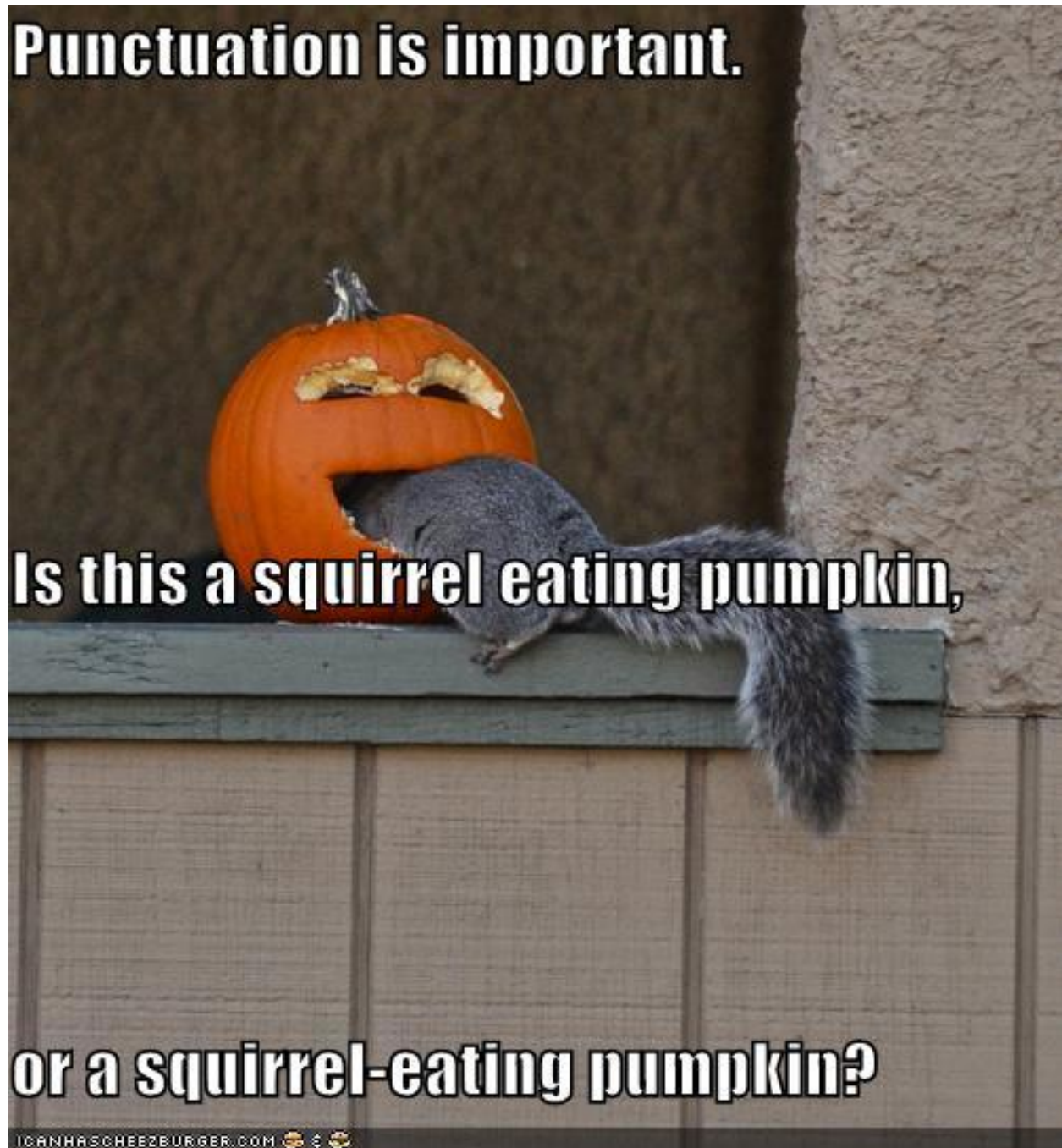
The hyphen's primary function is the formation of certain compound terms. Do not use a hyphen unless it serves a purpose. If a compound adjective cannot be misread or, as with many psychological terms, its meaning is established, a hyphen is not necessary.

Use hyphens to avoid ambiguity or confusion:

```
a little-used car  
a little used-car  
  
cross complaint  
cross-complaint  
  
high-school girl  
high schoolgirl  
  
fine-tooth comb (most people do not comb their teeth)
```

(continues on next page)

```
third-world war  
third world war
```



In professionally printed material (particularly books, magazines, and newspapers), the hyphen is used to divide words between the end of one line and the beginning of the next. This allows for an evenly aligned right margin without highly variable (and distracting) word spacing.

### Lists

Keep the structure of bulleted lists equivalent and consistent. If one bullet is a verb phrase, they should all be verb phrases. If one is a complete sentence, they should all be complete sentences, etc.

Capitalize the first word of each bullet. Unless it is obvious that it is just a list of items, such as a list of items like: \* computer \* monitor \* keyboard \* mouse

When the bulleted list appears within the context of other copy, (unless it's a straight list like the previous example) add periods, even if the bullets are sentence fragments. Part of the reason behind this is that each bullet is said to complete the original sentence.

In some cases where the bullets are appearing independently, such as in a poster or a homepage promotion, they do not need periods.

When giving instructional steps, use numbered lists instead of bulleted lists.

### Months and States

Abbreviate months and states according to AP. Months are only abbreviated if they are used in conjunction with a day. Example: "The President visited in January 1999." or "The President visited Jan. 12."

Months: Jan., Feb., March, April, May, June, July, Aug., Sept., Nov., Dec.

States: Ala., Ariz., Ark., Calif., Colo., Conn., Del., Fla., Ga., Ill., Ind., Kan., Ky., La., Md., Mass., Mich., Minn., Miss., Mo., Mont., Neb., Nev., NH, NJ, NM, NY, NC, ND, Okla., Ore., Pa., RI, SC, SD, Tenn., Vt., Va., Wash., W.Va., Wis., Wyo.

### Numbers

Numbers between one and nine are written out. 10 and above are numerals. The exception to this is writing "4 million" or "4 GB." It's also acceptable to use numerals in tables and charts.

### Phone Numbers

Phone number style: 1 (919) 555-0123 x002 and 1 888-GOTTEXT

### Quotations (Using Quotation Marks and Writing Quotes)

"Place the punctuation inside the quotes," the editor said.

Except in rare instances, use only "said" or "says" because anything else just gets in the way of the quote itself, and also tends to editorialize.

**Place the name first right after the quote:** "I like to write first-person because I like to become the character I'm writing," Wally Lamb said.

**Not:** "I like to write first-person because I like to become the character I'm writing," said Wally Lamb.

### Semicolon

Use a semicolon to separate items in a series if the items contain commas:

- Everyday I have coffee, toast, and fruit for breakfast; a salad for lunch; and a peanut butter sandwich, cookies, ice cream, and chocolate cake for dinner.

Use a semicolon before a conjunctive adverb (however, therefore, otherwise, namely, for example, etc.): - I think; therefore, I am.

### Spacing after sentences

Use only a single space after a sentence.

### Time

- Time of day is written as "4 p.m."

## Spelling - Word Usage - Common Words and Phrases to Use and Avoid

### Acronyms

Always uppercase. An acronym is a word formed from the initial letters of a name, such as ROM for Read-only memory, SaaS for Software as a Service, or by combining initial letters or part of a series of words, such as LILO for Linux LOader.

Spell out the acronym before using it in alone text, such as "The Embedded DevKit (EDK)..."

### Applications

When used as a proper name, use the capitalization of the product, such as GNUPro, Source-Navigator, and Ansible Tower. When used as a command, use lowercase as appropriate, such as "To start GCC, type `gcc`."

---

: "vi" is always lowercase.

---

### As

This is often used to mean "because", but has other connotations, for example, parallel or simultaneous actions. If you mean "because", say "because".

### Asks for

Use "requests" instead.

### Assure/Ensure/Insure

Assure implies a sort of mental comfort. As in "I assured my husband that I would eventually bring home beer."

Ensure means "to make sure."

Insure relates to monetary insurance.

### Back up

This is a verb. You "back up" files; you do not "backup" files.

### Backup

This is a noun. You create "backup" files; you do not create "back up" files.

### Backward

Correct. Avoid using backwards unless you are stating that something has "backwards compatibility."

### Backwards compatibility

Correct as is.

### By way of

Use "using" instead.

### Can/May

Use "can" to describe actions or conditions that are possible. Use "may" only to describe situations where permission is being given. If either "can," "could," or "may" apply, use "can" because it's less tentative.

### CD or cd

When referring to a compact disk, use CD, such as "Insert the CD into the CD-ROM drive." When referring to the change directory command, use cd.

### CD-ROM

Correct. Do not use "cdrom," "CD-Rom," "CDROM," "cd-rom" or any other variation. When referring to the drive, use CD-ROM drive, such as "Insert the CD into the CD-ROM drive." The plural is "CD-ROMs."

### Command line

Correct. Do not use "command-line" or "commandline."

Use to describes where to place options for a command, but not where to type the command. Use "shell prompt" instead to describe where to type commands. The line on the display screen where a command is expected. Generally, the command line is the line that contains the most recently displayed command prompt.



### Daylight saving time (DST)

Correct. Do not use daylight savings time. Daylight Saving Time (DST) is often misspelled "Daylight Savings", with an "s" at the end. Other common variations are "Summer Time" and "Daylight-Saving Time". (<http://www.timeanddate.com/time/dst/daylight-savings-time.html>)

### Download

Correct. Do not use "down load" or "down-load."

### e.g.

Spell it out: "For example."

### Failover

When used as a noun, a failover is a backup operation that automatically switches to a standby database, server or network if the primary system fails or is temporarily shut down for servicing. Failover is an important fault tolerance function of mission-critical systems that rely on constant accessibility. Failover automatically and transparently to the user redirects requests from the failed or down system to the backup system that mimics the operations of the primary system.

### Fail over

When used as a verb, fail over is two words since there can be different tenses such as failed over.

### Fewer

Fewer is used with plural nouns. Think things you could count. Time, money, distance, and weight are often listed as exceptions to the traditional "can you count it" rule, often thought of as singular amounts (the work will take less than 5 hours, for example).

### File name

Correct. Do not use "filename."

### File system

Correct. Do not use "filesystem." The system that an operating system or program uses to organize and keep track of files. For example, a hierarchical file system is one that uses directories to organize files into a tree structure. Although the operating system provides its own file management system, you can buy separate file management systems. These systems interact smoothly with the operating system but provide more features, such as improved backup procedures and stricter file protection.

### For instance

For example," instead.

### For further/additional/whatever information

Use "For more information"

### For this reason

Use "therefore".

### Forward

Correct. Avoid using "forwards."

### Gigabyte (GB)

2 to the 30th power (1,073,741,824) bytes. One gigabyte is equal to 1,024 megabytes. Gigabyte is often abbreviated as G or GB.

### Got

Avoid. Use "must" instead.

### High-availability

Correct. Do not use "high availability."

### Highly available

Correct. Do not use highly-available."

### Hostname

Correct. Do not use host name.

### i.e.

Spell it out: "That is."

### Installer

Avoid. Use "installation program" instead.



## It's and its

"It's" is a contraction for "it is;" use "it is" instead of "it's." Use "its" as a possessive pronoun (for example, "the store is known for its low prices").

## Less

Less is used with singular nouns. For example "View less details" wouldn't be correct but "View less detail" works. Use fewer when you have plural nouns (things you can count).

## Linux

Correct. Do not use "LINUX" or "linux" unless referring to a command, such as "To start Linux, type linux." Linux is a registered trademark of Linus Torvalds.

## Login

A noun used to refer to the login prompt, such as "At the login prompt, enter your username."

## Log in

A verb used to refer to the act of logging in. Do not use "login," "loggin," "logon," and other variants. For example, "When starting your computer, you are requested to log in..."

## Log on

To make a computer system or network recognize you so that you can begin a computer session. Most personal computers have no log-on procedure – you just turn the machine on and begin working. For larger systems and networks, however, you usually need to enter a username and password before the computer system will allow you to execute programs.

## Lots of

Use "Several" or something equivalent instead.

## Make sure

This means "be careful to remember, attend to, or find out something." For example, "...make sure that the rhedk group is listed in the output." Try to use verify or ensure instead.

## Manual/man page

Correct. Two words. Do not use "manpage"

### MB

1. When spelled MB, short for megabyte (1,000,000 or 1,048,576 bytes, depending on the context).
2. When spelled Mb, short for megabit.

### MBps

Short for megabytes per second, a measure of data transfer speed. Mass storage devices are generally measured in MBps.

### MySQL

Common open source database server and client package. Do not use "MYSQL" or "mySQL."

### Need to

Avoid. Use "must" instead.

### Read-only

Correct. Use when referring to the access permissions of files or directories.

### Real time/real-time

Depends. If used as a noun, it is the actual time during which something takes place. For example, "The computer may partly analyze the data in real time (as it comes in) – R. H. March." If used as an adjective, "real-time" is appropriate. For example, "XEmacs is a self-documenting, customizable, extensible, real-time display editor."

### Refer to

Use to indicate a reference (within a manual or website) or a cross-reference (to another manual or documentation source).

### See

Don't use. Use "Refer to" instead.

### Since

This is often used to mean "because", but "since" has connotations of time, so be careful. If you mean "because", say "because".

### Tells

Use "Instructs" instead.

## That/which

"That" introduces a restrictive clause—a clause that must be there for the sentence to make sense. A restrictive clause often defines the noun or phrase preceding it. "Which" introduces a non-restrictive, parenthetical clause—a clause that could be omitted without affecting the meaning of the sentence. For example: The car was travelling at a speed that would endanger lives. The car, which was traveling at a speed that would endanger lives, swerved onto the sidewalk. Use "who" or "whom," rather than "that" or "which," when referring to a person.

## Then/than

"Then" refers to a time in the past or the next step in a sequence. "Than" is used for comparisons.

# Then

is used for time.

*First I stole a panda bear, then we drank malt liquor together.*

The sequence of actions indicates time: first stealing the panda, and then drinking.

# Than

is used for comparison.

*I'm much better at holding my liquor than a panda bear.*

This is comparing a panda's drinking ability with your own, so you should use "than."



## Third-party

Correct. Do not use "third party".

## Troubleshoot

Correct. Do not use "trouble shoot" or "trouble-shoot." To isolate the source of a problem and fix it. In the case of computer systems, the term troubleshoot is usually used when the problem is suspected to be hardware -related. If the problem is known to be in software, the term debug is more commonly used.

**UK**

Correct as is, no periods.

**UNIX®**

Correct. Do not use "Unix" or "unix." UNIX® is a registered trademark of The Open Group.

**Unset**

Don't use. Use Clear.

**US**

Correct as is, no periods.

**User**

When referring to the reader, use "you" instead of "user." For example, "The user must..." is incorrect. Use "You must..." instead. If referring to more than one user, calling the collection "users" is acceptable, such as "Other users may wish to access your database."

**Username**

Correct. Do not use "user name."

**View**

When using as a reference ("View the documentation available online."), do not use View. Use "Refer to" instead.

**Within**

Don't use to refer to a file that exists in a directory. Use "In".

**World Wide Web**

Correct. Capitalize each word. Abbreviate as "WWW" or "Web."

**Webpage**

Correct. Do not use "web page" or "Web page."

**Web server**

Correct. Do not use "webserver". For example, "The Apache HTTP Server is the default Web server..."

**Website**

Correct. Do not use "web site" or "Web site." For example, "The Ansible website contains ..."

**Who/whom**

Use the pronoun "who" as a subject. Use the pronoun "whom" as a direct object, an indirect object, or the object of a preposition. For example: Who owns this? To whom does this belong?

**Will**

Do not use future tense unless it is absolutely necessary. For instance, do not use the sentence, "The next section will describe the process in more detail." Instead, use the sentence, "The next section describes the process in more detail."

**Wish**

Use "need" instead of "desire" and "wish." Use "want" when the reader's actions are optional (that is, they may not "need" something but may still "want" something).

**x86**

Correct. Do not capitalize the "x."

**x86\_64**

Do not use. Do not use "Hammer". Always use "AMD64 and Intel® EM64T" when referring to this architecture.

**You**

Correct. Do not use "I," "he," or "she."

**You may**

Try to avoid using this. For example, "you may" can be eliminated from this sentence "You may double-click on the desktop..."

## 1.7 Cisco ACI Guide

### 1.7.1 What is Cisco ACI ?

#### Application Centric Infrastructure (ACI)

The Cisco Application Centric Infrastructure (ACI) allows application requirements to define the network. This architecture simplifies, optimizes, and accelerates the entire application deployment life cycle.

#### Application Policy Infrastructure Controller (APIC)

The APIC manages the scalable ACI multi-tenant fabric. The APIC provides a unified point of automation and management, policy programming, application deployment, and health monitoring for the fabric. The APIC, which is implemented as a replicated synchronized clustered controller, optimizes performance, supports any application anywhere, and provides unified operation of the physical and virtual infrastructure.

The APIC enables network administrators to easily define the optimal network for applications. Data center operators can clearly see how applications consume network resources, easily isolate and troubleshoot application and infrastructure problems, and monitor and profile resource usage patterns.

The Cisco Application Policy Infrastructure Controller (APIC) API enables applications to directly connect with a secure, shared, high-performance resource pool that includes network, compute, and storage capabilities.

#### ACI Fabric

The Cisco Application Centric Infrastructure (ACI) Fabric includes Cisco Nexus 9000 Series switches with the APIC to run in the leaf/spine ACI fabric mode. These switches form a "fat-tree" network by connecting each leaf node to each spine node; all other devices connect to the leaf nodes. The APIC manages the ACI fabric.

The ACI fabric provides consistent low-latency forwarding across high-bandwidth links (40 Gbps, with a 100-Gbps future capability). Traffic with the source and destination on the same leaf switch is handled locally, and all other traffic travels from the ingress leaf to the egress leaf through a spine switch. Although this architecture appears as two hops from a physical perspective, it is actually a single Layer 3 hop because the fabric operates as a single Layer 3 switch.

The ACI fabric object-oriented operating system (OS) runs on each Cisco Nexus 9000 Series node. It enables programming of objects for each configurable element of the system. The ACI fabric OS renders policies from the APIC into a concrete model that runs in the physical infrastructure. The concrete model is analogous to compiled software; it is the form of the model that the switch operating system can execute.

All the switch nodes contain a complete copy of the concrete model. When an administrator creates a policy in the APIC that represents a configuration, the APIC updates the logical model. The APIC then performs the intermediate step of creating a fully elaborated policy that it pushes into all the switch nodes where the concrete model is updated.

The APIC is responsible for fabric activation, switch firmware management, network policy configuration, and instantiation. While the APIC acts as the centralized policy and network management engine for the fabric, it is completely removed from the data path, including the forwarding topology. Therefore, the fabric can still forward traffic even when communication with the APIC is lost.

#### More information

Various resources exist to start learning ACI, here is a list of interesting articles from the community.

- [Adam Raffe: Learning ACI](#)

- [Luca Relandini: ACI for dummies](#)
- [Cisco DevNet Learning Labs about ACI](#)

## 1.7.2 Using the ACI modules

The Ansible ACI modules provide a user-friendly interface to managing your ACI environment using Ansible play-books.

For instance ensuring that a specific tenant exists, is done using the following Ansible task using module `aci_tenant`:

```
- name: Ensure tenant customer-xyz exists
  aci_tenant:
    host: my-apic-1
    username: admin
    password: my-password

    tenant: customer-xyz
    description: Customer XYZ
    state: present
```

A complete list of existing ACI modules is available for the latest stable release on the list of network modules. You can also view the [current development version](#).

### Querying ACI configuration

A module can also be used to query a specific object.

```
- name: Query tenant customer-xyz
  aci_tenant:
    host: my-apic-1
    username: admin
    password: my-password

    tenant: customer-xyz
    state: query
```

Or query all objects.

```
- name: Query all tenants
  aci_tenant:
    host: my-apic-1
    username: admin
    password: my-password

    state: query
    register: all_tenants
```

After registering the return values of the `aci_tenant` task as shown above, you can access all tenant information from variable `all_tenants`.

### Common parameters

Every Ansible ACI module accepts the following parameters that influence the module's communication with the APIC REST API:

**host** Hostname or IP address of the APIC.

**port** Port to use for communication. (Defaults to 443 for HTTPS, and 80 for HTTP)

**username** User name used to log on to the APIC. (Defaults to `admin`)

**password** Password for `username` to log on to the APIC, using password-based authentication.

**private\_key** Private key for `username` to log on to APIC, using signature-based authentication. *New in version 2.5*

**certificate\_name** Name of the certificate in the ACI Web GUI. (Defaults to `private_key` file base name) *New in version 2.5*

**timeout** Timeout value for socket-level communication.

**use\_proxy** Use system proxy settings. (Defaults to `yes`)

**use\_ssl** Use HTTPS or HTTP for APIC REST communication. (Defaults to `yes`)

**validate\_certs** Validate certificate when using HTTPS communication. (Defaults to `yes`)

**output\_level** Influence the level of detail ACI modules return to the user. (One of `normal`, `info` or `debug`) *New in version 2.5*

## Proxy support

By default, if an environment variable `<protocol>_proxy` is set on the target host, requests will be sent through that proxy. This behaviour can be overridden by setting a variable for this task (see [Setting the Environment \(and Working With Proxies\)](#)), or by using the `use_proxy` module parameter.

HTTP redirects can redirect from HTTP to HTTPS so you should be sure that your proxy environment for both protocols is correct.

If you don't need proxy support, but the system may have it configured nevertheless, you can add this parameter setting: `use_proxy: no` to avoid accidental proxy usage.

---

: Selective proxy support using the `no_proxy` environment variable is also supported.

---

## Return values

2.5 .

The following values are always returned:

**current** The resulting state of the managed object, or results of your query.

The following values are returned when `output_level: info`:

**previous** The original state of the managed object (before any change was made).

**proposed** The proposed config payload, based on user-supplied values.

**sent** The sent config payload, based on user-supplied values and the existing configuration.

The following values are returned when `output_level: debug` or `ANSIBLE_DEBUG=1`:

**filter\_string** The filter used for specific APIC queries.

**method** The HTTP method used for the sent payload. (Either `GET` for queries, `DELETE` or `POST` for changes)



**response** The HTTP response from the APIC.

**status** The HTTP status code for the request.

**url** The url used for the request.

---

: The module return values are documented in detail as part of each module's documentation.

---

## More information

Various resources exist to start learn more about ACI programmability, we recommend the following links:

- [Jacob McGill: Automating Cisco ACI with Ansible](#)
- [Cisco DevNet Learning Labs about ACI and Ansible](#)

### 1.7.3 ACI authentication

#### Password-based authentication

If you want to log on using a username and password, you can use the following parameters with your ACI modules:

```
username: admin
password: my-password
```

Password-based authentication is very simple to work with, but it is not the most efficient form of authentication from ACI's point-of-view as it requires a separate login-request and an open session to work. To avoid having your session time-out and requiring another login, you can use the more efficient Signature-based authentication.

---

: Password-based authentication also may trigger anti-DoS measures in ACI v3.1+ that causes session throttling and results in HTTP 503 errors and login failures.

---

: Never store passwords in plain text.

The "Vault" feature of Ansible allows you to keep sensitive data such as passwords or keys in encrypted files, rather than as plain text in your playbooks or roles. These vault files can then be distributed or placed in source control. See [Using Vault in playbooks](#) for more information.

#### Signature-based authentication using certificates

2.5 .

Using signature-based authentication is more efficient and more reliable than password-based authentication.

#### Generate certificate and private key

Signature-based authentication requires a (self-signed) X.509 certificate with private key, and a configuration step for your AAA user in ACI. To generate a working X.509 certificate and private key, use the following procedure:

```
$ openssl req -new -newkey rsa:1024 -days 36500 -nodes -x509 -keyout admin.key -out_
↪admin.crt -subj '/CN=Admin/O=Your Company/C=US'
```

## Configure your local user

Perform the following steps:

- Add the X.509 certificate to your ACI AAA local user at *ADMIN » AAA*
- Click *AAA Authentication*
- Check that in the *Authentication* field the *Realm* field displays *Local*
- Expand *Security Management » Local Users*
- Click the name of the user you want to add a certificate to, in the *User Certificates* area
- Click the + sign and in the *Create X509 Certificate* enter a certificate name in the *Name* field
  - If you use the basename of your private key here, you don't need to enter `certificate_name` in Ansible
- Copy and paste your X.509 certificate in the *Data* field.

You can automate this by using the following Ansible task:

```
- name: Ensure we have a certificate installed
  aci_aaa_user_certificate:
    host: my-apic-1
    username: admin
    password: my-password

    aaa_user: admin
    certificate_name: admin
    certificate: "{{ lookup('file', 'pki/admin.crt') }}" # This will read the_
↪certificate data from a local file
```

---

: Signature-based authentication only works with local users.

---

## Use signature-based authentication with Ansible

You need the following parameters with your ACI module(s) for it to work:

```
username: admin
private_key: pki/admin.key
certificate_name: admin # This could be left out !
```

---

: If you use a certificate name in ACI that matches the private key's basename, you can leave out the `certificate_name` parameter like the example above.

---

## More information

Detailed information about Signature-based Authentication is available from [Cisco APIC Signature-Based Transactions](#).

### 1.7.4 Using ACI REST with Ansible

While already a lot of ACI modules exists in the Ansible distribution, and the most common actions can be performed with these existing modules, there's always something that may not be possible with off-the-shelf modules.

The `aci_rest` module provides you with direct access to the APIC REST API and enables you to perform any task not already covered by the existing modules. This may seem like a complex undertaking, but you can generate the needed REST payload for any action performed in the ACI web interface effortlessly.

#### Built-in idempotency

Because the APIC REST API is intrinsically idempotent and can report whether a change was made, the `aci_rest` module automatically inherits both capabilities and is a first-class solution for automating your ACI infrastructure. As a result, users that require more powerful low-level access to their ACI infrastructure don't have to give up on idempotency and don't have to guess whether a change was performed when using the `aci_rest` module.

#### Using the `aci_rest` module

The `aci_rest` module accepts the native XML and JSON payloads, but additionally accepts inline YAML payload (structured like JSON). The XML payload requires you to use a path ending with `.xml` whereas JSON or YAML require the path to end with `.json`.

When you're making modifications, you can use the POST or DELETE methods, whereas doing just queries require the GET method.

For instance, if you would like to ensure a specific tenant exists on ACI, these below four examples are functionally identical:

##### XML (Native ACI REST)

```
- aci_rest:
  host: my-apic-1
  private_key: pki/admin.key

  method: post
  path: /api/mo/uni.xml
  content: |
    <fvTenant name="customer-xyz" descr="Customer XYZ"/>
```

##### JSON (Native ACI REST)

```
- aci_rest:
  host: my-apic-1
  private_key: pki/admin.key

  method: post
  path: /api/mo/uni.json
  content:
    {
```

(continues on next page)

```
    "fvTenant": {
      "attributes": {
        "name": "customer-xyz",
        "descr": "Customer XYZ"
      }
    }
  }
}
```

### YAML (Ansible-style REST)

```
- aci_rest:
  host: my-apic-1
  private_key: pki/admin.key

  method: post
  path: /api/mo/uni.json
  content:
    fvTenant:
      attributes:
        name: customer-xyz
        descr: Customer XYZ
```

### Ansible task (Dedicated module)

```
- aci_tenant:
  host: my-apic-1
  private_key: pki/admin.key

  tenant: customer-xyz
  description: Customer XYZ
  state: present
```

---

: The XML format is more practical when there is a need to template the REST payload (inline), but the YAML format is more convenient for maintaining your infrastructure-as-code and feels more naturally integrated with Ansible playbooks. The dedicated modules offer a more simple, abstracted, but also a more limited experience. Use what feels best for your use-case.

---

### More information

Plenty of resources exist to learn about ACI's APIC REST interface, we recommend the links below:

- The `aci_rest` module documentation
- [APIC REST API Configuration Guide](#) – Detailed guide on how the APIC REST API is designed and used, incl. many examples
- [APIC Management Information Model reference](#) – Complete reference of the APIC object model
- [Cisco DevNet Learning Labs about ACI and REST](#)

## 1.7.5 Operational examples

Here is a small overview of useful operational tasks to reuse in your playbooks.

Feel free to contribute more useful snippets.

### Waiting for all controllers to be ready

You can use the below task after you started to build your APICs and configured the cluster to wait until all the APICs have come online. It will wait until the number of controllers equals the number listed in the apic inventory group.

```
- name: Waiting for all controllers to be ready
  aci_rest:
    host: '{{ apic_ip }}'
    private_key: pki/admin.key
    method: get
    path: /api/node/class/topSystem.json?query-target-filter=eq(topSystem.role,
↪"controller")
    register: topsystem
    until: topsystem|success and topsystem.totalCount|int >= groups['apic']|count >= 3
    retries: 20
    delay: 30
```

### Waiting for cluster to be fully-fit

The below example waits until the cluster is fully-fit. In this example you know the number of APICs in the cluster and you verify each APIC reports a 'fully-fit' status.

```
- name: Waiting for cluster to be fully-fit
  aci_rest:
    host: '{{ apic_ip }}'
    private_key: pki/admin.key
    method: get
    path: /api/node/class/infraWiNode.json?query-target-filter=wcard(infraWiNode.dn,
↪"topology/pod-1/node-1/av")
    register: infrawinode
    until: >
      infrawinode|success and
      infrawinode.totalCount|int >= groups['apic']|count >= 3 and
      infrawinode.imdata[0].infraWiNode.attributes.health == 'fully-fit' and
      infrawinode.imdata[1].infraWiNode.attributes.health == 'fully-fit' and
      infrawinode.imdata[2].infraWiNode.attributes.health == 'fully-fit'
  # all(apic.infraWiNode.attributes.health == 'fully-fit' for apic in infrawinode.
↪imdata)
  retries: 30
  delay: 30
```

## 1.7.6 APIC error messages

The following error messages may occur and this section can help you understand what exactly is going on and how to fix/avoid them.

**APIC Error 122: unknown managed object class 'polUni'** In case you receive this error while you are certain your aci\_rest payload and object classes are seemingly correct, the issue might be that your payload is not in fact correct JSON (e.g. the sent payload is using single quotes, rather than double quotes), and as a result the APIC is not correctly parsing your object classes from the payload. One way to avoid this is by using a YAML or an XML formatted payload, which are easier to construct correctly and modify later.

**APIC Error 400: invalid data at line '1'. Attributes are missing, tag 'attributes' must be specified first, before any other ta**

Although the JSON specification allows unordered elements, the APIC REST API requires that the JSON `attributes` element precede the `children` array or other elements. So you need to ensure that your payload conforms to this requirement. Sorting your dictionary keys will do the trick just fine. If you don't have any attributes, it may be necessary to add: `attributes: {}` as the APIC does expect the entry to precede any `children`.

**APIC Error 801: property descr of uni/tn-TENANT/ap-AP failed validation for value 'A "legacy" network'**

Some values in the APIC have strict format-rules to comply to, and the internal APIC validation check for the provided value failed. In the above case, the `description` parameter (internally known as `descr`) only accepts values conforming to `Regex: [a-zA-Z0-9\!#$%()*,-./:;@_{}~?&+]+`, in general it must not include quotes or square brackets.

## 1.7.7 Known issues

The `aci_rest` module is a wrapper around the APIC REST API. As a result any issues related to the APIC will be reflected in the use of this module.

All below issues either have been reported to the vendor, and most can simply be avoided.

**Too many consecutive API calls may result in connection throttling** Starting with ACI v3.1 the APIC will actively throttle password-based authenticated connection rates over a specific threshold. This is as part of an anti-DDOS measure but can act up when using Ansible with ACI using password-based authentication. Currently, one solution is to increase this threshold within the `nginx` configuration, but using signature-based authentication is recommended.

**NOTE:** It is advisable to use signature-based authentication with ACI as it not only prevents connection-throttling, but also improves general performance when using the ACI modules.

**Specific requests may not reflect changes correctly (#35401)** There is a known issue where specific requests to the APIC do not properly reflect changed in the resulting output, even when we request those changes explicitly from the APIC. In one instance using the path `api/node/mo/uni/infra.xml` fails, where `api/node/mo/uni/infra/.xml` does work correctly.

**NOTE:** A workaround is to register the task return values (e.g. `register: this`) and influence when the task should report a change by adding: `changed_when: this.imdata != []`.

**Specific requests are known to not be idempotent (#35050)** The behaviour of the APIC is inconsistent to the use of `status="created"` and `status="deleted"`. The result is that when you use `status="created"` in your payload the resulting tasks are not idempotent and creation will fail when the object was already created. However this is not the case with `status="deleted"` where such call to a non-existing object does not cause any failure whatsoever.

**NOTE:** A workaround is to avoid using `status="created"` and instead use `status="modified"` when idempotency is essential to your workflow..

**Setting user password is not idempotent (#35544)** Due to an inconsistency in the APIC REST API, a task that sets the password of a locally-authenticated user is not idempotent. The APIC will complain with message `Password history check: user dag should not use previous 5 passwords`.

**NOTE:** There is no workaround for this issue.

## 1.7.8 ACI Ansible community

If you have specific issues with the ACI modules, or a feature request, or you like to contribute to the ACI project by proposing changes or documentation updates, look at the Ansible Community wiki ACI page at: <https://github.com/ansible/community/wiki/Network:-ACI>

You will find our roadmap, an overview of open ACI issues and pull-requests and more information about who we are. If you have an interest in using ACI with Ansible, feel free to join ! We occasionally meet online to track progress and prepare for new Ansible releases.

:

***Ansible for Network Automation*** A detailed guide on how to use Ansible for automating network infrastructure.

**List of ACI modules** A complete list of supported ACI modules.

**ACI community** The Ansible ACI community wiki page, includes roadmap, ideas and development documentation.

**Network Working Group** The Ansible Network community page, includes contact information and meeting information.

**#ansible-network** The #ansible-network IRC chat channel on Freenode.net.

**User Mailing List** Have a question? Stop by the google group!

## 1.8 Amazon Web Services Guide

### 1.8.1 Introduction

Ansible contains a number of modules for controlling Amazon Web Services (AWS). The purpose of this section is to explain how to put Ansible modules together (and use inventory scripts) to use Ansible in AWS context.

Requirements for the AWS modules are minimal.

All of the modules require and are tested against recent versions of boto. You'll need this Python module installed on your control machine. Boto can be installed from your OS distribution or python's "pip install boto".

Whereas classically ansible will execute tasks in its host loop against multiple remote machines, most cloud-control steps occur on your local machine with reference to the regions to control.

In your playbook steps we'll typically be using the following pattern for provisioning steps:

```
- hosts: localhost
  connection: local
  gather_facts: False
  tasks:
    - ...
```

### 1.8.2 Authentication

Authentication with the AWS-related modules is handled by either specifying your access and secret key as ENV variables or module arguments.

For environment variables:

```
export AWS_ACCESS_KEY_ID='AK123'
export AWS_SECRET_ACCESS_KEY='abc123'
```

For storing these in a vars\_file, ideally encrypted with ansible-vault:

```
---
ec2_access_key: "--REMOVED--"
ec2_secret_key: "--REMOVED--"
```

Note that if you store your credentials in vars\_file, you need to refer to them in each AWS-module. For example:

```
- ec2
  aws_access_key: "{{ec2_access_key}}"
  aws_secret_key: "{{ec2_secret_key}}"
  image: "..."
```

### 1.8.3 Provisioning

The ec2 module provisions and de-provisions instances within EC2.

An example of making sure there are only 5 instances tagged 'Demo' in EC2 follows.

In the example below, the "exact\_count" of instances is set to 5. This means if there are 0 instances already existing, then 5 new instances would be created. If there were 2 instances, only 3 would be created, and if there were 8 instances, 3 instances would be terminated.

What is being counted is specified by the "count\_tag" parameter. The parameter "instance\_tags" is used to apply tags to the newly created instance.:

```
# demo_setup.yml

- hosts: localhost
  connection: local
  gather_facts: False

  tasks:

    - name: Provision a set of instances
      ec2:
        key_name: my_key
        group: test
        instance_type: t2.micro
        image: "{{ ami_id }}"
        wait: true
        exact_count: 5
        count_tag:
          Name: Demo
        instance_tags:
          Name: Demo
      register: ec2
```

The data about what instances are created is being saved by the "register" keyword in the variable named "ec2".

From this, we'll use the add\_host module to dynamically create a host group consisting of these new instances. This facilitates performing configuration actions on the hosts immediately in a subsequent task.:

```
# demo_setup.yml

- hosts: localhost
  connection: local
```

(continues on next page)



()

```
gather_facts: False

tasks:

  - name: Provision a set of instances
    ec2:
      key_name: my_key
      group: test
      instance_type: t2.micro
      image: "{{ ami_id }}"
      wait: true
      exact_count: 5
      count_tag:
        Name: Demo
      instance_tags:
        Name: Demo
      register: ec2

  - name: Add all instance public IPs to host group
    add_host: hostname={{ item.public_ip }} groups=ec2hosts
    loop: "{{ ec2.instances }}"
```

With the host group now created, a second play at the bottom of the same provisioning playbook file might now have some configuration steps:

```
# demo_setup.yml

- name: Provision a set of instances
  hosts: localhost
  # ... AS ABOVE ...

- hosts: ec2hosts
  name: configuration play
  user: ec2-user
  gather_facts: true

  tasks:

    - name: Check NTP service
      service: name=ntpd state=started
```

## 1.8.4 Host Inventory

Once your nodes are spun up, you'll probably want to talk to them again. With a cloud setup, it's best to not maintain a static list of cloud hostnames in text files. Rather, the best way to handle this is to use the ec2 dynamic inventory script. See *Working With Dynamic Inventory*.

This will also dynamically select nodes that were even created outside of Ansible, and allow Ansible to manage them. See *Working With Dynamic Inventory* for how to use this, then return to this chapter.

## 1.8.5 Tags And Groups And Variables

When using the ec2 inventory script, hosts automatically appear in groups based on how they are tagged in EC2.

For instance, if a host is given the "class" tag with the value of "webserver", it will be automatically discoverable via a dynamic group like so:

```
- hosts: tag_class_webserver
  tasks:
    - ping
```

Using this philosophy can be a great way to keep systems separated by the function they perform.

In this example, if we wanted to define variables that are automatically applied to each machine tagged with the 'class' of 'webserver', 'group\_vars' in ansible can be used. See *Splitting Out Host and Group Specific Data*.

Similar groups are available for regions and other classifications, and can be similarly assigned variables using the same mechanism.

### 1.8.6 Autoscaling with Ansible Pull

Amazon Autoscaling features automatically increase or decrease capacity based on load. There are also Ansible modules shown in the cloud documentation that can configure autoscaling policy.

When nodes come online, it may not be sufficient to wait for the next cycle of an ansible command to come along and configure that node.

To do this, pre-bake machine images which contain the necessary ansible-pull invocation. Ansible-pull is a command line tool that fetches a playbook from a git server and runs it locally.

One of the challenges of this approach is that there needs to be a centralized way to store data about the results of pull commands in an autoscaling context. For this reason, the autoscaling solution provided below in the next section can be a better approach.

Read `ansible-pull` for more information on pull-mode playbooks.

### 1.8.7 Autoscaling with Ansible Tower

*Ansible Tower* also contains a very nice feature for auto-scaling use cases. In this mode, a simple curl script can call a defined URL and the server will "dial out" to the requester and configure an instance that is spinning up. This can be a great way to reconfigure ephemeral nodes. See the Tower install and product documentation for more details.

A benefit of using the callback in Tower over pull mode is that job results are still centrally recorded and less information has to be shared with remote hosts.

### 1.8.8 Ansible With (And Versus) CloudFormation

CloudFormation is a Amazon technology for defining a cloud stack as a JSON document.

Ansible modules provide an easier to use interface than CloudFormation in many examples, without defining a complex JSON document. This is recommended for most users.

However, for users that have decided to use CloudFormation, there is an Ansible module that can be used to apply a CloudFormation template to Amazon.

When using Ansible with CloudFormation, typically Ansible will be used with a tool like Packer to build images, and CloudFormation will launch those images, or ansible will be invoked through user data once the image comes online, or a combination of the two.

Please see the examples in the Ansible CloudFormation module for more details.

### 1.8.9 AWS Image Building With Ansible

Many users may want to have images boot to a more complete configuration rather than configuring them entirely after instantiation. To do this, one of many programs can be used with Ansible playbooks to define and upload a base image, which will then get its own AMI ID for usage with the `ec2` module or other Ansible AWS modules such as `ec2_asg` or the `cloudformation` module. Possible tools include Packer, aminator, and Ansible's `ec2_ami` module.

Generally speaking, we find most users using Packer.

See the Packer documentation of the [Ansible local Packer provisioner](#) and [Ansible remote Packer provisioner](#).

If you do not want to adopt Packer at this time, configuring a base-image with Ansible after provisioning (as shown above) is acceptable.

### 1.8.10 Next Steps: Explore Modules

Ansible ships with lots of modules for configuring a wide array of EC2 services. Browse the "Cloud" category of the module documentation for a full list with examples.

:

**all\_modules** All the documentation for Ansible modules

*Working With Playbooks* An introduction to playbooks

*Delegation, Rolling Updates, and Local Actions* Delegation, useful for working with load balancers, clouds, and locally executed steps.

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## 1.9 Microsoft Azure Guide

Ansible includes a suite of modules for interacting with Azure Resource Manager, giving you the tools to easily create and orchestrate infrastructure on the Microsoft Azure Cloud.

### 1.9.1 Requirements

Using the Azure Resource Manager modules requires having specific Azure SDK modules installed on the host running Ansible.

```
$ pip install ansible[azure]
```

If you are running Ansible from source, you can install the dependencies from the root directory of the Ansible repo.

```
$ pip install .[azure]
```

You can also directly run Ansible in [Azure Cloud Shell](#), where Ansible is pre-installed.

### 1.9.2 Authenticating with Azure

Using the Azure Resource Manager modules requires authenticating with the Azure API. You can choose from two authentication strategies:

- Active Directory Username/Password
- Service Principal Credentials

Follow the directions for the strategy you wish to use, then proceed to *Providing Credentials to Azure Modules* for instructions on how to actually use the modules and authenticate with the Azure API.

### Using Service Principal

There is now a detailed official tutorial describing [how to create a service principal](#).

After stepping through the tutorial you will have:

- Your Client ID, which is found in the "client id" box in the "Configure" page of your application in the Azure portal
- Your Secret key, generated when you created the application. You cannot show the key after creation. If you lost the key, you must create a new one in the "Configure" page of your application.
- And finally, a tenant ID. It's a UUID (e.g. ABCDEFGH-1234-ABCD-1234-ABCDEFGHIJKL) pointing to the AD containing your application. You will find it in the URL from within the Azure portal, or in the "view endpoints" of any given URL.

### Using Active Directory Username/Password

To create an Active Directory username/password:

- Connect to the Azure Classic Portal with your admin account
- Create a user in your default AAD. You must NOT activate Multi-Factor Authentication
- Go to Settings - Administrators
- Click on Add and enter the email of the new user.
- Check the checkbox of the subscription you want to test with this user.
- Login to Azure Portal with this new user to change the temporary password to a new one. You will not be able to use the temporary password for OAuth login.

### Providing Credentials to Azure Modules

The modules offer several ways to provide your credentials. For a CI/CD tool such as Ansible Tower or Jenkins, you will most likely want to use environment variables. For local development you may wish to store your credentials in a file within your home directory. And of course, you can always pass credentials as parameters to a task within a playbook. The order of precedence is parameters, then environment variables, and finally a file found in your home directory.

### Using Environment Variables

To pass service principal credentials via the environment, define the following variables:

- AZURE\_CLIENT\_ID
- AZURE\_SECRET
- AZURE\_SUBSCRIPTION\_ID
- AZURE\_TENANT

To pass Active Directory username/password via the environment, define the following variables:

- AZURE\_AD\_USER
- AZURE\_PASSWORD

To pass Active Directory username/password in ADFS via the environment, define the following variables:

- AZURE\_AD\_USER
- AZURE\_PASSWORD
- AZURE\_CLIENT\_ID
- AZURE\_TENANT
- AZURE\_ADFS\_AUTHORITY\_URL

"AZURE\_ADFS\_AUTHORITY\_URL" is optional. It's necessary only when you have own ADFS authority like <https://xxx.com/adfs>.

## Storing in a File

When working in a development environment, it may be desirable to store credentials in a file. The modules will look for credentials in \$HOME/.azure/credentials. This file is an ini style file. It will look as follows:

```
[default]
subscription_id=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
client_id=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
secret=xxxxxxxxxxxxxxxxxxxx
tenant=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

It is possible to store multiple sets of credentials within the credentials file by creating multiple sections. Each section is considered a profile. The modules look for the [default] profile automatically. Define AZURE\_PROFILE in the environment or pass a profile parameter to specify a specific profile.

## Passing as Parameters

If you wish to pass credentials as parameters to a task, use the following parameters for service principal:

- client\_id
- secret
- subscription\_id
- tenant

Or, pass the following parameters for Active Directory username/password:

- ad\_user
- password

Or, pass the following parameters for ADFS username/password:

- ad\_user
- password
- client\_id
- tenant

- `adfs_authority_url`

"`adfs_authority_url`" is optional. It's necessary only when you have own ADFS authority like <https://xxx.com/adfs>.

### 1.9.3 Other Cloud Environments

To use an Azure Cloud other than the default public cloud (eg, Azure China Cloud, Azure US Government Cloud, Azure Stack), pass the "`cloud_environment`" argument to modules, configure it in a credential profile, or set the "`AZURE_CLOUD_ENVIRONMENT`" environment variable. The value is either a cloud name as defined by the Azure Python SDK (eg, "`AzureChinaCloud`", "`AzureUSGovernment`"; defaults to "`AzureCloud`") or an Azure meta-data discovery URL (for Azure Stack).

### 1.9.4 Creating Virtual Machines

There are two ways to create a virtual machine, both involving the `azure_rm_virtualmachine` module. We can either create a storage account, network interface, security group and public IP address and pass the names of these objects to the module as parameters, or we can let the module do the work for us and accept the defaults it chooses.

#### Creating Individual Components

An Azure module is available to help you create a storage account, virtual network, subnet, network interface, security group and public IP. Here is a full example of creating each of these and passing the names to the `azure_rm_virtualmachine` module at the end:

```
- name: Create storage account
  azure_rm_storageaccount:
    resource_group: Testing
    name: testaccount001
    account_type: Standard_LRS

- name: Create virtual network
  azure_rm_virtualnetwork:
    resource_group: Testing
    name: testvn001
    address_prefixes: "10.10.0.0/16"

- name: Add subnet
  azure_rm_subnet:
    resource_group: Testing
    name: subnet001
    address_prefix: "10.10.0.0/24"
    virtual_network: testvn001

- name: Create public ip
  azure_rm_publicipaddress:
    resource_group: Testing
    allocation_method: Static
    name: publicip001

- name: Create security group that allows SSH
  azure_rm_securitygroup:
    resource_group: Testing
    name: secgroup001
    rules:
```

(continues on next page)

()

```

- name: SSH
  protocol: Tcp
  destination_port_range: 22
  access: Allow
  priority: 101
  direction: Inbound

- name: Create NIC
  azure_rm_networkinterface:
    resource_group: Testing
    name: testnic001
    virtual_network: testvn001
    subnet: subnet001
    public_ip_name: publicip001
    security_group: secgroup001

- name: Create virtual machine
  azure_rm_virtualmachine:
    resource_group: Testing
    name: testvm001
    vm_size: Standard_D1
    storage_account: testaccount001
    storage_container: testvm001
    storage_blob: testvm001.vhd
    admin_username: admin
    admin_password: Password!
    network_interfaces: testnic001
    image:
      offer: CentOS
      publisher: OpenLogic
      sku: '7.1'
      version: latest

```

Each of the Azure modules offers a variety of parameter options. Not all options are demonstrated in the above example. See each individual module for further details and examples.

### Creating a Virtual Machine with Default Options

If you simply want to create a virtual machine without specifying all the details, you can do that as well. The only caveat is that you will need a virtual network with one subnet already in your resource group. Assuming you have a virtual network already with an existing subnet, you can run the following to create a VM:

```

azure_rm_virtualmachine:
  resource_group: Testing
  name: testvm10
  vm_size: Standard_D1
  admin_username: chouseknecht
  ssh_password: false
  ssh_public_keys: "{{ ssh_keys }}"
  image:
    offer: CentOS
    publisher: OpenLogic
    sku: '7.1'
    version: latest

```

## 1.9.5 Dynamic Inventory Script

If you are not familiar with Ansible's dynamic inventory scripts, check out *Intro to Dynamic Inventory*.

The Azure Resource Manager inventory script is called `azure_rm.py`. It authenticates with the Azure API exactly the same as the Azure modules, which means you will either define the same environment variables described above in *Using Environment Variables*, create a `$HOME/.azure/credentials` file (also described above in *Storing in a File*), or pass command line parameters. To see available command line options execute the following:

```
$ ./ansible/contrib/inventory/azure_rm.py --help
```

As with all dynamic inventory scripts, the script can be executed directly, passed as a parameter to the `ansible` command, or passed directly to `ansible-playbook` using the `-i` option. No matter how it is executed the script produces JSON representing all of the hosts found in your Azure subscription. You can narrow this down to just hosts found in a specific set of Azure resource groups, or even down to a specific host.

For a given host, the inventory script provides the following host variables:

```
{
  "ansible_host": "XXX.XXX.XXX.XXX",
  "computer_name": "computer_name2",
  "fqdn": null,
  "id": "/subscriptions/subscription-id/resourceGroups/galaxy-production/providers/
↪Microsoft.Compute/virtualMachines/object-name",
  "image": {
    "offer": "CentOS",
    "publisher": "OpenLogic",
    "sku": "7.1",
    "version": "latest"
  },
  "location": "westus",
  "mac_address": "00-00-5E-00-53-FE",
  "name": "object-name",
  "network_interface": "interface-name",
  "network_interface_id": "/subscriptions/subscription-id/resourceGroups/galaxy-
↪production/providers/Microsoft.Network/networkInterfaces/object-name1",
  "network_security_group": null,
  "network_security_group_id": null,
  "os_disk": {
    "name": "object-name",
    "operating_system_type": "Linux"
  },
  "plan": null,
  "powerstate": "running",
  "private_ip": "172.26.3.6",
  "private_ip_alloc_method": "Static",
  "provisioning_state": "Succeeded",
  "public_ip": "XXX.XXX.XXX.XXX",
  "public_ip_alloc_method": "Static",
  "public_ip_id": "/subscriptions/subscription-id/resourceGroups/galaxy-production/
↪providers/Microsoft.Network/publicIPAddresses/object-name",
  "public_ip_name": "object-name",
  "resource_group": "galaxy-production",
  "security_group": "object-name",
  "security_group_id": "/subscriptions/subscription-id/resourceGroups/galaxy-
↪production/providers/Microsoft.Network/networkSecurityGroups/object-name",
  "tags": {
    "db": "mysql"
  }
}
```

(continues on next page)



()

```

},
"type": "Microsoft.Compute/virtualMachines",
"virtual_machine_size": "Standard_DS4"
}

```

## Host Groups

By default hosts are grouped by:

- azure (all hosts)
- location name
- resource group name
- security group name
- tag key
- tag key\_value

You can control host groupings and host selection by either defining environment variables or creating an `azure_rm.ini` file in your current working directory.

NOTE: An `.ini` file will take precedence over environment variables.

NOTE: The name of the `.ini` file is the basename of the inventory script (i.e. `'azure_rm'`) with a `'ini'` extension. This allows you to copy, rename and customize the inventory script and have matching `.ini` files all in the same directory.

Control grouping using the following variables defined in the environment:

- `AZURE_GROUP_BY_RESOURCE_GROUP=yes`
- `AZURE_GROUP_BY_LOCATION=yes`
- `AZURE_GROUP_BY_SECURITY_GROUP=yes`
- `AZURE_GROUP_BY_TAG=yes`

Select hosts within specific resource groups by assigning a comma separated list to:

- `AZURE_RESOURCE_GROUPS=resource_group_a,resource_group_b`

Select hosts for specific tag key by assigning a comma separated list of tag keys to:

- `AZURE_TAGS=key1,key2,key3`

Select hosts for specific locations by assigning a comma separated list of locations to:

- `AZURE_LOCATIONS=eastus,eastus2,westus`

Or, select hosts for specific tag key:value pairs by assigning a comma separated list key:value pairs to:

- `AZURE_TAGS=key1:value1,key2:value2`

If you don't need the powerstate, you can improve performance by turning off powerstate fetching:

- `AZURE_INCLUDE_POWERSTATE=no`

A sample `azure_rm.ini` file is included along with the inventory script in `contrib/inventory`. An `.ini` file will contain the following:

```
[azure]
# Control which resource groups are included. By default all resources groups are
↳ included.
# Set resource_groups to a comma separated list of resource groups names.
#resource_groups=

# Control which tags are included. Set tags to a comma separated list of keys or
↳ key:value pairs
#tags=

# Control which locations are included. Set locations to a comma separated list of
↳ locations.
#locations=

# Include powerstate. If you don't need powerstate information, turning it off
↳ improves runtime performance.
# Valid values: yes, no, true, false, True, False, 0, 1.
include_powerstate=yes

# Control grouping with the following boolean flags. Valid values: yes, no, true,
↳ false, True, False, 0, 1.
group_by_resource_group=yes
group_by_location=yes
group_by_security_group=yes
group_by_tag=yes
```

## Examples

Here are some examples using the inventory script:

```
# Execute /bin/uname on all instances in the Testing resource group
$ ansible -i azure_rm.py Testing -m shell -a "/bin/uname -a"

# Use the inventory script to print instance specific information
$ ./ansible/contrib/inventory/azure_rm.py --host my_instance_host_name --resource-
↳ groups=Testing --pretty

# Use the inventory script with ansible-playbook
$ ansible-playbook -i ./ansible/contrib/inventory/azure_rm.py test_playbook.yml
```

Here is a simple playbook to exercise the Azure inventory script:

```
- name: Test the inventory script
  hosts: azure
  connection: local
  gather_facts: no
  tasks:
    - debug: msg="{{ inventory_hostname }}" has powerstate {{ powerstate }}
```

You can execute the playbook with something like:

```
$ ansible-playbook -i ./ansible/contrib/inventory/azure_rm.py test_azure_inventory.yml
```

## Disabling certificate validation on Azure endpoints

When an HTTPS proxy is present, or when using Azure Stack, it may be necessary to disable certificate validation for Azure endpoints in the Azure modules. This is not a recommended security practice, but may be necessary when the system CA store cannot be altered to include the necessary CA certificate. Certificate validation can be controlled by setting the "cert\_validation\_mode" value in a credential profile, via the "AZURE\_CERT\_VALIDATION\_MODE" environment variable, or by passing the "cert\_validation\_mode" argument to any Azure module. The default value is "validate"; setting the value to "ignore" will prevent all certificate validation. The module argument takes precedence over a credential profile value, which takes precedence over the environment value.

## 1.10 CloudStack Cloud Guide

### 1.10.1 Introduction

The purpose of this section is to explain how to put Ansible modules together to use Ansible in a CloudStack context. You will find more usage examples in the details section of each module.

Ansible contains a number of extra modules for interacting with CloudStack based clouds. All modules support check mode, are designed to be idempotent, have been created and tested, and are maintained by the community.

---

: Some of the modules will require domain admin or root admin privileges.

---

### 1.10.2 Prerequisites

Prerequisites for using the CloudStack modules are minimal. In addition to Ansible itself, all of the modules require the python library `cs` <https://pypi.org/project/cs/>

You'll need this Python module installed on the execution host, usually your workstation.

```
$ pip install cs
```

Or alternatively starting with Debian 9 and Ubuntu 16.04:

```
$ sudo apt install python-cs
```

---

: `cs` also includes a command line interface for ad-hoc interaction with the CloudStack API e.g. `$ cs listVirtualMachines state=Running`.

---

### 1.10.3 Limitations and Known Issues

VPC support has been improved since Ansible 2.3 but is still not yet fully implemented. The community is working on the VPC integration.

### 1.10.4 Credentials File

You can pass credentials and the endpoint of your cloud as module arguments, however in most cases it is a far less work to store your credentials in the `cloudstack.ini` file.

The python library `cs` looks for the credentials file in the following order (last one wins):

- A `.cloudstack.ini` (note the dot) file in the home directory.
- A `CLOUDSTACK_CONFIG` environment variable pointing to an `.ini` file.
- A `cloudstack.ini` (without the dot) file in the current working directory, same directory as your playbooks are located.

The structure of the ini file must look like this:

```
$ cat $HOME/.cloudstack.ini
[cloudstack]
endpoint = https://cloud.example.com/client/api
key = api key
secret = api secret
timeout = 30
```

---

: The section `[cloudstack]` is the default section. `CLOUDSTACK_REGION` environment variable can be used to define the default section.

---

### 2.4.

The ENV variables support `CLOUDSTACK_*` as written in the documentation of the library `cs`, like e.g `CLOUDSTACK_TIMEOUT`, `CLOUDSTACK_METHOD`, etc. has been implemented into Ansible. It is even possible to have some incomplete config in your `cloudstack.ini`:

```
$ cat $HOME/.cloudstack.ini
[cloudstack]
endpoint = https://cloud.example.com/client/api
timeout = 30
```

and fulfill the missing data by either setting ENV variables or tasks params:

```
---
- name: provision our VMs
  hosts: cloud-vm
  connection: local
  tasks:
    - name: ensure VMs are created and running
      cs_instance:
        api_key: your api key
        api_secret: your api secret
        ...
```

## 1.10.5 Regions

If you use more than one CloudStack region, you can define as many sections as you want and name them as you like, e.g.:

```
$ cat $HOME/.cloudstack.ini
[exoscale]
endpoint = https://api.exoscale.ch/compute
key = api key
secret = api secret
```

(continues on next page)

()

```
[exmaple_cloud_one]
endpoint = https://cloud-one.example.com/client/api
key = api key
secret = api secret

[exmaple_cloud_two]
endpoint = https://cloud-two.example.com/client/api
key = api key
secret = api secret
```

: Sections can also be used to for login into the same region using different accounts.

By passing the argument `api_region` with the CloudStack modules, the region wanted will be selected.

```
- name: ensure my ssh public key exists on Exoscale
  local_action: cs_sshkeypair
    name: my-ssh-key
    public_key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"
    api_region: exoscale
```

Or by looping over a regions list if you want to do the task in every region:

```
- name: ensure my ssh public key exists in all CloudStack regions
  local_action: cs_sshkeypair
    name: my-ssh-key
    public_key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"
    api_region: "{{ item }}"
  loop:
    - exoscale
    - exmaple_cloud_one
    - exmaple_cloud_two
```

## 1.10.6 Environment Variables

2.3.

Since Ansible 2.3 it is possible to use environment variables for domain (`CLOUDSTACK_DOMAIN`), account (`CLOUDSTACK_ACCOUNT`), project (`CLOUDSTACK_PROJECT`), VPC (`CLOUDSTACK_VPC`) and zone (`CLOUDSTACK_ZONE`). This simplifies the tasks by not repeating the arguments for every tasks.

Below you see an example how it can be used in combination with Ansible's block feature:

```
- hosts: cloud-vm
  tasks:
    - block:
        - name: ensure my ssh public key
          local_action:
            module: cs_sshkeypair
            name: my-ssh-key
            public_key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"

        - name: ensure my ssh public key
          local_action:
            module: cs_instance:
```

(continues on next page)

```

    display_name: "{{ inventory_hostname_short }}"
    template: Linux Debian 7 64-bit 20GB Disk
    service_offering: "{{ cs_offering }}"
    ssh_key: my-ssh-key
    state: running

environment:
    CLOUDSTACK_DOMAIN: root/customers
    CLOUDSTACK_PROJECT: web-app
    CLOUDSTACK_ZONE: sf-1

```

---

: You are still able to overwrite the environment variables using the module arguments, e.g. `zone: sf-2`

---



---

: Unlike `CLOUDSTACK_REGION` these additional environment variables are ignored in the CLI `cs`.

---

### 1.10.7 Use Cases

The following should give you some ideas how to use the modules to provision VMs to the cloud. As always, there isn't only one way to do it. But as always: keep it simple for the beginning is always a good start.

#### Use Case: Provisioning in a Advanced Networking CloudStack setup

Our CloudStack cloud has an advanced networking setup, we would like to provision web servers, which get a static NAT and open firewall ports 80 and 443. Further we provision database servers, to which we do not give any access to. For accessing the VMs by SSH we use a SSH jump host.

This is how our inventory looks like:

```

[cloud-vm:children]
webserver
db-server
jumphost

[webserver]
web-01.example.com  public_ip=198.51.100.20
web-02.example.com  public_ip=198.51.100.21

[db-server]
db-01.example.com
db-02.example.com

[jumphost]
jump.example.com  public_ip=198.51.100.22

```

As you can see, the public IPs for our web servers and jumphost has been assigned as variable `public_ip` directly in the inventory.

To configure the jumphost, web servers and database servers, we use `group_vars`. The `group_vars` directory contains 4 files for configuration of the groups: `cloud-vm`, `jumphost`, `webserver` and `db-server`. The `cloud-vm` is there for specifying the defaults of our cloud infrastructure.

```
# file: group_vars/cloud-vm
---
cs_offering: Small
cs_firewall: []
```

Our database servers should get more CPU and RAM, so we define to use a Large offering for them.

```
# file: group_vars/db-server
---
cs_offering: Large
```

The web servers should get a Small offering as we would scale them horizontally, which is also our default offering. We also ensure the known web ports are opened for the world.

```
# file: group_vars/webserver
---
cs_firewall:
  - { port: 80 }
  - { port: 443 }
```

Further we provision a jump host which has only port 22 opened for accessing the VMs from our office IPv4 network.

```
# file: group_vars/jumphost
---
cs_firewall:
  - { port: 22, cidr: "17.17.17.0/24" }
```

Now to the fun part. We create a playbook to create our infrastructure we call it `infra.yml`:

```
# file: infra.yml
---
- name: provision our VMs
  hosts: cloud-vm
  connection: local
  tasks:
    - name: ensure VMs are created and running
      cs_instance:
        name: "{{ inventory_hostname_short }}"
        template: Linux Debian 7 64-bit 20GB Disk
        service_offering: "{{ cs_offering }}"
        state: running

    - name: ensure firewall ports opened
      cs_firewall:
        ip_address: "{{ public_ip }}"
        port: "{{ item.port }}"
        cidr: "{{ item.cidr | default('0.0.0.0/0') }}"
        loop: "{{ cs_firewall }}"
        when: public_ip is defined

    - name: ensure static NATs
      cs_staticnat: vm="{{ inventory_hostname_short }}" ip_address="{{ public_ip }}"
        when: public_ip is defined
```

In the above play we defined 3 tasks and use the group `cloud-vm` as target to handle all VMs in the cloud but instead SSH to these VMs, we use `connection=local` to execute the API calls locally from our workstation.

In the first task, we ensure we have a running VM created with the Debian template. If the VM is already created but

stopped, it would just start it. If you like to change the offering on an existing VM, you must add `force: yes` to the task, which would stop the VM, change the offering and start the VM again.

In the second task we ensure the ports are opened if we give a public IP to the VM.

In the third task we add static NAT to the VMs having a public IP defined.

---

: The public IP addresses must have been acquired in advance, also see `cs_ip_address`

---

---

: For some modules, e.g. `cs_sshkeypair` you usually want this to be executed only once, not for every VM. Therefore you would make a separate play for it targeting `localhost`. You find an example in the use cases below.

---

### Use Case: Provisioning on a Basic Networking CloudStack setup

A basic networking CloudStack setup is slightly different: Every VM gets a public IP directly assigned and security groups are used for access restriction policy.

This is how our inventory looks like:

```
[cloud-vm:children]
webserver

[webserver]
web-01.example.com
web-02.example.com
```

The default for your VMs looks like this:

```
# file: group_vars/cloud-vm
---
cs_offering: Small
cs_securitygroups: [ 'default' ]
```

Our webserver will also be in security group web:

```
# file: group_vars/webserver
---
cs_securitygroups: [ 'default', 'web' ]
```

The playbook looks like the following:

```
# file: infra.yaml
---
- name: cloud base setup
  hosts: localhost
  connection: local
  tasks:
    - name: upload ssh public key
      cs_sshkeypair:
        name: defaultkey
        public_key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"

    - name: ensure security groups exist
      cs_securitygroup:
```

(continues on next page)



()

```

    name: "{{ item }}"
  loop:
    - default
    - web

- name: add inbound SSH to security group default
  cs_securitygroup_rule:
    security_group: default
    start_port: "{{ item }}"
    end_port: "{{ item }}"
  loop:
    - 22

- name: add inbound TCP rules to security group web
  cs_securitygroup_rule:
    security_group: web
    start_port: "{{ item }}"
    end_port: "{{ item }}"
  loop:
    - 80
    - 443

- name: install VMs in the cloud
  hosts: cloud-vm
  connection: local
  tasks:
    - name: create and run VMs on CloudStack
      cs_instance:
        name: "{{ inventory_hostname_short }}"
        template: Linux Debian 7 64-bit 20GB Disk
        service_offering: "{{ cs_offering }}"
        security_groups: "{{ cs_securitygroups }}"
        ssh_key: defaultkey
        state: Running
        register: vm

    - name: show VM IP
      debug: msg="VM {{ inventory_hostname }} {{ vm.default_ip }}"

    - name: assign IP to the inventory
      set_fact: ansible_ssh_host={{ vm.default_ip }}

    - name: waiting for SSH to come up
      wait_for: port=22 host={{ vm.default_ip }} delay=5

```

In the first play we setup the security groups, in the second play the VMs will created be assigned to these groups. Further you see, that we assign the public IP returned from the modules to the host inventory. This is needed as we do not know the IPs we will get in advance. In a next step you would configure the DNS servers with these IPs for accessing the VMs with their DNS name.

In the last task we wait for SSH to be accessible, so any later play would be able to access the VM by SSH without failure.

## 1.11 Getting Started with Docker

Ansible offers the following modules for orchestrating Docker containers:

**docker\_service** Use your existing Docker compose files to orchestrate containers on a single Docker daemon or on Swarm. Supports compose versions 1 and 2.

**docker\_container** Manages the container lifecycle by providing the ability to create, update, stop, start and destroy a container.

**docker\_image** Provides full control over images, including: build, pull, push, tag and remove.

**docker\_image\_facts** Inspects one or more images in the Docker host's image cache, providing the information as facts for making decision or assertions in a playbook.

**docker\_login** Authenticates with Docker Hub or any Docker registry and updates the Docker Engine config file, which in turn provides password-free pushing and pulling of images to and from the registry.

**docker (dynamic inventory)** Dynamically builds an inventory of all the available containers from a set of one or more Docker hosts.

Ansible 2.1.0 includes major updates to the Docker modules, marking the start of a project to create a complete and integrated set of tools for orchestrating containers. In addition to the above modules, we are also working on the following:

Still using Dockerfile to build images? Check out [ansible-container](#), and start building images from your Ansible playbooks.

Use the *shipit* command in [ansible-container](#) to launch your docker-compose file on [OpenShift](#). Go from an app on your laptop to a fully scalable app in the cloud in just a few moments.

There's more planned. See the latest ideas and thinking at the [Ansible proposal repo](#).

### 1.11.1 Requirements

Using the docker modules requires having [docker-py](#) installed on the host running Ansible. You will need to have `>= 1.7.0` installed.

```
$ pip install 'docker-py>=1.7.0'
```

The `docker_service` module also requires `docker-compose`

```
$ pip install 'docker-compose>=1.7.0'
```

### 1.11.2 Connecting to the Docker API

You can connect to a local or remote API using parameters passed to each task or by setting environment variables. The order of precedence is command line parameters and then environment variables. If neither a command line option or an environment variable is found, a default value will be used. The default values are provided under [Parameters](#)

#### Parameters

Control how modules connect to the Docker API by passing the following parameters:

**docker\_host** The URL or Unix socket path used to connect to the Docker API. Defaults to `unix://var/run/docker.sock`. To connect to a remote host, provide the TCP connection string. For example: `tcp://192.0.2.23:2376`. If TLS is used to encrypt the connection to the API, then the module will automatically replace 'tcp' in the connection URL with 'https'.

**api\_version** The version of the Docker API running on the Docker Host. Defaults to the latest version of the API supported by docker-py.

**timeout** The maximum amount of time in seconds to wait on a response from the API. Defaults to 60 seconds.

**tls** Secure the connection to the API by using TLS without verifying the authenticity of the Docker host server. Defaults to False.

**tls\_verify** Secure the connection to the API by using TLS and verifying the authenticity of the Docker host server. Default is False.

**cacert\_path** Use a CA certificate when performing server verification by providing the path to a CA certificate file.

**cert\_path** Path to the client's TLS certificate file.

**key\_path** Path to the client's TLS key file.

**tls\_hostname** When verifying the authenticity of the Docker Host server, provide the expected name of the server. Defaults to 'localhost'.

**ssl\_version** Provide a valid SSL version number. Default value determined by docker-py, which at the time of this writing was 1.0

## Environment Variables

Control how the modules connect to the Docker API by setting the following variables in the environment of the host running Ansible:

**DOCKER\_HOST** The URL or Unix socket path used to connect to the Docker API.

**DOCKER\_API\_VERSION** The version of the Docker API running on the Docker Host. Defaults to the latest version of the API supported by docker-py.

**DOCKER\_TIMEOUT** The maximum amount of time in seconds to wait on a response from the API.

**DOCKER\_CERT\_PATH** Path to the directory containing the client certificate, client key and CA certificate.

**DOCKER\_SSL\_VERSION** Provide a valid SSL version number.

**DOCKER\_TLS** Secure the connection to the API by using TLS without verifying the authenticity of the Docker Host.

**DOCKER\_TLS\_VERIFY** Secure the connection to the API by using TLS and verify the authenticity of the Docker Host.

### 1.11.3 Dynamic Inventory Script

The inventory script generates dynamic inventory by making API requests to one or more Docker APIs. It's dynamic because the inventory is generated at run-time rather than being read from a static file. The script generates the inventory by connecting to one or many Docker APIs and inspecting the containers it finds at each API. Which APIs the script contacts can be defined using environment variables or a configuration file.

## Groups

The script will create the following host groups:

- container id
- container name
- container short id
- image\_name (image\_<image name>)
- docker\_host
- running
- stopped

## Examples

You can run the script interactively from the command line or pass it as the inventory to a playbook. Here are few examples to get you started:

```
# Connect to the Docker API on localhost port 4243 and format the JSON output
DOCKER_HOST=tcp://localhost:4243 ./docker.py --pretty

# Any container's ssh port exposed on 0.0.0.0 will be mapped to
# another IP address (where Ansible will attempt to connect via SSH)
DOCKER_DEFAULT_IP=192.0.2.5 ./docker.py --pretty

# Run as input to a playbook:
ansible-playbook -i ~/projects/ansible/contrib/inventory/docker.py docker_inventory_
↳test.yml

# Simple playbook to invoke with the above example:

- name: Test docker_inventory
  hosts: all
  connection: local
  gather_facts: no
  tasks:
    - debug: msg="Container - {{ inventory_hostname }}"
```

## Configuration

You can control the behavior of the inventory script by defining environment variables, or creating a docker.yml file (sample provided in ansible/contrib/inventory). The order of precedence is the docker.yml file and then environment variables.

## Environment Variables

To connect to a single Docker API the following variables can be defined in the environment to control the connection options. These are the same environment variables used by the Docker modules.

**DOCKER\_HOST** The URL or Unix socket path used to connect to the Docker API. Defaults to  
unix:///var/run/docker.sock.

**DOCKER\_API\_VERSION:** The version of the Docker API running on the Docker Host. Defaults to the latest version of the API supported by docker-py.

**DOCKER\_TIMEOUT:** The maximum amount of time in seconds to wait on a response from the API. Defaults to 60 seconds.

**DOCKER\_TLS:** Secure the connection to the API by using TLS without verifying the authenticity of the Docker host server. Defaults to False.

**DOCKER\_TLS\_VERIFY:** Secure the connection to the API by using TLS and verifying the authenticity of the Docker host server. Default is False

**DOCKER\_TLS\_HOSTNAME:** When verifying the authenticity of the Docker Host server, provide the expected name of the server. Defaults to localhost.

**DOCKER\_CERT\_PATH:** Path to the directory containing the client certificate, client key and CA certificate.

**DOCKER\_SSL\_VERSION:** Provide a valid SSL version number. Default value determined by docker-py, which at the time of this writing was 1.0

In addition to the connection variables there are a couple variables used to control the execution and output of the script:

**DOCKER\_CONFIG\_FILE** Path to the configuration file. Defaults to ./docker.yml.

**DOCKER\_PRIVATE\_SSH\_PORT:** The private port (container port) on which SSH is listening for connections. Defaults to 22.

**DOCKER\_DEFAULT\_IP:** The IP address to assign to ansible\_host when the container's SSH port is mapped to interface '0.0.0.0'.

## Configuration File

Using a configuration file provides a means for defining a set of Docker APIs from which to build an inventory.

The default name of the file is derived from the name of the inventory script. By default the script will look for basename of the script (i.e. docker) with an extension of '.yml'.

You can also override the default name of the script by defining DOCKER\_CONFIG\_FILE in the environment.

Here's what you can define in docker\_inventory.yml:

**defaults** Defines a default connection. Defaults will be taken from this and applied to any values not provided for a host defined in the hosts list.

**hosts** If you wish to get inventory from more than one Docker host, define a hosts list.

For the default host and each host in the hosts list define the following attributes:

```
host:
  description: The URL or Unix socket path used to connect to the Docker API.
  required: yes

tls:
  description: Connect using TLS without verifying the authenticity of the Docker_
↪host server.
  default: false
  required: false

tls_verify:
```

(continues on next page)

```

    description: Connect using TLS without verifying the authenticity of the Docker_
↪host server.
    default: false
    required: false

cert_path:
    description: Path to the client's TLS certificate file.
    default: null
    required: false

cacert_path:
    description: Use a CA certificate when performing server verification by providing_
↪the path to a CA certificate file.
    default: null
    required: false

key_path:
    description: Path to the client's TLS key file.
    default: null
    required: false

version:
    description: The Docker API version.
    required: false
    default: will be supplied by the docker-py module.

timeout:
    description: The amount of time in seconds to wait on an API response.
    required: false
    default: 60

default_ip:
    description: The IP address to assign to ansible_host when the container's SSH_
↪port is mapped to interface
    '0.0.0.0'.
    required: false
    default: 127.0.0.1

private_ssh_port:
    description: The port containers use for SSH
    required: false
    default: 22

```

## 1.12 Google Cloud Platform Guide

### 1.12.1 Introduction

Ansible + Google have been working together on a set of auto-generated Ansible modules designed to consistently and comprehensively cover the entirety of the Google Cloud Platform.

Ansible contains modules for managing Google Cloud Platform resources, including creating instances, controlling network access, working with persistent disks, managing load balancers, and a lot more.

These new modules can be found under a new consistent name scheme "gcp\_\*" (Note: gcp\_target\_proxy and gcp\_url\_map are legacy modules, despite the "gcp\_\*" name. Please use gcp\_compute\_target\_proxy and

`gcp_compute_url_map` instead).

Additionally, the `gcp_compute` inventory plugin can discover all GCE instances and make them automatically available in your Ansible inventory.

You may see a collection of other GCP modules that do not conform to this naming convention. These are the original modules primarily developed by the Ansible community. You will find some overlapping functionality such as with the the "gce" module and the new "gcp\_compute\_instance" module. Either can be used, but you may experience issues trying to use them together.

While the community GCP modules are not going away, Google is investing effort into the new "gcp\_\*" modules. Google is committed to ensuring the Ansible community has a great experience with GCP and therefore recommends that begin adopting these new modules if possible.

### 1.12.2 Introduction

The Google Cloud Platform (GCP) modules require both the `requests` and the `google-auth` libraries to be installed.

```
$ pip install requests google-auth
```

### 1.12.3 Credentials

It's easy to create a GCP account with credentials for Ansible. You have multiple options to get your credentials - here are two of the most common options:

- **Service Accounts (Recommended):** Use JSON service accounts with specific permissions.
- **Machine Accounts:** Use the permissions associated with the GCP Instance you're using Ansible on.

For the following examples, we'll be using service account credentials.

To work with the GCP modules, you'll first need to get some credentials in the JSON format:

1. [Create a Service Account](#)
2. [Download JSON credentials](#)

Once you have your credentials, there are two different ways to provide them to Ansible:

- by specifying them directly as module parameters
- by setting environment variables

#### Providing Credentials as Module Parameters

For the GCE modules you can specify the credentials as arguments:

- `auth_kind`: type of authentication being used (choices: `machineaccount`, `serviceaccount`, `application`)
- `service_account_email`: email associated with the project
- `service_account_file`: path to the JSON credentials file
- `project`: id of the project
- `scopes`: The specific scopes that you want the actions to use.

For example, to create a new IP address using the `gcp_compute_address` module, you can use the following configuration:

```
- name: Create IP address
  hosts: localhost
  connection: local
  gather_facts: no

vars:
  service_account_file: /home/my_account.json
  project: my-project
  auth_kind: serviceaccount
  scopes:
    - www.googleapis.com/auth/compute

tasks:

- name: Allocate an IP Address
  gcp_compute_address:
    state: present
    name: 'test-address1'
    region: 'us-west1'
    project: "{{ project }}"
    auth_kind: "{{ auth_kind }}"
    service_account_file: "{{ service_account_file }}"
    scopes: "{{ scopes }}"
```

## Providing Credentials as Environment Variables

Set the following environment variables before running Ansible in order to configure your credentials:

```
GCP_AUTH_KIND
GCP_SERVICE_ACCOUNT_EMAIL
GCP_SERVICE_ACCOUNT_FILE
GCP_SCOPES
```

### 1.12.4 GCE Dynamic Inventory

The best way to interact with your hosts is to use the `gcp_compute` inventory plugin, which dynamically queries GCE and tells Ansible what nodes can be managed.

To use the `gcp_compute` inventory plugin, create a file that ends in `.gcp.yml` file in your root directory. The `gcp_compute` inventory script takes in the same authentication information as any module.

Here's an example of a valid inventory file:

```
plugin: gcp_compute
projects:
  - google.com:graphite-playground
filters:
auth_kind: serviceaccount
service_account_file: /home/alexstephen/my_account.json
```

Executing `ansible-inventory --list -i <filename>.gcp.yml` will create a list of GCP instances that are ready to be configured using Ansible.



## Create an instance

The full range of GCP modules provide the ability to create a wide variety of GCP resources with the full support of the entire GCP API.

The following playbook creates a GCE Instance. This instance relies on a GCP network and a Disk. By creating the Disk and Network separately, we can give as much detail as necessary about how we want the disk and network formatted. By registering a Disk/Network to a variable, we can simply insert the variable into the instance task. The `gcp_compute_instance` module will figure out the rest.

```
- name: Create an instance
  hosts: localhost
  gather_facts: no
  connection: local
  vars:
    project: my-project
    auth_kind: serviceaccount
    service_account_file: /home/my_account.json
    zone: "us-centrall-a"
    region: "us-centrall"

  tasks:
    - name: create a disk
      gcp_compute_disk:
        name: 'disk-instance'
        size_gb: 50
        source_image: 'projects/ubuntu-os-cloud/global/images/family/ubuntu-1604-lts'
        zone: "{{ zone }}"
        project: "{{ gcp_project }}"
        auth_kind: "{{ gcp_cred_kind }}"
        service_account_file: "{{ gcp_cred_file }}"
        scopes:
          - https://www.googleapis.com/auth/compute
        state: present
      register: disk

    - name: create a network
      gcp_compute_network:
        name: 'network-instance'
        project: "{{ gcp_project }}"
        auth_kind: "{{ gcp_cred_kind }}"
        service_account_file: "{{ gcp_cred_file }}"
        scopes:
          - https://www.googleapis.com/auth/compute
        state: present
      register: network

    - name: create a address
      gcp_compute_address:
        name: 'address-instance'
        region: "{{ region }}"
        project: "{{ gcp_project }}"
        auth_kind: "{{ gcp_cred_kind }}"
        service_account_file: "{{ gcp_cred_file }}"
        scopes:
          - https://www.googleapis.com/auth/compute
        state: present
      register: address

    - name: create a instance
      gcp_compute_instance:
```

(continues on next page)

```

state: present
name: test-vm
machine_type: n1-standard-1
disks:
  - auto_delete: true
    boot: true
    source: "{{ disk }}"
network_interfaces:
  - network: "{{ network }}"
    access_configs:
      - name: 'External NAT'
        nat_ip: "{{ address }}"
        type: 'ONE_TO_ONE_NAT'
zone: "{{ zone }}"
project: "{{ gcp_project }}"
auth_kind: "{{ gcp_cred_kind }}"
service_account_file: "{{ gcp_cred_file }}"
scopes:
  - https://www.googleapis.com/auth/compute
register: instance

- name: Wait for SSH to come up
  wait_for: host={{ instance.address }} port=22 delay=10 timeout=60

- name: Add host to groupname
  add_host: hostname={{ instance.address }} groupname=new_instances

- name: Manage new instances
  hosts: new_instances
  connection: ssh
  sudo: True
  roles:
    - base_configuration
    - production_server

```

Note that use of the "add\_host" module above creates a temporary, in-memory group. This means that a play in the same playbook can then manage machines in the 'new\_instances' group, if so desired. Any sort of arbitrary configuration is possible at this point.

For more information about Google Cloud, please visit the *Google Cloud website* <<https://cloud.google.com>>

## 1.13 Using Ansible with the Packet host

### 1.13.1 Introduction

[Packet.net](https://www.packet.net) is a bare metal infrastructure host that's supported by Ansible (>=2.3) via a dynamic inventory script and two cloud modules. The two modules are:

- `packet_sshkey`: adds a public SSH key from file or value to the Packet infrastructure. Every subsequently-created device will have this public key installed in `.ssh/authorized_keys`.
- `packet_device`: manages servers on Packet. You can use this module to create, restart and delete devices.

Note, this guide assumes you are familiar with Ansible and how it works. If you're not, have a look at their *docs* before getting started.

### 1.13.2 Requirements

The Packet modules and inventory script connect to the Packet API using the packet-python package. You can install it with pip:

```
$ pip install packet-python
```

In order to check the state of devices created by Ansible on Packet, it's a good idea to install one of the [Packet CLI clients](#). Otherwise you can check them via the [Packet portal](#).

To use the modules and inventory script you'll need a Packet API token. You can generate an API token via the Packet portal [here](#). The simplest way to authenticate yourself is to set the Packet API token in an environment variable:

```
$ export PACKET_API_TOKEN=Bfse9F24SFtfs423Gsd3ifGsd43sSdfs
```

If you're not comfortable exporting your API token, you can pass it as a parameter to the modules.

On Packet, devices and reserved IP addresses belong to [projects](#). In order to use the packet\_device module, you need to specify the UUID of the project in which you want to create or manage devices. You can find a project's UUID in the Packet portal [here](#) (it's just under the project table) or via one of the available [CLIs](#).

If you want to use a new SSH keypair in this tutorial, you can generate it to `./id_rsa` and `./id_rsa.pub` as:

```
$ ssh-keygen -t rsa -f ./id_rsa
```

If you want to use an existing keypair, just copy the private and public key over to the playbook directory.

### 1.13.3 Device Creation

The following code block is a simple playbook that creates one [Type 0](#) server (the 'plan' parameter). You have to supply 'plan' and 'operating\_system'. 'location' defaults to 'ewr1' (Parsippany, NJ). You can find all the possible values for the parameters via a [CLI client](#).

```
# playbook_create.yml

- name: create ubuntu device
  hosts: localhost
  tasks:

  - packet_sshkey:
    key_file: ./id_rsa.pub
    label: tutorial key

  - packet_device:
    project_id: <your_project_id>
    hostnames: myserver
    operating_system: ubuntu_16_04
    plan: baremetal_0
    facility: sjc1
```

After running `ansible-playbook playbook_create.yml`, you should have a server provisioned on Packet. You can verify via a CLI or in the [Packet portal](#).

If you get an error with the message "failed to set machine state present, error: Error 404: Not Found", please verify your project UUID.

### 1.13.4 Updating Devices

The two parameters used to uniquely identify Packet devices are: "device\_ids" and "hostnames". Both parameters accept either a single string (later converted to a one-element list), or a list of strings.

The 'device\_ids' and 'hostnames' parameters are mutually exclusive. The following values are all acceptable:

- device\_ids: a27b7a83-fc93-435b-a128-47a5b04f2dcf
- hostnames: mydev1
- device\_ids: [a27b7a83-fc93-435b-a128-47a5b04f2dcf, 4887130f-0ccd-49a0-99b0-323c1ceb527b]
- hostnames: [mydev1, mydev2]

In addition, hostnames can contain a special '%d' formatter along with a 'count' parameter that lets you easily expand hostnames that follow a simple name and number pattern; i.e. hostnames: "mydev%d", count: 2 will expand to [mydev1, mydev2].

If your playbook acts on existing Packet devices, you can only pass the 'hostname' and 'device\_ids' parameters. The following playbook shows how you can reboot a specific Packet device by setting the 'hostname' parameter:

```
# playbook_reboot.yml

- name: reboot myserver
  hosts: localhost
  tasks:

  - packet_device:
      project_id: <your_project_id>
      hostnames: myserver
      state: rebooted
```

You can also identify specific Packet devices with the 'device\_ids' parameter. The device's UUID can be found in the [Packet Portal](#) or by using a [CLI](#). The following playbook removes a Packet device using the 'device\_ids' field:

```
# playbook_remove.yml

- name: remove a device
  hosts: localhost
  tasks:

  - packet_device:
      project_id: <your_project_id>
      device_ids: <myserver_device_id>
      state: absent
```

### 1.13.5 More Complex Playbooks

In this example, we'll create a CoreOS cluster with [user data](#).

The CoreOS cluster will use [etcd](#) for discovery of other servers in the cluster. Before provisioning your servers, you'll need to generate a discovery token for your cluster:

```
$ curl -w "\n" 'https://discovery.etcd.io/new?size=3'
```

The following playbook will create an SSH key, 3 Packet servers, and then wait until SSH is ready (or until 5 minutes passed). Make sure to substitute the discovery token URL in 'user\_data', and the 'project\_id' before running `ansible-playbook`. Also, feel free to change 'plan' and 'facility'.

```
# playbook_coreos.yml

- name: Start 3 CoreOS nodes in Packet and wait until SSH is ready
  hosts: localhost
  tasks:

  - packet_sshkey:
    key_file: ./id_rsa.pub
    label: new

  - packet_device:
    hostnames: [coreos-one, coreos-two, coreos-three]
    operating_system: coreos_beta
    plan: baremetal_0
    facility: ewr1
    project_id: <your_project_id>
    wait_for_public_IPv: 4
    user_data: |
      #cloud-config
      coreos:
        etcd2:
          discovery: https://discovery.etcd.io/<token>
          advertise-client-urls: http://$private_ipv4:2379,http://$private_ipv4:4001
          initial-advertise-peer-urls: http://$private_ipv4:2380
          listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
          listen-peer-urls: http://$private_ipv4:2380
        fleet:
          public-ip: $private_ipv4
        units:
          - name: etcd2.service
            command: start
          - name: fleet.service
            command: start
    register: newhosts

  - name: wait for ssh
    wait_for:
      delay: 1
      host: "{{ item.public_ipv4 }}"
      port: 22
      state: started
      timeout: 500
    loop: "{{ newhosts.devices }}"
```

As with most Ansible modules, the default states of the Packet modules are idempotent, meaning the resources in your project will remain the same after re-runs of a playbook. Thus, we can keep the `packet_sshkey` module call in our playbook. If the public key is already in your Packet account, the call will have no effect.

The second module call provisions 3 Packet Type 0 (specified using the 'plan' parameter) servers in the project identified via the 'project\_id' parameter. The servers are all provisioned with CoresOS beta (the 'operating\_system' parameter) and are customized with cloud-config user data passed to the 'user\_data' parameter.

The `packet_device` module has a `wait_for_public_IPv` that is used to specify the version of the IP address to wait for (valid values are 4 or 6 for IPv4 or IPv6). If specified, Ansible will wait until the GET API call for a device contains an Internet-routeable IP address of the specified version. When referring to an IP address of a created device in subsequent module calls, it's wise to use the `wait_for_public_IPv` parameter, or `state: active` in the `packet_device` module call.

Run the playbook:

```
$ ansible-playbook playbook_coreos.yml
```

Once the playbook quits, your new devices should be reachable via SSH. Try to connect to one and check if etcd has started properly:

```
tomk@work $ ssh -i id_rsa core@$one_of_the_servers_ip
core@coreos-one ~ $ etcdctl cluster-health
```

Once you create a couple of devices, you might appreciate the dynamic inventory script...

### 1.13.6 Dynamic Inventory Script

The dynamic inventory script queries the Packet API for a list of hosts, and exposes it to Ansible so you can easily identify and act on Packet devices. You can find it in Ansible's git repo at [contrib/inventory/packet\\_net.py](#).

The inventory script is configurable via a [ini file](#).

If you want to use the inventory script, you must first export your Packet API token to a `PACKET_API_TOKEN` environment variable.

You can either copy the inventory and ini config out from the cloned git repo, or you can download it to your working directory like so:

```
$ wget https://github.com/ansible/ansible/raw/devel/contrib/inventory/packet_net.py
$ chmod +x packet_net.py
$ wget https://github.com/ansible/ansible/raw/devel/contrib/inventory/packet_net.ini
```

In order to understand what the inventory script gives to Ansible you can run:

```
$ ./packet_net.py --list
```

It should print a JSON document looking similar to following trimmed dictionary:

```
{
  "_meta": {
    "hostvars": {
      "147.75.64.169": {
        "packet_billing_cycle": "hourly",
        "packet_created_at": "2017-02-09T17:11:26Z",
        "packet_facility": "ewrl",
        "packet_hostname": "coreos-two",
        "packet_href": "/devices/d0ab8972-54a8-4bff-832b-28549dlbec96",
        "packet_id": "d0ab8972-54a8-4bff-832b-28549dlbec96",
        "packet_locked": false,
        "packet_operating_system": "coreos_beta",
        "packet_plan": "baremetal_0",
        "packet_state": "active",
        "packet_updated_at": "2017-02-09T17:16:35Z",
        "packet_user": "core",
        "packet_userdata": "#cloud-config\ncoreos:\n  etcd2:\n    discovery: https://
↪discovery.etcd.io/e0c8a4a9b8fe61acd51ec599e2a4f68e\n    advertise-client-urls:
↪http://$private_ipv4:2379,http://$private_ipv4:4001\n    initial-advertise-peer-
↪urls: http://$private_ipv4:2380\n    listen-client-urls: http://0.0.0.0:2379,http://
↪0.0.0.0:4001\n    listen-peer-urls: http://$private_ipv4:2380\n  fleet:\n    public-
↪ip: $private_ipv4\n    units:\n      - name: etcd2.service\n        command: start\n
↪name: fleet.service\n        command: start"
```

(continues on next page)

()

```

    }
  }
},
"baremetal_0": [
  "147.75.202.255",
  "147.75.202.251",
  "147.75.202.249",
  "147.75.64.129",
  "147.75.192.51",
  "147.75.64.169"
],
"coreos_beta": [
  "147.75.202.255",
  "147.75.202.251",
  "147.75.202.249",
  "147.75.64.129",
  "147.75.192.51",
  "147.75.64.169"
],
"ewr1": [
  "147.75.64.129",
  "147.75.192.51",
  "147.75.64.169"
],
"sjc1": [
  "147.75.202.255",
  "147.75.202.251",
  "147.75.202.249"
],
"coreos-two": [
  "147.75.64.169"
],
"d0ab8972-54a8-4bff-832b-28549d1bec96": [
  "147.75.64.169"
]
}

```

In the [ '\_meta' ] [ 'hostvars' ] key, there is a list of devices (uniquely identified by their public IPv4 address) with their parameters. The other keys under [ '\_meta' ] are lists of devices grouped by some parameter. Here, it is type (all devices are of type baremetal\_0), operating system, and facility (ewr1 and sjc1).

In addition to the parameter groups, there are also one-item groups with the UUID or hostname of the device.

You can now target groups in playbooks! The following playbook will install a role that supplies resources for an Ansible target into all devices in the "coreos\_beta" group:

```

# playbook_bootstrap.yml

- hosts: coreos_beta
  gather_facts: false
  roles:
    - defunctzombie.coreos-bootstrap

```

Don't forget to supply the dynamic inventory in the `-i` argument!

```
$ ansible-playbook -u core -i packet_net.py playbook_bootstrap.yml
```

If you have any questions or comments let us know! [help@packet.net](mailto:help@packet.net)

## 1.14 Rackspace Cloud Guide

### 1.14.1 Introduction

---

: This section of the documentation is under construction. We are in the process of adding more examples about the Rackspace modules and how they work together. Once complete, there will also be examples for Rackspace Cloud in [ansible-examples](#).

---

Ansible contains a number of core modules for interacting with Rackspace Cloud.

The purpose of this section is to explain how to put Ansible modules together (and use inventory scripts) to use Ansible in a Rackspace Cloud context.

Prerequisites for using the *rax* modules are minimal. In addition to ansible itself, all of the modules require and are tested against *pyrax* 1.5 or higher. You'll need this Python module installed on the execution host.

*pyrax* is not currently available in many operating system package repositories, so you will likely need to install it via *pip*:

```
$ pip install pyrax
```

The following steps will often execute from the control machine against the Rackspace Cloud API, so it makes sense to add *localhost* to the inventory file. (Ansible may not require this manual step in the future):

```
[localhost]
localhost ansible_connection=local
```

In playbook steps, we'll typically be using the following pattern:

```
- hosts: localhost
  connection: local
  gather_facts: False
  tasks:
```

### 1.14.2 Credentials File

The *rax.py* inventory script and all *rax* modules support a standard *pyrax* credentials file that looks like:

```
[rackspace_cloud]
username = myraxusername
api_key = d41d8cd98f00b204e9800998ecf8427e
```

Setting the environment parameter *RAX\_CREDS\_FILE* to the path of this file will help Ansible find how to load this information.

More information about this credentials file can be found at [https://github.com/rackspace/pyrax/blob/master/docs/getting\\_started.md#authenticating](https://github.com/rackspace/pyrax/blob/master/docs/getting_started.md#authenticating)

### Running from a Python Virtual Environment (Optional)

Most users will not be using *virtualenv*, but some users, particularly Python developers sometimes like to.

There are special considerations when Ansible is installed to a Python *virtualenv*, rather than the default of installing at a global scope. Ansible assumes, unless otherwise instructed, that the python binary will live at */usr/bin/python*.



This is done via the interpreter line in modules, however when instructed by setting the inventory variable 'ansible\_python\_interpreter', Ansible will use this specified path instead to find Python. This can be a cause of confusion as one may assume that modules running on 'localhost', or perhaps running via 'local\_action', are using the virtualenv Python interpreter. By setting this line in the inventory, the modules will execute in the virtualenv interpreter and have available the virtualenv packages, specifically pyrax. If using virtualenv, you may wish to modify your localhost inventory definition to find this location as follows:

```
[localhost]
localhost ansible_connection=local ansible_python_interpreter=/path/to/ansible_venv/
↳bin/python
```

: pyrax may be installed in the global Python package scope or in a virtual environment. There are no special considerations to keep in mind when installing pyrax.

### 1.14.3 Provisioning

Now for the fun parts.

The 'rax' module provides the ability to provision instances within Rackspace Cloud. Typically the provisioning task will be performed from your Ansible control server (in our example, localhost) against the Rackspace cloud API. This is done for several reasons:

- Avoiding installing the pyrax library on remote nodes
- No need to encrypt and distribute credentials to remote nodes
- Speed and simplicity

: Authentication with the Rackspace-related modules is handled by either specifying your username and API key as environment variables or passing them as module arguments, or by specifying the location of a credentials file.

Here is a basic example of provisioning an instance in ad-hoc mode:

```
$ ansible localhost -m rax -a "name=awx flavor=4 image=ubuntu-1204-lts-precise-
↳pangolin wait=yes" -c local
```

Here's what it would look like in a playbook, assuming the parameters were defined in variables:

```
tasks:
  - name: Provision a set of instances
    local_action:
      module: rax
      name: "{{ rax_name }}"
      flavor: "{{ rax_flavor }}"
      image: "{{ rax_image }}"
      count: "{{ rax_count }}"
      group: "{{ group }}"
      wait: yes
      register: rax
```

The rax module returns data about the nodes it creates, like IP addresses, hostnames, and login passwords. By registering the return value of the step, it is possible used this data to dynamically add the resulting hosts to inventory (temporarily, in memory). This facilitates performing configuration actions on the hosts in a follow-on task. In the

following example, the servers that were successfully created using the above task are dynamically added to a group called "raxhosts", with each nodes hostname, IP address, and root password being added to the inventory.

```
- name: Add the instances we created (by public IP) to the group 'raxhosts'
  local_action:
    module: add_host
    hostname: "{{ item.name }}"
    ansible_host: "{{ item.rax_accessipv4 }}"
    ansible_ssh_pass: "{{ item.rax_adminpass }}"
    groups: raxhosts
  loop: "{{ rax.success }}"
  when: rax.action == 'create'
```

With the host group now created, the next play in this playbook could now configure servers belonging to the raxhosts group.

```
- name: Configuration play
  hosts: raxhosts
  user: root
  roles:
    - ntp
    - webserver
```

The method above ties the configuration of a host with the provisioning step. This isn't always what you want, and leads us to the next section.

### 1.14.4 Host Inventory

Once your nodes are spun up, you'll probably want to talk to them again. The best way to handle this is to use the "rax" inventory plugin, which dynamically queries Rackspace Cloud and tells Ansible what nodes you have to manage. You might want to use this even if you are spinning up cloud instances via other tools, including the Rackspace Cloud user interface. The inventory plugin can be used to group resources by metadata, region, OS, etc. Utilizing metadata is highly recommended in "rax" and can provide an easy way to sort between host groups and roles. If you don't want to use the `rax.py` dynamic inventory script, you could also still choose to manually manage your INI inventory file, though this is less recommended.

In Ansible it is quite possible to use multiple dynamic inventory plugins along with INI file data. Just put them in a common directory and be sure the scripts are `chmod +x`, and the INI-based ones are not.

#### **`rax.py`**

To use the rackspace dynamic inventory script, copy `rax.py` into your inventory directory and make it executable. You can specify a credentials file for `rax.py` utilizing the `RAX_CREDS_FILE` environment variable.

---

: Dynamic inventory scripts (like `rax.py`) are saved in `/usr/share/ansible/inventory` if Ansible has been installed globally. If installed to a virtualenv, the inventory scripts are installed to `$VIRTUALENV/share/inventory`.

---

: Users of *Ansible Tower* will note that dynamic inventory is natively supported by Tower, and all you have to do is associate a group with your Rackspace Cloud credentials, and it will easily synchronize without going through these steps:

```
$ RAX_CREDS_FILE=~/.raxpub ansible all -i rax.py -m setup
```

`rax.py` also accepts a `RAX_REGION` environment variable, which can contain an individual region, or a comma separated list of regions.

When using `rax.py`, you will not have a `'localhost'` defined in the inventory.

As mentioned previously, you will often be running most of these modules outside of the host loop, and will need `'localhost'` defined. The recommended way to do this, would be to create an `inventory` directory, and place both the `rax.py` script and a file containing `localhost` in it.

Executing `ansible` or `ansible-playbook` and specifying the `inventory` directory instead of an individual file, will cause `ansible` to evaluate each file in that directory for inventory.

Let's test our inventory script to see if it can talk to Rackspace Cloud.

```
$ RAX_CREDS_FILE=~/.raxpub ansible all -i inventory/ -m setup
```

Assuming things are properly configured, the `rax.py` inventory script will output information similar to the following information, which will be utilized for inventory and variables.

```
{
  "ORD": [
    "test"
  ],
  "_meta": {
    "hostvars": {
      "test": {
        "ansible_host": "198.51.100.1",
        "rax_accessip4": "198.51.100.1",
        "rax_accessip6": "2001:DB8::2342",
        "rax_addresses": {
          "private": [
            {
              "addr": "192.0.2.2",
              "version": 4
            }
          ],
          "public": [
            {
              "addr": "198.51.100.1",
              "version": 4
            },
            {
              "addr": "2001:DB8::2342",
              "version": 6
            }
          ]
        },
        "rax_config_drive": "",
        "rax_created": "2013-11-14T20:48:22Z",
        "rax_flavor": {
          "id": "performance1-1",
          "links": [
            {
              "href": "https://ord.servers.api.rackspacecloud.com/
↪111111/flavors/performance1-1",
```

(continues on next page)

```

        "rel": "bookmark"
    }
}
},
"rax_hostid":
↪ "e7b6961a9bd943ee82b13816426f1563bfda6846aad84d52af45a4904660cde0",
"rax_human_id": "test",
"rax_id": "099a447b-a644-471f-87b9-a7f580eb0c2a",
"rax_image": {
    "id": "b211c7bf-b5b4-4ede-a8de-a4368750c653",
    "links": [
        {
            "href": "https://ord.servers.api.rackspacecloud.com/
↪ 111111/images/b211c7bf-b5b4-4ede-a8de-a4368750c653",
            "rel": "bookmark"
        }
    ]
},
"rax_key_name": null,
"rax_links": [
    {
        "href": "https://ord.servers.api.rackspacecloud.com/v2/111111/
↪ servers/099a447b-a644-471f-87b9-a7f580eb0c2a",
        "rel": "self"
    },
    {
        "href": "https://ord.servers.api.rackspacecloud.com/111111/
↪ servers/099a447b-a644-471f-87b9-a7f580eb0c2a",
        "rel": "bookmark"
    }
],
"rax_metadata": {
    "foo": "bar"
},
"rax_name": "test",
"rax_name_attr": "name",
"rax_networks": {
    "private": [
        "192.0.2.2"
    ],
    "public": [
        "198.51.100.1",
        "2001:DB8::2342"
    ]
},
"rax_os-dcf_diskconfig": "AUTO",
"rax_os-ext-sts_power_state": 1,
"rax_os-ext-sts_task_state": null,
"rax_os-ext-sts_vm_state": "active",
"rax_progress": 100,
"rax_status": "ACTIVE",
"rax_tenant_id": "111111",
"rax_updated": "2013-11-14T20:49:27Z",
"rax_user_id": "22222"
}
}
}

```

(continues on next page)

()

}

## Standard Inventory

When utilizing a standard ini formatted inventory file (as opposed to the inventory plugin), it may still be advantageous to retrieve discoverable hostvar information from the Rackspace API.

This can be achieved with the `rax_facts` module and an inventory file similar to the following:

```
[test_servers]
hostname1 rax_region=ORD
hostname2 rax_region=ORD
```

```
- name: Gather info about servers
  hosts: test_servers
  gather_facts: False
  tasks:
    - name: Get facts about servers
      local_action:
        module: rax_facts
        credentials: ~/.raxpub
        name: "{{ inventory_hostname }}"
        region: "{{ rax_region }}"
    - name: Map some facts
      set_fact:
        ansible_host: "{{ rax_accessip4 }}"
```

While you don't need to know how it works, it may be interesting to know what kind of variables are returned.

The `rax_facts` module provides facts as followings, which match the `rax.py` inventory script:

```
{
  "ansible_facts": {
    "rax_accessip4": "198.51.100.1",
    "rax_accessip6": "2001:DB8::2342",
    "rax_addresses": {
      "private": [
        {
          "addr": "192.0.2.2",
          "version": 4
        }
      ],
      "public": [
        {
          "addr": "198.51.100.1",
          "version": 4
        },
        {
          "addr": "2001:DB8::2342",
          "version": 6
        }
      ]
    },
    "rax_config_drive": "",
    "rax_created": "2013-11-14T20:48:22Z",
```

(continues on next page)

```

    "rax_flavor": {
      "id": "performancel-1",
      "links": [
        {
          "href": "https://ord.servers.api.rackspacecloud.com/111111/
↪ flavors/performancel-1",
          "rel": "bookmark"
        }
      ]
    },
    "rax_hostid":
↪ "e7b6961a9bd943ee82b13816426f1563bfda6846aad84d52af45a4904660cde0",
    "rax_human_id": "test",
    "rax_id": "099a447b-a644-471f-87b9-a7f580eb0c2a",
    "rax_image": {
      "id": "b211c7bf-b5b4-4ede-a8de-a4368750c653",
      "links": [
        {
          "href": "https://ord.servers.api.rackspacecloud.com/111111/images/
↪ b211c7bf-b5b4-4ede-a8de-a4368750c653",
          "rel": "bookmark"
        }
      ]
    },
    "rax_key_name": null,
    "rax_links": [
      {
        "href": "https://ord.servers.api.rackspacecloud.com/v2/111111/servers/
↪ 099a447b-a644-471f-87b9-a7f580eb0c2a",
        "rel": "self"
      },
      {
        "href": "https://ord.servers.api.rackspacecloud.com/111111/servers/
↪ 099a447b-a644-471f-87b9-a7f580eb0c2a",
        "rel": "bookmark"
      }
    ],
    "rax_metadata": {
      "foo": "bar"
    },
    "rax_name": "test",
    "rax_name_attr": "name",
    "rax_networks": {
      "private": [
        "192.0.2.2"
      ],
      "public": [
        "198.51.100.1",
        "2001:DB8::2342"
      ]
    },
    "rax_os-dcf_diskconfig": "AUTO",
    "rax_os-ext-sts_power_state": 1,
    "rax_os-ext-sts_task_state": null,
    "rax_os-ext-sts_vm_state": "active",
    "rax_progress": 100,
    "rax_status": "ACTIVE",

```

(continues on next page)

()

```

    "rax_tenant_id": "111111",
    "rax_updated": "2013-11-14T20:49:27Z",
    "rax_user_id": "22222"
  },
  "changed": false
}
```

### 1.14.5 Use Cases

This section covers some additional usage examples built around a specific use case.

#### Network and Server

Create an isolated cloud network and build a server

```

- name: Build Servers on an Isolated Network
  hosts: localhost
  connection: local
  gather_facts: False
  tasks:
    - name: Network create request
      local_action:
        module: rax_network
        credentials: ~/.raxpub
        label: my-net
        cidr: 192.168.3.0/24
        region: IAD
        state: present

    - name: Server create request
      local_action:
        module: rax
        credentials: ~/.raxpub
        name: web%04d.example.org
        flavor: 2
        image: ubuntu-1204-lts-precise-pangolin
        disk_config: manual
        networks:
          - public
          - my-net
        region: IAD
        state: present
        count: 5
        exact_count: yes
        group: web
        wait: yes
        wait_timeout: 360
        register: rax
```

#### Complete Environment

Build a complete webserver environment with servers, custom networks and load balancers, install nginx and create a custom index.html

```
---
- name: Build environment
  hosts: localhost
  connection: local
  gather_facts: False
  tasks:
    - name: Load Balancer create request
      local_action:
        module: rax_clb
        credentials: ~/.raxpub
        name: my-lb
        port: 80
        protocol: HTTP
        algorithm: ROUND_ROBIN
        type: PUBLIC
        timeout: 30
        region: IAD
        wait: yes
        state: present
        meta:
          app: my-cool-app
        register: clb

    - name: Network create request
      local_action:
        module: rax_network
        credentials: ~/.raxpub
        label: my-net
        cidr: 192.168.3.0/24
        state: present
        region: IAD
        register: network

    - name: Server create request
      local_action:
        module: rax
        credentials: ~/.raxpub
        name: web%04d.example.org
        flavor: performancel-1
        image: ubuntu-1204-lts-precise-pangolin
        disk_config: manual
        networks:
          - public
          - private
          - my-net
        region: IAD
        state: present
        count: 5
        exact_count: yes
        group: web
        wait: yes
        register: rax

    - name: Add servers to web host group
      local_action:
        module: add_host
        hostname: "{{ item.name }}"
```

(continues on next page)



()

```

    ansible_host: "{{ item.rax_accessipv4 }}"
    ansible_ssh_pass: "{{ item.rax_adminpass }}"
    ansible_user: root
    groups: web
    loop: "{{ rax.success }}"
    when: rax.action == 'create'

- name: Add servers to Load balancer
  local_action:
    module: rax_clb_nodes
    credentials: ~/.raxpub
    load_balancer_id: "{{ clb.balancer.id }}"
    address: "{{ item.rax_networks.private|first }}"
    port: 80
    condition: enabled
    type: primary
    wait: yes
    region: IAD
    loop: "{{ rax.success }}"
    when: rax.action == 'create'

- name: Configure servers
  hosts: web
  handlers:
    - name: restart nginx
      service: name=nginx state=restarted

  tasks:
    - name: Install nginx
      apt: pkg=nginx state=latest update_cache=yes cache_valid_time=86400
      notify:
        - restart nginx

    - name: Ensure nginx starts on boot
      service: name=nginx state=started enabled=yes

    - name: Create custom index.html
      copy: content="{{ inventory_hostname }}" dest=/usr/share/nginx/www/index.html
           owner=root group=root mode=0644

```

## RackConnect and Managed Cloud

When using RackConnect version 2 or Rackspace Managed Cloud there are Rackspace automation tasks that are executed on the servers you create after they are successfully built. If your automation executes before the RackConnect or Managed Cloud automation, you can cause failures and un-usable servers.

These examples show creating servers, and ensuring that the Rackspace automation has completed before Ansible continues onwards.

For simplicity, these examples are joined, however both are only needed when using RackConnect. When only using Managed Cloud, the RackConnect portion can be ignored.

The RackConnect portions only apply to RackConnect version 2.

## Using a Control Machine

```

- name: Create an exact count of servers
  hosts: localhost
  connection: local
  gather_facts: False
  tasks:
    - name: Server build requests
      local_action:
        module: rax
        credentials: ~/.raxpub
        name: web%03d.example.org
        flavor: performancel-1
        image: ubuntu-1204-lts-precise-pangolin
        disk_config: manual
        region: DFW
        state: present
        count: 1
        exact_count: yes
        group: web
        wait: yes
        register: rax

    - name: Add servers to in memory groups
      local_action:
        module: add_host
        hostname: "{{ item.name }}"
        ansible_host: "{{ item.rax_accessipv4 }}"
        ansible_ssh_pass: "{{ item.rax_adminpass }}"
        ansible_user: root
        rax_id: "{{ item.rax_id }}"
        groups: web,new_web
        loop: "{{ rax.success }}"
        when: rax.action == 'create'

- name: Wait for rackconnect and managed cloud automation to complete
  hosts: new_web
  gather_facts: false
  tasks:
    - name: Wait for rackconnect automation to complete
      local_action:
        module: rax_facts
        credentials: ~/.raxpub
        id: "{{ rax_id }}"
        region: DFW
        register: rax_facts
        until: rax_facts.ansible_facts['rax_metadata']['rackconnect_automation_status
↪']|default('') == 'DEPLOYED'
        retries: 30
        delay: 10

    - name: Wait for managed cloud automation to complete
      local_action:
        module: rax_facts
        credentials: ~/.raxpub
        id: "{{ rax_id }}"
        region: DFW

```

(continues on next page)

()

```

    register: rax_facts
    until: rax_facts.ansible_facts['rax_metadata']['rax_service_level_automation
↪']|default('') == 'Complete'
    retries: 30
    delay: 10

- name: Update new_web hosts with IP that RackConnect assigns
  hosts: new_web
  gather_facts: false
  tasks:
    - name: Get facts about servers
      local_action:
        module: rax_facts
        name: "{{ inventory_hostname }}"
        region: DFW
    - name: Map some facts
      set_fact:
        ansible_host: "{{ rax_accessipv4 }}"

- name: Base Configure Servers
  hosts: web
  roles:
    - role: users

    - role: openssh
      opensshd_PermitRootLogin: "no"

    - role: ntp

```

## Using Ansible Pull

```

---
- name: Ensure Rackconnect and Managed Cloud Automation is complete
  hosts: all
  connection: local
  tasks:
    - name: Check for completed bootstrap
      stat:
        path: /etc/bootstrap_complete
      register: bootstrap

    - name: Get region
      command: xenstore-read vm-data/provider_data/region
      register: rax_region
      when: bootstrap.stat.exists != True

    - name: Wait for rackconnect automation to complete
      uri:
        url: "https://{{ rax_region.stdout|trim }}.api.rackconnect.rackspace.com/v1/
↪automation_status?format=json"
        return_content: yes
        register: automation_status
        when: bootstrap.stat.exists != True
        until: automation_status['automation_status']|default('') == 'DEPLOYED'

```

(continues on next page)

```

    retries: 30
    delay: 10

- name: Wait for managed cloud automation to complete
  wait_for:
    path: /tmp/rs_managed_cloud_automation_complete
    delay: 10
  when: bootstrap.stat.exists != True

- name: Set bootstrap completed
  file:
    path: /etc/bootstrap_complete
    state: touch
    owner: root
    group: root
    mode: 0400

- name: Base Configure Servers
  hosts: all
  connection: local
  roles:
    - role: users

    - role: openssh
      opensshd_PermitRootLogin: "no"

  - role: ntp

```

## Using Ansible Pull with XenStore

```

---
- name: Ensure Rackconnect and Managed Cloud Automation is complete
  hosts: all
  connection: local
  tasks:
    - name: Check for completed bootstrap
      stat:
        path: /etc/bootstrap_complete
      register: bootstrap

    - name: Wait for rackconnect_automation_status xenstore key to exist
      command: xenstore-exists vm-data/user-metadata/rackconnect_automation_status
      register: rcas_exists
      when: bootstrap.stat.exists != True
      failed_when: rcas_exists.rc|int > 1
      until: rcas_exists.rc|int == 0
      retries: 30
      delay: 10

    - name: Wait for rackconnect automation to complete
      command: xenstore-read vm-data/user-metadata/rackconnect_automation_status
      register: rcas
      when: bootstrap.stat.exists != True
      until: rcas.stdout|replace('"', '') == 'DEPLOYED'

```

(continues on next page)

()

```

    retries: 30
    delay: 10

- name: Wait for rax_service_level_automation xenstore key to exist
  command: xenstore-exists vm-data/user-metadata/rax_service_level_automation
  register: rsla_exists
  when: bootstrap.stat.exists != True
  failed_when: rsla_exists.rc|int > 1
  until: rsla_exists.rc|int == 0
  retries: 30
  delay: 10

- name: Wait for managed cloud automation to complete
  command: xenstore-read vm-data/user-metadata/rackconnect_automation_status
  register: rsla
  when: bootstrap.stat.exists != True
  until: rsla.stdout|replace('"', '') == 'DEPLOYED'
  retries: 30
  delay: 10

- name: Set bootstrap completed
  file:
    path: /etc/bootstrap_complete
    state: touch
    owner: root
    group: root
    mode: 0400

- name: Base Configure Servers
  hosts: all
  connection: local
  roles:
    - role: users

    - role: openssh
      opensshd_PermitRootLogin: "no"

    - role: ntp

```

## 1.14.6 Advanced Usage

### Autoscaling with Tower

*Ansible Tower* also contains a very nice feature for auto-scaling use cases. In this mode, a simple curl script can call a defined URL and the server will "dial out" to the requester and configure an instance that is spinning up. This can be a great way to reconfigure ephemeral nodes. See the Tower documentation for more details.

A benefit of using the callback in Tower over pull mode is that job results are still centrally recorded and less information has to be shared with remote hosts.

### Orchestration in the Rackspace Cloud

Ansible is a powerful orchestration tool, and rax modules allow you the opportunity to orchestrate complex tasks, deployments, and configurations. The key here is to automate provisioning of infrastructure, like any other piece

of software in an environment. Complex deployments might have previously required manual manipulation of load balancers, or manual provisioning of servers. Utilizing the `rax` modules included with Ansible, one can make the deployment of additional nodes contingent on the current number of running nodes, or the configuration of a clustered application dependent on the number of nodes with common metadata. One could automate the following scenarios, for example:

- Servers that are removed from a Cloud Load Balancer one-by-one, updated, verified, and returned to the load balancer pool
- Expansion of an already-online environment, where nodes are provisioned, bootstrapped, configured, and software installed
- A procedure where app log files are uploaded to a central location, like Cloud Files, before a node is decommissioned
- Servers and load balancers that have DNS records created and destroyed on creation and decommissioning, respectively

## 1.15 Continuous Delivery and Rolling Upgrades

### 1.15.1 Introduction

Continuous Delivery is the concept of frequently delivering updates to your software application.

The idea is that by updating more often, you do not have to wait for a specific timed period, and your organization gets better at the process of responding to change.

Some Ansible users are deploying updates to their end users on an hourly or even more frequent basis – sometimes every time there is an approved code change. To achieve this, you need tools to be able to quickly apply those updates in a zero-downtime way.

This document describes in detail how to achieve this goal, using one of Ansible’s most complete example playbooks as a template: `lamp_haproxy`. This example uses a lot of Ansible features: roles, templates, and group variables, and it also comes with an orchestration playbook that can do zero-downtime rolling upgrades of the web application stack.

---

: [Click here for the latest playbooks for this example.](#)

---

The playbooks deploy Apache, PHP, MySQL, Nagios, and HAProxy to a CentOS-based set of servers.

We’re not going to cover how to run these playbooks here. Read the included README in the github project along with the example for that information. Instead, we’re going to take a close look at every part of the playbook and describe what it does.

### 1.15.2 Site Deployment

Let’s start with `site.yml`. This is our site-wide deployment playbook. It can be used to initially deploy the site, as well as push updates to all of the servers:

```
---
# This playbook deploys the whole application stack in this site.

# Apply common configuration to all hosts
- hosts: all
```

(continues on next page)

()

```

roles:
- common

# Configure and deploy database servers.
- hosts: dbservers

    roles:
    - db

# Configure and deploy the web servers. Note that we include two roles
# here, the 'base-apache' role which simply sets up Apache, and 'web'
# which includes our example web application.

- hosts: webservers

    roles:
    - base-apache
    - web

# Configure and deploy the load balancer(s).
- hosts: lbserver

    roles:
    - haproxy

# Configure and deploy the Nagios monitoring node(s).
- hosts: monitoring

    roles:
    - base-apache
    - nagios

```

---

: If you're not familiar with terms like playbooks and plays, you should review [Working With Playbooks](#).

---

In this playbook we have 5 plays. The first one targets all hosts and applies the `common` role to all of the hosts. This is for site-wide things like yum repository configuration, firewall configuration, and anything else that needs to apply to all of the servers.

The next four plays run against specific host groups and apply specific roles to those servers. Along with the roles for Nagios monitoring, the database, and the web application, we've implemented a `base-apache` role that installs and configures a basic Apache setup. This is used by both the sample web application and the Nagios hosts.

### 1.15.3 Reusable Content: Roles

By now you should have a bit of understanding about roles and how they work in Ansible. Roles are a way to organize content: tasks, handlers, templates, and files, into reusable components.

This example has six roles: `common`, `base-apache`, `db`, `haproxy`, `nagios`, and `web`. How you organize your roles is up to you and your application, but most sites will have one or more common roles that are applied to all systems, and then a series of application-specific roles that install and configure particular parts of the site.

Roles can have variables and dependencies, and you can pass in parameters to roles to modify their behavior. You can read more about roles in the [Roles](#) section.

### 1.15.4 Configuration: Group Variables

Group variables are variables that are applied to groups of servers. They can be used in templates and in playbooks to customize behavior and to provide easily-changed settings and parameters. They are stored in a directory called `group_vars` in the same location as your inventory. Here is `lamp_haproxy`'s `group_vars/all` file. As you might expect, these variables are applied to all of the machines in your inventory:

```
---
httpd_port: 80
ntpserver: 192.0.2.23
```

This is a YAML file, and you can create lists and dictionaries for more complex variable structures. In this case, we are just setting two variables, one for the port for the web server, and one for the NTP server that our machines should use for time synchronization.

Here's another group variables file. This is `group_vars/dbservers` which applies to the hosts in the `dbservers` group:

```
---
mysqlservice: mysqld
mysql_port: 3306
dbuser: root
dbname: foodb
upassword: usersecret
```

If you look in the example, there are group variables for the `webservers` group and the `lb_servers` group, similarly.

These variables are used in a variety of places. You can use them in playbooks, like this, in `roles/db/tasks/main.yml`:

```
- name: Create Application Database
  mysql_db:
    name: "{{ dbname }}"
    state: present

- name: Create Application DB User
  mysql_user:
    name: "{{ dbuser }}"
    password: "{{ upassword }}"
    priv: "*.*:ALL"
    host: '%'
    state: present
```

You can also use these variables in templates, like this, in `roles/common/templates/ntp.conf.j2`:

```
driftfile /var/lib/ntp/drift

restrict 127.0.0.1
restrict -6 ::1

server {{ ntpserver }}

includefile /etc/ntp/crypto/pw

keys /etc/ntp/keys
```

You can see that the variable substitution syntax of `{{` and `}}` is the same for both templates and variables. The



syntax inside the curly braces is Jinja2, and you can do all sorts of operations and apply different filters to the data inside. In templates, you can also use for loops and if statements to handle more complex situations, like this, in `roles/common/templates/iptables.j2`:

```
{% if inventory_hostname in groups['dbservers'] %}
-A INPUT -p tcp --dport 3306 -j ACCEPT
{% endif %}
```

This is testing to see if the inventory name of the machine we're currently operating on (`inventory_hostname`) exists in the inventory group `dbservers`. If so, that machine will get an iptables ACCEPT line for port 3306.

Here's another example, from the same template:

```
{% for host in groups['monitoring'] %}
-A INPUT -p tcp -s {{ hostvars[host].ansible_default_ipv4.address }} --dport 5666 -j ACCEPT
{% endfor %}
```

This loops over all of the hosts in the group called `monitoring`, and adds an ACCEPT line for each monitoring hosts' default IPv4 address to the current machine's iptables configuration, so that Nagios can monitor those hosts.

You can learn a lot more about Jinja2 and its capabilities [here](#), and you can read more about Ansible variables in general in the [Variables](#) section.

### 1.15.5 The Rolling Upgrade

Now you have a fully-deployed site with web servers, a load balancer, and monitoring. How do you update it? This is where Ansible's orchestration features come into play. While some applications use the term 'orchestration' to mean basic ordering or command-blasting, Ansible refers to orchestration as 'conducting machines like an orchestra', and has a pretty sophisticated engine for it.

Ansible has the capability to do operations on multi-tier applications in a coordinated way, making it easy to orchestrate a sophisticated zero-downtime rolling upgrade of our web application. This is implemented in a separate playbook, called `rolling_update.yml`.

Looking at the playbook, you can see it is made up of two plays. The first play is very simple and looks like this:

```
- hosts: monitoring
  tasks: []
```

What's going on here, and why are there no tasks? You might know that Ansible gathers "facts" from the servers before operating upon them. These facts are useful for all sorts of things: networking information, OS/distribution versions, etc. In our case, we need to know something about all of the monitoring servers in our environment before we perform the update, so this simple play forces a fact-gathering step on our monitoring servers. You will see this pattern sometimes, and it's a useful trick to know.

The next part is the update play. The first part looks like this:

```
- hosts: webservers
  user: root
  serial: 1
```

This is just a normal play definition, operating on the `webservers` group. The `serial` keyword tells Ansible how many servers to operate on at once. If it's not specified, Ansible will parallelize these operations up to the default "forks" limit specified in the configuration file. But for a zero-downtime rolling upgrade, you may not want to operate on that many hosts at once. If you had just a handful of web servers, you may want to set `serial` to 1, for one host at a time. If you have 100, maybe you could set `serial` to 10, for ten at a time.

Here is the next part of the update play:

```
pre_tasks:
- name: disable nagios alerts for this host webserver service
  nagios:
    action: disable_alerts
    host: "{{ inventory_hostname }}"
    services: webserver
  delegate_to: "{{ item }}"
  loop: "{{ groups.monitoring }}"

- name: disable the server in haproxy
  shell: echo "disable server myaplb/{{ inventory_hostname }}" | socat stdio /var/
↳ lib/haproxy/stats
  delegate_to: "{{ item }}"
  loop: "{{ groups.lbservers }}"
```

---

:

- The `serial` keyword forces the play to be executed in 'batches'. Each batch counts as a full play with a subselection of hosts. This has some consequences on play behavior. For example, if all hosts in a batch fails, the play fails, which in turn fails the entire run. You should consider this when combining with `max_fail_percentage`.

---

The `pre_tasks` keyword just lets you list tasks to run before the roles are called. This will make more sense in a minute. If you look at the names of these tasks, you can see that we are disabling Nagios alerts and then removing the webserver that we are currently updating from the HAProxy load balancing pool.

The `delegate_to` and `loop` arguments, used together, cause Ansible to loop over each monitoring server and load balancer, and perform that operation (delegate that operation) on the monitoring or load balancing server, "on behalf" of the webserver. In programming terms, the outer loop is the list of web servers, and the inner loop is the list of monitoring servers.

Note that the HAProxy step looks a little complicated. We're using HAProxy in this example because it's freely available, though if you have (for instance) an F5 or Netscaler in your infrastructure (or maybe you have an AWS Elastic IP setup?), you can use modules included in core Ansible to communicate with them instead. You might also wish to use other monitoring modules instead of nagios, but this just shows the main goal of the 'pre tasks' section – take the server out of monitoring, and take it out of rotation.

The next step simply re-applies the proper roles to the web servers. This will cause any configuration management declarations in `web` and `base-apache` roles to be applied to the web servers, including an update of the web application code itself. We don't have to do it this way—we could instead just purely update the web application, but this is a good example of how roles can be used to reuse tasks:

```
roles:
- common
- base-apache
- web
```

Finally, in the `post_tasks` section, we reverse the changes to the Nagios configuration and put the web server back in the load balancing pool:

```
post_tasks:
- name: Enable the server in haproxy
  shell: echo "enable server myaplb/{{ inventory_hostname }}" | socat stdio /var/lib/
↳ haproxy/stats
```

(continues on next page)

()

```

delegate_to: "{{ item }}"
loop: "{{ groups.lbservers }}"

- name: re-enable nagios alerts
  nagios:
    action: enable_alerts
    host: "{{ inventory_hostname }}"
    services: webserver
  delegate_to: "{{ item }}"
  loop: "{{ groups.monitoring }}"

```

Again, if you were using a Netscaler or F5 or Elastic Load Balancer, you would just substitute in the appropriate modules instead.

### 1.15.6 Managing Other Load Balancers

In this example, we use the simple HAProxy load balancer to front-end the web servers. It's easy to configure and easy to manage. As we have mentioned, Ansible has built-in support for a variety of other load balancers like Citrix NetScaler, F5 BigIP, Amazon Elastic Load Balancers, and more. See the *Working With Modules* documentation for more information.

For other load balancers, you may need to send shell commands to them (like we do for HAProxy above), or call an API, if your load balancer exposes one. For the load balancers for which Ansible has modules, you may want to run them as a `local_action` if they contact an API. You can read more about local actions in the *Delegation, Rolling Updates, and Local Actions* section. Should you develop anything interesting for some hardware where there is not a core module, it might make for a good module for core inclusion!

### 1.15.7 Continuous Delivery End-To-End

Now that you have an automated way to deploy updates to your application, how do you tie it all together? A lot of organizations use a continuous integration tool like [Jenkins](#) or [Atlassian Bamboo](#) to tie the development, test, release, and deploy steps together. You may also want to use a tool like [Gerrit](#) to add a code review step to commits to either the application code itself, or to your Ansible playbooks, or both.

Depending on your environment, you might be deploying continuously to a test environment, running an integration test battery against that environment, and then deploying automatically into production. Or you could keep it simple and just use the rolling-update for on-demand deployment into test or production specifically. This is all up to you.

For integration with Continuous Integration systems, you can easily trigger playbook runs using the `ansible-playbook` command line tool, or, if you're using [Ansible Tower](#), the `tower-cli` or the built-in REST API. (The `tower-cli` command `'joblaunch'` will spawn a remote job over the REST API and is pretty slick).

This should give you a good idea of how to structure a multi-tier application with Ansible, and orchestrate operations upon that app, with the eventual goal of continuous delivery to your customers. You could extend the idea of the rolling upgrade to lots of different parts of the app; maybe add front-end web servers along with application servers, for instance, or replace the SQL database with something like MongoDB or Riak. Ansible gives you the capability to easily manage complicated environments and automate common operations.

:

**lamp\_haproxy example** The `lamp_haproxy` example discussed here.

**Working With Playbooks** An introduction to playbooks

**Roles** An introduction to playbook roles

*Variables* An introduction to Ansible variables

**Ansible.com: Continuous Delivery** An introduction to Continuous Delivery with Ansible

## 1.16 Using Vagrant and Ansible

### 1.16.1 Introduction

**Vagrant** is a tool to manage virtual machine environments, and allows you to configure and use reproducible work environments on top of various virtualization and cloud platforms. It also has integration with Ansible as a provisioner for these virtual machines, and the two tools work together well.

This guide will describe how to use Vagrant 1.7+ and Ansible together.

If you're not familiar with Vagrant, you should visit [the documentation](#).

This guide assumes that you already have Ansible installed and working. Running from a Git checkout is fine. Follow the *Installation Guide* guide for more information.

### 1.16.2 Vagrant Setup

The first step once you've installed Vagrant is to create a `Vagrantfile` and customize it to suit your needs. This is covered in detail in the Vagrant documentation, but here is a quick example that includes a section to use the Ansible provisioner to manage a single machine:

```
# This guide is optimized for Vagrant 1.7 and above.
# Although versions 1.6.x should behave very similarly, it is recommended
# to upgrade instead of disabling the requirement below.
Vagrant.require_version ">= 1.7.0"

Vagrant.configure(2) do |config|

  config.vm.box = "ubuntu/trusty64"

  # Disable the new default behavior introduced in Vagrant 1.7, to
  # ensure that all Vagrant machines will use the same SSH key pair.
  # See https://github.com/mitchellh/vagrant/issues/5005
  config.ssh.insert_key = false

  config.vm.provision "ansible" do |ansible|
    ansible.verbosity = "v"
    ansible.playbook = "playbook.yml"
  end
end
```

Notice the `config.vm.provision` section that refers to an Ansible playbook called `playbook.yml` in the same directory as the `Vagrantfile`. Vagrant runs the provisioner once the virtual machine has booted and is ready for SSH access.

There are a lot of Ansible options you can configure in your `Vagrantfile`. Visit the [Ansible Provisioner documentation](#) for more information.

```
$ vagrant up
```

This will start the VM, and run the provisioning playbook (on the first VM startup).

To re-run a playbook on an existing VM, just run:

```
$ vagrant provision
```

This will re-run the playbook against the existing VM.

Note that having the `ansible.verbose` option enabled will instruct Vagrant to show the full `ansible-playbook` command used behind the scene, as illustrated by this example:

```
$ PYTHONUNBUFFERED=1 ANSIBLE_FORCE_COLOR=true ANSIBLE_HOST_KEY_CHECKING=false ANSIBLE_
↪SSH_ARGS='-o UserKnownHostsFile=/dev/null -o ControlMaster=auto -o_
↪ControlPersist=60s' ansible-playbook --private-key=/home/someone/.vagrant.d/
↪insecure_private_key --user=vagrant --connection=ssh --limit='machine1' --inventory-
↪file=/home/someone/coding-in-a-project/.vagrant/provisioners/ansible/inventory/
↪vagrant_ansible_inventory playbook.yml
```

This information can be quite useful to debug integration issues and can also be used to manually execute Ansible from a shell, as explained in the next section.

### 1.16.3 Running Ansible Manually

Sometimes you may want to run Ansible manually against the machines. This is faster than kicking `vagrant provision` and pretty easy to do.

With our Vagrantfile example, Vagrant automatically creates an Ansible inventory file in `.vagrant/provisioners/ansible/inventory/vagrant_ansible_inventory`. This inventory is configured according to the SSH tunnel that Vagrant automatically creates. A typical automatically-created inventory file for a single machine environment may look something like this:

```
# Generated by Vagrant

default ansible_ssh_host=127.0.0.1 ansible_ssh_port=2222
```

If you want to run Ansible manually, you will want to make sure to pass `ansible` or `ansible-playbook` commands the correct arguments, at least for the *username*, the *SSH private key* and the *inventory*.

Here is an example using the Vagrant global insecure key (`config.ssh.insert_key` must be set to `false` in your Vagrantfile):

```
$ ansible-playbook --private-key=~/.vagrant.d/insecure_private_key -u vagrant -i .
↪vagrant/provisioners/ansible/inventory/vagrant_ansible_inventory playbook.yml
```

Here is a second example using the random private key that Vagrant 1.7+ automatically configures for each new VM (each key is stored in a path like `.vagrant/machines/[machine name]/[provider]/private_key`):

```
$ ansible-playbook --private-key=.vagrant/machines/default/virtualbox/private_key -u_
↪vagrant -i .vagrant/provisioners/ansible/inventory/vagrant_ansible_inventory_
↪playbook.yml
```

### 1.16.4 Advanced Usages

The "Tips and Tricks" chapter of the [Ansible Provisioner documentation](#) provides detailed information about more advanced Ansible features like:

- how to parallelly execute a playbook in a multi-machine environment
- how to integrate a local `ansible.cfg` configuration file

:

**Vagrant Home** The Vagrant homepage with downloads

**Vagrant Documentation** Vagrant Documentation

**Ansible Provisioner** The Vagrant documentation for the Ansible provisioner

**Vagrant Issue Tracker** The open issues for the Ansible provisioner in the Vagrant project

**Working With Playbooks** An introduction to playbooks

## 1.17 Getting Started with VMware

### 1.17.1 Introduction

Ansible provides various modules to manage VMware infrastructure, which includes datacenter, cluster, host system and virtual machine.

### 1.17.2 Requirements

Ansible VMware modules are written on top of [pyVmomi](#). pyVmomi is the Python SDK for the VMware vSphere API that allows user to manage ESX, ESXi, and vCenter infrastructure. You can install pyVmomi using pip:

```
$ pip install pyvmomi
```

### 1.17.3 vmware\_guest module

The `vmware_guest` module manages various operations related to virtual machines in the given ESXi or vCenter server.

**Prior to Ansible version 2.5, `folder` was an optional parameter with a default value of `/vm`. The `folder` parameter was used to discover information about virtual machines in the given infrastructure.**

Starting with Ansible version 2.5, `folder` is still an optional parameter with no default value. This parameter will be now used to identify a user's virtual machine, if multiple virtual machines or virtual machine templates are found with same name. VMware does not restrict the system administrator from creating virtual machines with same name.

### 1.17.4 Debugging

When debugging or creating a new issue, you will need information about your VMware infrastructure. You can get this information using `govc`. For example:

```
$ export GOVC_USERNAME=ESXI_OR_VCENTER_USERNAME
$ export GOVC_PASSWORD=ESXI_OR_VCENTER_PASSWORD
$ export GOVC_URL=https://ESXI_OR_VCENTER_HOSTNAME:443
$ govc find /
```

:

**pyVmomi** The GitHub Page of pyVmomi

**pyVmomi Issue Tracker** The issue tracker for the pyVmomi project

**govc** govc is a vSphere CLI built on top of govmmomi

*Working With Playbooks* An introduction to playbooks

## 1.18 Ansible for VMware

Welcome to the Ansible for VMware Guide! This is a set of placeholder docs to be filled in.

The purpose of this guide is to teach you everything you need to know about using Ansible with VMware.

To get started, please select one of the following topics.

### 1.18.1 Introduction to Ansible for VMware

Make the case. What does it do?

- Cool thing 1
- Cool thing 2
- Cool thing 3

### 1.18.2 Ansible for VMware Concepts

Introduction...blah blah

#### Concept 1

Explanation goes here.

#### Concept 2

Explanation goes here.

### 1.18.3 VMware Prerequisites

This is what you'll need to get started...

### 1.18.4 Getting Started with Ansible for VMware

This will have a basic "hello world" scenario/walkthrough that gets the user introduced to the basics.

### 1.18.5 Ansible for VMware Scenarios

Welcome to the Ansible for VMWare Guide!

The purpose of this guide is to teach you everything you need to know about using Ansible with VMWare.

To get started, please select one of the following topics.

### Deploy a virtual machine from a template

#### Introduction

This guide will show you how to utilize Ansible to clone a virtual machine from already existing VMware template.

#### Scenario Requirements

- Software
  - Ansible 2.5 or later must be installed
  - The Python module *Pyvmomi* must be installed on the Ansible (or Target host if not executing against localhost)
  - Installing the latest *Pyvmomi* via pip is recommended [as the OS packages are usually out of date and incompatible]
- Hardware
  - At least one standalone ESXi server or
  - vCenter Server with at least one ESXi server
- Access / Credentials
  - Ansible (or the target server) must have network access to the either vCenter server or the ESXi server you will be deploying to
  - Username and Password
  - Administrator user with following privileges
    - \* Virtual machine.Provisioning.Clone virtual machine on the virtual machine you are cloning
    - \* Virtual machine.Inventory.Create from existing on the datacenter or virtual machine folder
    - \* Virtual machine.Configuration.Add new disk on the datacenter or virtual machine folder.
    - \* Resource.Assign virtual machine to resource pool on the destination host, cluster, or resource pool.
    - \* Datastore.Allocate space on the destination datastore or datastore folder.
    - \* Network.Assign network on the network to which the virtual machine will be assigned.
    - \* Virtual machine.Provisioning.Customize on the virtual machine or virtual machine folder if you are customizing the guest operating system.
    - \* Virtual machine.Provisioning.Read customization specifications on the root vCenter Server if you are customizing the guest operating system.

#### Assumptions

- All variable names and VMware object names are case sensitive
- VMware allows creation of virtual machine and templates with same name across datacenters and within datacenters.
- You need to use Python 2.7.9 version in order to use *validate\_certs* option, as this version is capable of changing the SSL verification behaviours.



## Caveats

- Hosts in the ESXi cluster must have access to the datastore that the template resides on.
- Multiple templates with the same name will cause module failures.
- In order to utilize Guest Customization, VMWare Tools must be installed on the template. For Linux, the `open-vm-tools` package is recommended, and it requires that Perl be installed.

## Example Description

In this use case / example, we will be selecting a virtual machine template and cloning it into a specific folder in our Datacenter / Cluster. The following Ansible playbook showcases the basic parameters that are needed for this.

```
---
- name: Create a VM from a template
  hosts: localhost
  connection: local
  gather_facts: no
  tasks:
  - name: Clone the template
    vmware_guest:
      hostname: 192.0.2.44
      username: administrator@vsphere.local
      password: vmware
      validate_certs: False
      name: testvm_2
      template: template_el7
      datacenter: DC1
      folder: /DC1/vm
      state: poweredon
      wait_for_ip_address: yes
```

Since Ansible utilizes the VMware API to perform actions, in this use case we will be connecting directly to the API from our localhost. This means that our playbooks will not be running from the vCenter or ESXi Server. We do not necessarily need to collect facts about our localhost, so the `gather_facts` parameter will be disabled. You can run these modules against another server that would then connect to the API if your localhost does not have access to vCenter. If so, the required Python modules will need to be installed on that target server.

To begin, there are a few bits of information we will need. First and foremost is the hostname of the ESXi server or vCenter server. After this, you will need the username and password for this server. For now, you will be entering these directly, but in a more advanced playbook this can be abstracted out and stored in a more secure fashion [1][2]. If your vCenter or ESXi server is not setup with proper CA certificates that can be verified from the Ansible server, then it is necessary to disable validation of these certificates by using the `validate_certs` parameter. To do this you need to set `validate_certs=False` in your playbook.

Now you need to supply the information about the virtual machine which will be created. Give your virtual machine a name, one that conforms to all VMware requirements for naming conventions. Next, select the display name of the template from which you want to clone new virtual machine. This must match what's displayed in VMware Web UI exactly. Then you can specify a folder to place this new virtual machine in. This path can either be a relative path or a full path to the folder including the Datacenter. You may need to specify a state for the virtual machine. This simply tells the module which action you want to take, in this case you will ensure that the virtual machine exists and is powered on. An optional parameter is `wait_for_ip_address`, this will tell Ansible to wait for the virtual machine to fully boot up and VMware Tools is running before completing this task.

## What to expect

- You will see a bit of JSON output after this playbook completes. This output shows various parameters that are returned from the module and from vCenter about the newly created VM.

```
{
  "changed": true,
  "instance": {
    "annotation": "",
    "current_snapshot": null,
    "customvalues": {},
    "guest_consolidation_needed": false,
    "guest_question": null,
    "guest_tools_status": "guestToolsNotRunning",
    "guest_tools_version": "0",
    "hw_cores_per_socket": 1,
    "hw_datastores": [
      "ds_215"
    ],
    "hw_esxi_host": "192.0.2.44",
    "hw_eth0": {
      "addresstype": "assigned",
      "ipaddresses": null,
      "label": "Network adapter 1",
      "macaddress": "00:50:56:8c:19:f4",
      "macaddress_dash": "00-50-56-8c-19-f4",
      "portgroup_key": "dvportgroup-17",
      "portgroup_portkey": "0",
      "summary": "DVSwitch: 50 0c 5b 22 b6 68 ab 89-fc 0b 59 a4 08 6e 80 fa"
    },
    "hw_files": [
      "[ds_215] testvm_2/testvm_2.vmx",
      "[ds_215] testvm_2/testvm_2.vmsd",
      "[ds_215] testvm_2/testvm_2.vmdk"
    ],
    "hw_folder": "/DC1/vm",
    "hw_guest_full_name": null,
    "hw_guest_ha_state": null,
    "hw_guest_id": null,
    "hw_interfaces": [
      "eth0"
    ],
    "hw_is_template": false,
    "hw_memtotal_mb": 512,
    "hw_name": "testvm_2",
    "hw_power_status": "poweredOff",
    "hw_processor_count": 2,
    "hw_product_uuid": "420cb25b-81e8-8d3b-dd2d-a439ee54fcc5",
    "hw_version": "vmx-13",
    "instance_uuid": "500cd53b-ed57-d74e-2da8-0dc0eddf54d5",
    "ipv4": null,
    "ipv6": null,
    "module_hw": true,
    "snapshots": []
  },
  "invocation": {
    "module_args": {
```

(continues on next page)

()

```

        "annotation": null,
        "cdrom": {},
        "cluster": "DC1_C1",
        "customization": {},
        "customization_spec": null,
        "customvalues": [],
        "datacenter": "DC1",
        "disk": [],
        "esxi_hostname": null,
        "folder": "/DC1/vm",
        "force": false,
        "guest_id": null,
        "hardware": {},
        "hostname": "192.0.2.44",
        "is_template": false,
        "linked_clone": false,
        "name": "testvm_2",
        "name_match": "first",
        "networks": [],
        "password": "VALUE_SPECIFIED_IN_NO_LOG_PARAMETER",
        "port": 443,
        "resource_pool": null,
        "snapshot_src": null,
        "state": "present",
        "state_change_timeout": 0,
        "template": "template_el7",
        "username": "administrator@vsphere.local",
        "uuid": null,
        "validate_certs": false,
        "vapp_properties": [],
        "wait_for_ip_address": true
    }
}

```

- State is changed to *True* which notifies that the virtual machine is built using given template. The module will not complete until the clone task in VMware is finished. This can take some time depending on your environment.
- If you utilize the *wait\_for\_ip\_address* parameter, then it will also increase the clone time as it will wait until virtual machine boots into the OS and an IP Address has been assigned to the given NIC.

## Troubleshooting

### Things to inspect

- Check if the values provided for username and password are correct
- Check if the datacenter you provided is available
- Check if the template specified exists and you have permissions to access the datastore
- Ensure the full folder path you specified already exists. It will not create folders automatically for you.

## Appendix

- [1] - <https://docs.ansible.com/ansible/latest/vault.html>

- [2] - <https://docs.ansible.com/ansible-tower/latest/html/userguide/credentials.html>

### Remove an existing VMware virtual machine

#### Topics

- *Remove an existing VMware virtual machine*
  - *Introduction*
  - *Scenario Requirements*
  - *Caveats*
  - *Example Description*
    - \* *What to expect*
    - \* *Troubleshooting*

#### Introduction

This guide will show you how to utilize Ansible to remove an existing VMware virtual machine.

#### Scenario Requirements

- Software
  - Ansible 2.5 or later must be installed.
  - The Python module `Pyvmomi` must be installed on the Ansible control node (or Target host if not executing against localhost).
  - We recommend installing the latest version with pip: `pip install Pyvmomi` (as the OS packages are usually out of date and incompatible).
- Hardware
  - At least one standalone ESXi server or
  - vCenter Server with at least one ESXi server
- Access / Credentials
  - Ansible (or the target server) must have network access to the either vCenter server or the ESXi server
  - Username and Password for vCenter or ESXi server
  - Hosts in the ESXi cluster must have access to the datastore that the template resides on.

#### Caveats

- All variable names and VMware object names are case sensitive.
- You need to use Python 2.7.9 version in order to use `validate_certs` option, as this version is capable of changing the SSL verification behaviours.

- `vmware_guest` module tries to mimick VMware Web UI and workflow, so the virtual machine must be in powered off state in order to remove it from the VMware inventory.

: The removal VMware virtual machine using `vmware_guest` module is destructive operation and can not be reverted, so it is strongly recommended to take the backup of virtual machine and related files (vmx and vmdk files) before proceeding.

## Example Description

In this use case / example, user will be removing a virtual machine using name. The following Ansible playbook showcases the basic parameters that are needed for this.

```
---
- name: Remove virtual machine
  gather_facts: no
  vars_files:
    - vcenter_vars.yml
  vars:
    ansible_python_interpreter: "/usr/bin/env python3"
  hosts: localhost
  tasks:
    - set_fact:
        vm_name: "VM_0003"
        datacenter: "DC1"

    - name: Remove "{{ vm_name }}"
      vmware_guest:
        hostname: "{{ vcenter_server }}"
        username: "{{ vcenter_user }}"
        password: "{{ vcenter_pass }}"
        validate_certs: no
        cluster: "DC1_C1"
        name: "{{ vm_name }}"
        state: absent
        delegate_to: localhost
        register: facts
```

Since Ansible utilizes the VMware API to perform actions, in this use case it will be connecting directly to the API from localhost.

This means that playbooks will not be running from the vCenter or ESXi Server.

Note that this play disables the `gather_facts` parameter, since you don't want to collect facts about localhost.

You can run these modules against another server that would then connect to the API if localhost does not have access to vCenter. If so, the required Python modules will need to be installed on that target server. We recommend installing the latest version with pip: `pip install Pyvmomi` (as the OS packages are usually out of date and incompatible).

Before you begin, make sure you have:

- Hostname of the ESXi server or vCenter server
- Username and password for the ESXi or vCenter server
- Name of the existing Virtual Machine you want to remove

For now, you will be entering these directly, but in a more advanced playbook this can be abstracted out and stored in a more secure fashion using `ansible-vault` or using [Ansible Tower credentials](#).

If your vCenter or ESXi server is not setup with proper CA certificates that can be verified from the Ansible server, then it is necessary to disable validation of these certificates by using the `validate_certs` parameter. To do this you need to set `validate_certs=False` in your playbook.

The name of existing virtual machine will be used as input for `vmware_guest` module via `name` parameter.

### What to expect

- You will not see any JSON output after this playbook completes as compared to other operations performed using `vmware_guest` module.

```
{
  "changed": true
}
```

- State is changed to `True` which notifies that the virtual machine is removed from the VMware inventory. This can take some time depending upon your environment and network connectivity.

### Troubleshooting

If your playbook fails:

- Check if the values provided for username and password are correct.
- Check if the datacenter you provided is available.
- Check if the virtual machine specified exists and you have permissions to access the datastore.
- Ensure the full folder path you specified already exists. It will not create folders automatically for you.

### Sample Scenario for Ansible VMware

Introductory paragraph.

### Scenario Requirements

Describe the requirements and assumptions for this scenario.

### Example Description

Description and code here.

### Example Output

What the user should expect to see.

### Troubleshooting

What to look for if it breaks.

## Conclusion and Where To Go Next

Blah blah for more information please see blah blah...

### 1.18.6 Ansible VMware Module Guide

This will be a listing similar to the module index in our core docs.

### 1.18.7 Troubleshooting Ansible for VMware

This section lists things that can go wrong and possible ways to fix them.

#### Troubleshooting Item

Description

#### Potential Workaround

How to fix...

### 1.18.8 List of useful links to VMware

Following is the list of various external documentation and guides which can help in further readings.

- [PyVmomi Documentation](#)
- [VMware API and SDK Documentation](#)
- [VCSIM test container image](#)
- [Ansible VMware community wiki page](#)
- [VMware's official Guest Operating system customization matrix](#)
- [VMware Compatibility Guide](#)

### 1.18.9 Ansible VMware FAQ

#### `vmware_guest`

#### Can I deploy a virtual machine on a standalone ESXi server ?

Yes. `vmware_guest` can deploy a virtual machine with required settings on a standalone ESXi server.

## 1.19 Ansible for Network Automation

### 1.19.1 Introduction

Ansible Network modules extend the benefits of simple, powerful, agentless automation to network administrators and teams. Ansible Network modules can configure your network stack, test and validate existing network state, and discover and correct network configuration drift.

If you're new to Ansible, or new to using Ansible for network management, start with the Getting Started Guide.

For documentation on using a particular network module, consult the list of all network modules. Some network modules are maintained by the Ansible community - here's a list of network modules maintained by the Ansible Network Team.

### An Introduction to Network Automation with Ansible

Ansible modules support a wide range of vendors, device types, and actions, so you can manage your entire network with a single automation tool. With Ansible, you can:

- Automate repetitive tasks to speed routine network changes and free up your time for more strategic work
- Leverage the same simple, powerful, and agentless automation tool for network tasks that operations and development use
- Separate the data model (in a playbook or role) from the execution layer (via Ansible modules) to manage heterogeneous network devices
- Benefit from community and vendor-generated sample playbooks and roles to help accelerate network automation projects
- Communicate securely with network hardware over SSH or HTTPS

### Who should use this guide?

This guide is intended for network engineers using Ansible for the first time. If you understand networks but have never used Ansible, work through the guide from start to finish.

This guide is also useful for experienced Ansible users automating network tasks for the first time. You can use Ansible commands, playbooks and modules to configure hubs, switches, routers, bridges and other network devices. But network modules are different from Linux/Unix and Windows modules, and you must understand some network-specific concepts to succeed. If you understand Ansible but have never automated a network task, start with the second section.

This guide introduces basic Ansible concepts and guides you through your first Ansible commands, playbooks and inventory entries.

### Basic Concepts

These concepts are common to all uses of Ansible, including network automation. You need to understand them to use Ansible for network automation. This basic introduction provides the background you need to follow the examples in this guide.

<b>Topics</b>
---------------



- *Basic Concepts*
  - *Control Node*
  - *Managed Nodes*
  - *Inventory*
  - *Modules*
  - *Tasks*
  - *Playbooks*

## Control Node

Any machine with Ansible installed. You can run commands and playbooks, invoking `/usr/bin/ansible` or `/usr/bin/ansible-playbook`, from any control node. You can use any computer that has Python installed on it as a control node - laptops, shared desktops, and servers can all run Ansible. However, you cannot use a Windows machine as a control node. You can have multiple control nodes.

## Managed Nodes

The network devices (and/or servers) you manage with Ansible. Managed nodes are also sometimes called "hosts". Ansible is not installed on managed nodes.

## Inventory

A list of managed nodes. An inventory file is also sometimes called a "hostfile". Your inventory can specify information like IP address for each managed node. An inventory can also organize managed nodes, creating and nesting groups for easier scaling. To learn more about inventory, see [the \*Working with Inventory\* pages](#).

## Modules

The units of code Ansible executes. Each module has a particular use, from administering users on a specific type of database to managing VLAN interfaces on a specific type of network device. You can invoke a single module with a task, or invoke several different modules in a playbook. For an idea of how many modules Ansible includes, take a look at the list of all modules or the list of network modules.

## Tasks

The units of action in Ansible. You can execute a single task once with an ad-hoc command.

## Playbooks

Ordered lists of tasks, saved so you can run those tasks in that order repeatedly. Playbooks can include variables as well as tasks. Playbooks are written in YAML and are easy to read, write, share and understand. To learn more about playbooks, see [Intro to Playbooks](#).

## How Network Automation is Different

Network automation leverages the basic Ansible concepts, but there are important differences in how the network modules work. This introduction prepares you to understand the exercises in this guide.

### Topics

- *How Network Automation is Different*
  - *Execution on the Control Node*
  - *Multiple Communication Protocols*
  - *Modules Organized by Network Platform*
  - *Privilege Escalation: `enable` mode, `become`, and `authorize`*
    - \* *Using `become` for privilege escalation*
    - \* *Legacy playbooks: `authorize` for privilege escalation*

## Execution on the Control Node

Unlike most Ansible modules, network modules do not run on the managed nodes. From a user's point of view, network modules work like any other modules. They work with ad-hoc commands, playbooks, and roles. Behind the scenes, however, network modules use a different methodology than the other (Linux/Unix and Windows) modules use. Ansible is written and executed in Python. Because the majority of network devices can not run Python, the Ansible network modules are executed on the Ansible control node, where `ansible` or `ansible-playbook` runs.

Network modules also use the control node as a destination for backup files, for those modules that offer a `backup` option. With Linux/Unix modules, where a configuration file already exists on the managed node(s), the backup file gets written by default in the same directory as the new, changed file. Network modules do not update configuration files on the managed nodes, because network configuration is not written in files. Network modules write backup files on the control node, usually in the *backup* directory under the playbook root directory.

## Multiple Communication Protocols

Because network modules execute on the control node instead of on the managed nodes, they can support multiple communication protocols. The communication protocol (XML over SSH, CLI over SSH, API over HTTPS) selected for each network module depends on the platform and the purpose of the module. Some network modules support only one protocol; some offer a choice. The most common protocol is CLI over SSH. You set the communication protocol with the `ansible_connection` variable:

Value of <code>ansible_connection</code>	Protocol	Requires	Persistent?
<code>network_cli</code>	CLI over SSH	<code>network_os</code> setting	yes
<code>netconf</code>	XML over SSH	<code>network_os</code> setting	yes
<code>httpapi</code>	API over HTTP/HTTPS	<code>network_os</code> setting	yes
<code>local</code>	depends on provider	provider setting	no

Beginning with Ansible 2.6, we recommend using one of the persistent connection types listed above instead of `local`. With persistent connections, you can define the hosts and credentials only once, rather than in every task. For more details on using each connection type on various platforms, see the *platform-specific* pages.

## Modules Organized by Network Platform

A network platform is a set of network devices with a common operating system that can be managed by a collection of modules. The modules for each network platform share a prefix, for example:

- Arista: `eos_`
- Cisco: `ios_`, `iosxr_`, `nxos_`
- Juniper: `junos_`
- VyOS `vyos_`

All modules within a network platform share certain requirements. Some network platforms have specific differences - see the *platform-specific* documentation for details.

## Privilege Escalation: `enable` mode, `become`, and `authorize`

Several network platforms support privilege escalation, where certain tasks must be done by a privileged user. On network devices this is called `enable` mode (the equivalent of `sudo` in \*nix administration). Ansible network modules offer privilege escalation for those network devices that support it. For details of which platforms support `enable` mode, with examples of how to use it, see the *platform-specific* documentation.

## Using `become` for privilege escalation

As of Ansible 2.6, you can use the top-level Ansible parameter `become: yes` with `become_method: enable` to run a task, play, or playbook with escalated privileges on any network platform that supports privilege escalation. You must use either `connection: network_cli` or `connection: httpapi` with `become: yes` with `become_method: enable`. If you are using `network_cli` to connect Ansible to your network devices, a `group_vars` file would look like:

```
ansible_connection: network_cli
ansible_network_os: ios
ansible_become: yes
ansible_become_method: enable
```

## Legacy playbooks: `authorize` for privilege escalation

If you are running Ansible 2.5 or older, some network platforms support privilege escalation but not `network_cli` or `httpapi` connections. This includes all platforms in versions 2.4 and older, and HTTPS connections using `eapi` in version 2.5. With a `local` connection, you must use a provider dictionary and include `authorize: yes` and `auth_pass: my_enable_password`. For that use case, a `group_vars` file looks like:

```
ansible_connection: local
ansible_network_os: eos
# provider settings
eapi:
  authorize: yes
```

(continues on next page)

()

```
auth_pass: " {{ secret_auth_pass }}"
port: 80
transport: eapi
use_ssl: no
```

And you use the `eapi` variable in your task(s):

```
tasks:
- name: provider demo with eos
  eos_banner:
    banner: motd
    text: |
      this is test
      of multiline
      string
    state: present
    provider: "{{ eapi }}"
```

Note that while Ansible 2.6 supports the use of `connection: local` with provider dictionaries, this usage will be deprecated in future and eventually removed.

For more information, see *Become and Networks*

## Run Your First Command and Playbook

Put the concepts you learned to work with this quick tutorial. Install Ansible, execute a network configuration command manually, execute the same command with Ansible, then create a playbook so you can execute the command any time on multiple network devices.

### Topics

- *Run Your First Command and Playbook*
  - *Prerequisites*
  - *Install Ansible*
  - *Establish a Manual Connection to a Managed Node*
  - *Run Your First Network Ansible Command*
  - *Create and Run Your First Network Ansible Playbook*

## Prerequisites

Before you work through this tutorial you need:

- Ansible 2.5 (or higher) installed
- One or more network devices that are compatible with Ansible
- Basic Linux command line knowledge
- Basic knowledge of network switch & router configuration

## Install Ansible

Install Ansible using your preferred method. See [Installation Guide](#). Then return to this tutorial.

Confirm the version of Ansible (must be  $\geq 2.5$ ):

```
ansible --version
```

## Establish a Manual Connection to a Managed Node

To confirm your credentials, connect to a network device manually and retrieve its configuration. Replace the sample user and device name with your real credentials. For example, for a VyOS router:

```
ssh my_vyos_user@vyos.example.net
show config
exit
```

This manual connection also establishes the authenticity of the network device, adding its RSA key fingerprint to your list of known hosts. (If you have connected to the device before, you have already established its authenticity.)

## Run Your First Network Ansible Command

Instead of manually connecting and running a command on the network device, you can retrieve its configuration with a single, stripped-down Ansible command:

```
ansible all -i vyos.example.net, -c network_cli -u my_vyos_user -k -m vyos_facts -e_
↪ansible_network_os=vyos
```

**The flags in this command set seven values:**

- the host group(s) to which the command should apply (in this case, all)
- the inventory (-i, the device or devices to target - without the trailing comma -i points to an inventory file)
- the connection method (-c, the method for connecting and executing ansible)
- the user (-u, the username for the SSH connection)
- the SSH connection method (-k, please prompt for the password)
- the module (-m, the ansible module to run)
- an extra variable (-e, in this case, setting the network OS value)

NOTE: If you use `ssh-agent` with ssh keys, Ansible loads them automatically. You can omit `-k` flag.

## Create and Run Your First Network Ansible Playbook

If you want to run this command every day, you can save it in a playbook and run it with `ansible-playbook` instead of `ansible`. The playbook can store a lot of the parameters you provided with flags at the command line, leaving less to type at the command line. You need two files for this - a playbook and an inventory file.

1. Download `first_playbook.yml`, which looks like this:

```

---
- name: Network Getting Started First Playbook
  connection: network_cli
  hosts: all
  tasks:

    - name: Get config for VyOS devices
      vyos_facts:
        gather_subset: all

    - name: Display the config
      debug:
        msg: "The hostname is {{ ansible_net_hostname }} and the OS is {{ ansible_net_
↪version }}"

```

The playbook sets three of the seven values from the command line above: the group (`hosts: all`), the connection method (`connection: network_cli`) and the module (in each task). With those values set in the playbook, you can omit them on the command line. The playbook also adds a second task to show the config output. When a module runs in a playbook, the output is held in memory for use by future tasks instead of written to the console. The debug task here lets you see the results in your shell.

2. Run the playbook with the command:

```

ansible-playbook -i vyos.example.net, -u ansible -k -e ansible_network_os=vyos first_
↪playbook.yml

```

The playbook contains one play with two tasks, and should generate output like this:

```

$ ansible-playbook -i vyos.example.net, -u ansible -k -e ansible_network_os=vyos_
↪first_playbook.yml

PLAY [First Playbook]
*****

TASK [Gathering Facts]
*****
ok: [vyos.example.net]

TASK [Get config for VyOS devices]
*****
ok: [vyos.example.net]

TASK [Display the config]
*****
ok: [vyos.example.net] => {
  "failed": false,
  "msg": "The hostname is vyos and the OS is VyOS"
}

```

3. Now that you can retrieve the device config, try updating it with Ansible. Download `first_playbook_ext.yml`, which is an extended version of the first playbook:

```

---
- name: Network Getting Started First Playbook Extended
  connection: network_cli

```

(continues on next page)

()

```

hosts: all
tasks:

  - name: Get config for VyOS devices
    vyos_facts:
      gather_subset: all

  - name: Display the config
    debug:
      msg: "The hostname is {{ ansible_net_hostname }} and the OS is {{ ansible_net_
↪version }}"

  - name: Update the hostname
    vyos_config:
      backup: yes
      lines:
        - set system host-name vyos-changed

  - name: Get changed config for VyOS devices
    vyos_facts:
      gather_subset: all

  - name: Display the changed config
    debug:
      msg: "The hostname is {{ ansible_net_hostname }} and the OS is {{ ansible_net_
↪version }}"

```

The extended first playbook has four tasks in a single play. Run it with the same command you used above. The output shows you the change Ansible made to the config:

```

$ ansible-playbook -i vyos.example.net, -u ansible -k -e ansible_network_os=vyos_
↪first_playbook_ext.yml

```

```
PLAY [First Playbook]
```

```
*****
```

```
TASK [Gathering Facts]
```

```
*****
```

```
ok: [vyos.example.net]
```

```
TASK [Get config for VyOS devices]
```

```
*****
```

```
ok: [vyos.example.net]
```

```
TASK [Display the config]
```

```
*****
```

```
ok: [vyos.example.net] => {
```

```
  "failed": false,
```

```
  "msg": "The hostname is vyos and the OS is VyOS"
```

```
}
```

```
TASK [Update the hostname]
```

```
*****
```

```
changed: [vyos.example.net]
```

```
TASK [Get changed config for VyOS devices]
```

(continues on next page)

```

*****
ok: [vyos.example.net]

TASK [Display the changed config]
*****
ok: [vyos.example.net] => {
    "failed": false,
    "msg": "The hostname is vyos-changed and the OS is VyOS"
}

PLAY RECAP
*****
vyos.example.net          : ok=6    changed=1    unreachable=0    failed=0

```

This playbook is useful. However, running it still requires several command-line flags. Also, running a playbook against a single device is not a huge efficiency gain over making the same change manually. The next step to harnessing the full power of Ansible is to use an inventory file to organize your managed nodes into groups with information like the `ansible_network_os` and the SSH user.

## Build Your Inventory

A fully-featured inventory file can serve as the source of truth for your network. Using an inventory file, a single playbook can maintain hundreds of network devices with a single command. This page shows you how to build an inventory file, step by step.

### Topics

- *Build Your Inventory*
  - *Basic Inventory*
  - *Add Variables to Inventory*
  - *Group Variables within Inventory*
  - *Variable Syntax*
  - *Group Inventory by Platform*
  - *Protecting Sensitive Variables with `ansible-vault`*

## Basic Inventory

First, group your inventory logically. Best practice is to group servers and network devices by their What (application, stack or microservice), Where (datacenter or region), and When (development stage):

- **What:** db, web, leaf, spine
- **Where:** east, west, floor\_19, building\_A
- **When:** dev, test, staging, prod

Avoid spaces, hyphens, and preceding numbers (use `floor_19`, not `19th_floor`) in your group names. Group names are case sensitive.



This tiny example data center illustrates a basic group structure. You can group groups using the syntax `[metagroupname:children]` and listing groups as members of the metagroup. Here, the group `network` includes all leafs and all spines; the group `datacenter` includes all network devices plus all webserver.

```
[leafs]
leaf01
leaf02

[spines]
spine01
spine02

[network:children]
leafs
spines

[webserver]
webserver01
webserver02

[datacenter:children]
network
webserver
```

### Add Variables to Inventory

Next, you can set values for many of the variables you needed in your first Ansible command in the inventory, so you can skip them in the `ansible-playbook` command. In this example, the inventory includes each network device's IP, OS, and SSH user. If your network devices are only accessible by IP, you must add the IP to the inventory file. If you access your network devices using hostnames, the IP is not necessary.

```
[leafs]
leaf01 ansible_host=10.16.10.11 ansible_network_os=vynos ansible_user=my_vynos_user
leaf02 ansible_host=10.16.10.12 ansible_network_os=vynos ansible_user=my_vynos_user

[spines]
spine01 ansible_host=10.16.10.13 ansible_network_os=vynos ansible_user=my_vynos_user
spine02 ansible_host=10.16.10.14 ansible_network_os=vynos ansible_user=my_vynos_user

[network:children]
leafs
spines

[servers]
server01 ansible_host=10.16.10.15 ansible_user=my_server_user
server02 ansible_host=10.16.10.16 ansible_user=my_server_user

[datacenter:children]
leafs
spines
servers
```

## Group Variables within Inventory

When devices in a group share the same variable values, such as OS or SSH user, you can reduce duplication and simplify maintenance by consolidating these into group variables:

```
[leafs]
leaf01 ansible_host=10.16.10.11
leaf02 ansible_host=10.16.10.12

[leafs:vars]
ansible_network_os=vynos
ansible_user=my_vynos_user

[spines]
spine01 ansible_host=10.16.10.13
spine02 ansible_host=10.16.10.14

[spines:vars]
ansible_network_os=vynos
ansible_user=my_vynos_user

[network:children]
leafs
spines

[servers]
server01 ansible_host=10.16.10.15
server02 ansible_host=10.16.10.16

[datacenter:children]
leafs
spines
servers
```

## Variable Syntax

The syntax for variable values is different in inventory, in playbooks and in `group_vars` files, which are covered below. Even though playbook and `group_vars` files are both written in YAML, you use variables differently in each.

- In an ini-style inventory file you **must** use the syntax `key=value` for variable values:  
`ansible_network_os=vynos.`
- In any file with the `.yaml` or `.yml` extension, including playbooks and `group_vars` files, you **must** use YAML syntax: `key: value`
  - In `group_vars` files, use the full key name: `ansible_network_os: vynos.`
  - In playbooks, use the short-form key name, which drops the `ansible` prefix: `network_os: vynos`

## Group Inventory by Platform

As your inventory grows, you may want to group devices by platform. This allows you to specify platform-specific variables easily for all devices on that platform:

```
[vyos_leafs]
leaf01 ansible_host=10.16.10.11
leaf02 ansible_host=10.16.10.12

[vyos_spines]
spine01 ansible_host=10.16.10.13
spine02 ansible_host=10.16.10.14

[vyos:children]
vyos_leafs
vyos_spines

[vyos:vars]
ansible_connection=network_cli
ansible_network_os=vyos
ansible_user=my_vyos_user

[network:children]
vyos

[servers]
server01 ansible_host=10.16.10.15
server02 ansible_host=10.16.10.16

[datacenter:children]
vyos
servers
```

With this setup, you can run `first_playbook.yml` with only two flags:

```
ansible-playbook -i inventory -k first_playbook.yml
```

With the `-k` flag, you provide the SSH password(s) at the prompt. Alternatively, you can store SSH and other secrets and passwords securely in your `group_vars` files with `ansible-vault`.

## Protecting Sensitive Variables with `ansible-vault`

The `ansible-vault` command provides encryption for files and/or individual variables like passwords. This tutorial will show you how to encrypt a single SSH password. You can use the commands below to encrypt other sensitive information, such as database passwords, privilege-escalation passwords and more.

First you must create a password for `ansible-vault` itself. It is used as the encryption key, and with this you can encrypt dozens of different passwords across your Ansible project. You can access all those secrets (encrypted values) with a single password (the `ansible-vault` password) when you run your playbooks. Here's a simple example.

Create a file and write your password for `ansible-vault` to it:

```
echo "my-ansible-vault-pw" > ~/my-ansible-vault-pw-file
```

Create the encrypted ssh password for your VyOS network devices, pulling your `ansible-vault` password from the file you just created:

```
ansible-vault encrypt_string --vault-id my_user@~/my-ansible-vault-pw-file 'VyOS_SSH_
↪password' --name 'ansible_ssh_pass'
```

If you prefer to type your `ansible-vault` password rather than store it in a file, you can request a prompt:

```
ansible-vault encrypt_string --vault-id my_user@prompt 'VyOS_SSH_password' --name
↳ 'ansible_ssh_pass'
```

and type in the vault password for my\_user.

The `--vault-id` flag allows different vault passwords for different users or different levels of access. The output includes the user name my\_user from your `ansible-vault` command and uses the YAML syntax key: value:

```
ansible_ssh_pass: !vault |
    $ANSIBLE_VAULT;1.2;AES256;my_user
↳ 66386134653765386232383236303063623663343437643766386435663632343266393064373933
↳ 3661666132363339303639353538316662616638356631650a316338316663666439383138353032
↳ 63393934343937373637306162366265383461316334383132626462656463363630613832313562
↳ 3837646266663835640a313164343535316666653031353763613037656362613535633538386539
    65656439626166666363323435613131643066353762333232326232323565376635
Encryption successful
```

Copy this output into your inventory file under `[vyos:vars]`, which now looks like this:

```
[vyos:vars]
ansible_connection=network_cli
ansible_network_os=vyos
ansible_user=my_vyos_user
ansible_ssh_pass= !vault |
    $ANSIBLE_VAULT;1.2;AES256;my_user
↳ 66386134653765386232383236303063623663343437643766386435663632343266393064373933
↳ 3661666132363339303639353538316662616638356631650a316338316663666439383138353032
↳ 63393934343937373637306162366265383461316334383132626462656463363630613832313562
↳ 3837646266663835640a313164343535316666653031353763613037656362613535633538386539
    65656439626166666363323435613131643066353762333232326232323565376635
```

To run a playbook with this setup, drop the `-k` flag and add a flag for your vault-id:

```
ansible-playbook -i inventory --vault-id my_user@~/my-ansible-vault-pw-file first_
↳ playbook.yml
```

Or with a prompt instead of the vault password file:

```
ansible-playbook -i inventory --vault-id my_user@prompt first_playbook.yml
```

To see the original value, you can use the debug module. Please note if your YAML file defines the *ansible\_connection* variable (as we used in our example), it will take effect when you execute the command below. To prevent this, please make a copy of the file without the *ansible\_connection* variable.

```
cat vyos.yml | grep -v ansible_connection >> vyos_no_connection.yml

ansible localhost -m debug -a var="ansible_ssh_pass" -e "@vyos_no_connection.yml" --
↳ ask-vault-pass
```

(continues on next page)

()

```
Vault password:

localhost | SUCCESS => {
  "ansible_ssh_pass": "VyOS_SSH_password"
}
```

: Vault content can only be decrypted with the password that was used to encrypt it. If you want to stop using one password and move to a new one, you can update and re-encrypt existing vault content with `ansible-vault rekey myfile`, then provide the old password and the new password. Copies of vault content still encrypted with the old password can still be decrypted with old password.

For more details on building inventory files, see *the introduction to inventory*; for more details on ansible-vault, see *the full Ansible Vault documentation*.

Now that you understand the basics of commands, playbooks, and inventory, it's time to explore some more complex Ansible Network examples.

## Beyond the Basics

This page introduces some concepts that help you manage your Ansible workflow: roles, directory structure, and source control. Like the Basic Concepts at the beginning of this guide, these intermediate concepts are common to all uses of Ansible. This page also offers resources for learning more.

### Topics

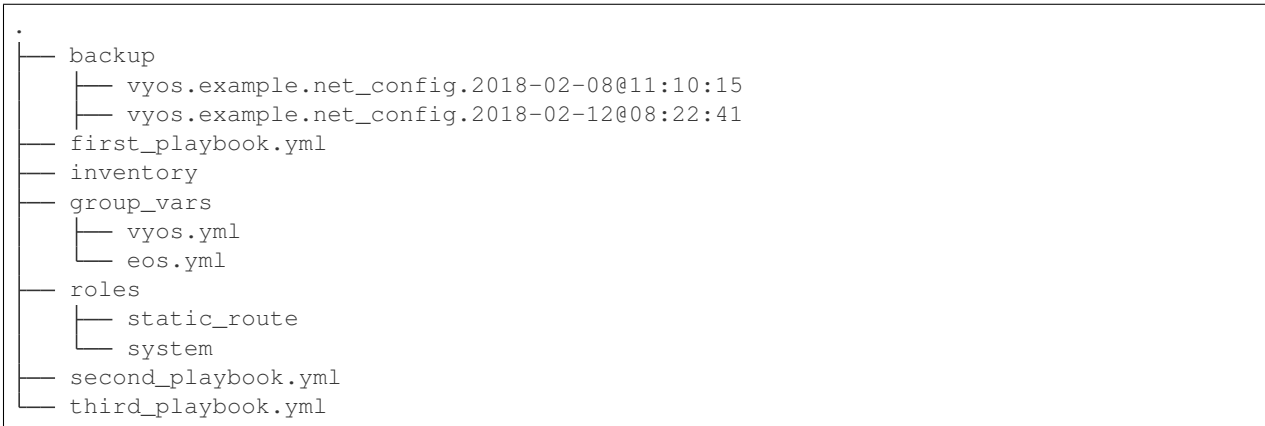
- *Beyond the Basics*
  - *Beyond Playbooks: Moving Tasks and Variables into Roles*
  - *A Typical Ansible Filetree*
  - *Tracking Changes to Inventory and Playbooks: Source Control with Git*
  - *Resources and Next Steps*
    - \* *Text*
    - \* *Events (on Video and in Person)*
    - \* *GitHub Repos*
    - \* *IRC*

## Beyond Playbooks: Moving Tasks and Variables into Roles

Roles are sets of Ansible defaults, files, tasks, templates, variables, and other Ansible components that work together. As you saw on the Working with Playbooks page, moving from a command to a playbook makes it easy to run multiple tasks and repeat the same tasks in the same order. Moving from a playbook to a role makes it even easier to reuse and share your ordered tasks. For more details, see the *documentation on roles*. You can also look at *Ansible Galaxy*, which lets you share your roles and use others' roles, either directly or as inspiration.

## A Typical Ansible Filetree

Ansible expects to find certain files in certain places. As you expand your inventory and create and run more network playbooks, keep your files organized in your working Ansible project directory like this:



The backup directory and the files in it get created when you run modules like `vyos_config` with the `backup: yes` parameter.

## Tracking Changes to Inventory and Playbooks: Source Control with Git

As you expand your inventory, roles and playbooks, you should place your Ansible projects under source control. We recommend `git` for source control. `git` provides an audit trail, letting you track changes, roll back mistakes, view history and share the workload of managing, maintaining and expanding your Ansible ecosystem. There are plenty of tutorials and guides to using `git` available.

## Resources and Next Steps

### Text

Read more about Ansible for Network Automation:

- Network Automation on the [Ansible website](#)
- Ansible Network [Blog posts](#)

### Events (on Video and in Person)

All sessions at Ansible events are recorded and include many Network-related topics (use Filter by Category to view only Network topics). You can also join us for future events in your area. See:

- [Recorded AnsibleFests](#)
- [Recorded AnsibleAutomates](#)
- [Upcoming Ansible Events page](#).

## GitHub Repos

Ansible hosts module code, examples, demonstrations, and other content on GitHub. Anyone with a GitHub account is able to create Pull Requests (PRs) or issues on these repos:

- [Network-Automation](#) is an open community for all things network automation. Have an idea, some playbooks, or roles to share? Email [ansible-network@redhat.com](mailto:ansible-network@redhat.com) and we will add you as a contributor to the repository.
- [Ansible](#) is the main codebase, including code for network modules

## IRC

Join us on Freenode IRC:

- `#ansible-network` Freenode channel

## User Guide to to Network Automation with Ansible

More content coming soon!

## Who should use this guide?

This guide is intended for network engineers using Ansible for automation. It covers advanced topics. If you understand networks and Ansible, this guide is for you. You may read through the entire guide if you choose, or use the links below to find the specific information you need.

If you're new to Ansible, or new to using Ansible for network automation, start with the [Getting Started Guide](#).

## Ansible Network FAQ

### Topics

- [Ansible Network FAQ](#)
  - *How can I improve performance for network playbooks?*
    - \* *Consider strategy: `free` if you are running on multiple hosts*
    - \* *Execute `show` running only if you absolutely must*
    - \* *Use `ProxyCommand` only if you absolutely must*
    - \* *Set `--forks` to match your needs*
  - *Why is my output sometimes replaced with `*****`?*
  - *Why do the `*_config` modules always return `changed=true` with abbreviated commands?*

## How can I improve performance for network playbooks?

### Consider `strategy: free` if you are running on multiple hosts

The `strategy` plugin tells Ansible how to order multiple tasks on multiple hosts. *Strategy* is set at the playbook level.

The default strategy is `linear`. With strategy set to `linear`, Ansible waits until the current task has run on all hosts before starting the next task on any host. Ansible may have forks free, but will not use them until all hosts have completed the current task. If each task in your playbook must succeed on all hosts before you run the next task, use the `linear` strategy.

Using the `free` strategy, Ansible uses available forks to execute tasks on each host as quickly as possible. Even if an earlier task is still running on one host, Ansible executes later tasks on other hosts. The `free` strategy uses available forks more efficiently. If your playbook stalls on each task, waiting for one slow host, consider using `strategy: free` to boost overall performance.

### Execute `show running` only if you absolutely must

The `show running` command is the most resource-intensive command to execute on a network device, because of the way queries are handled by the network OS. Using the command in your Ansible playbook will slow performance significantly, especially on large devices; repeating it will multiply the performance hit. If you have a playbook that checks the running config, then executes changes, then checks the running config again, you should expect that playbook to be very slow.

### Use `ProxyCommand` only if you absolutely must

Network modules support the use of a *proxy or jump host* with the `ProxyCommand` parameter. However, when you use a jump host, Ansible must open a new SSH connection for every task, even if you are using a persistent connection type (`network_cli` or `netconf`). To maximize the performance benefits of the persistent connection types introduced in version 2.5, avoid using jump hosts whenever possible.

### Set `--forks` to match your needs

Every time Ansible runs a task, it forks its own process. The `--forks` parameter defines the number of concurrent tasks - if you retain the default setting, which is `--forks=5`, and you are running a playbook on 10 hosts, five of those hosts will have to wait until a fork is available. Of course, the more forks you allow, the more memory and processing power Ansible will use. Since most network tasks are run on the control host, this means your laptop can quickly become cpu- or memory-bound.

### Why is my output sometimes replaced with `*****`?

Ansible replaces any string marked `no_log`, including passwords, with `*****` in Ansible output. This is done by design, to protect your sensitive data. Most users are happy to have their passwords redacted. However, Ansible replaces every string that matches your password with `*****`. If you use a common word for your password, this can be a problem. For example, if you choose `Admin` as your password, Ansible will replace every instance of the word `Admin` with `*****` in your output. This may make your output harder to read. To avoid this problem, select a secure password that will not occur elsewhere in your Ansible output.



## Why do the `*_config` modules always return `changed=true` with abbreviated commands?

When you issue commands directly on a network device, you can use abbreviated commands. For example, `int g1/0/11` and `interface GigabitEthernet1/0/11` do the same thing; `shut` and `shutdown` do the same thing. Ansible Network `*_command` modules work with abbreviations, because they run commands through the network OS.

When committing configuration, however, the network OS converts abbreviations into long-form commands. Whether you use `shut` or `shutdown` on `GigabitEthernet1/0/11`, the result in the configuration is the same: `shutdown`.

Ansible Network `*_config` modules compare the text of the commands you specify in `lines` to the text in the configuration. If you use `shut` in the `lines` section of your task, and the configuration reads `shutdown`, the module returns `changed=true` even though the configuration is already correct. Your task will update the configuration every time it runs.

To avoid this problem, use long-form commands with the `*_config` modules:

```
---
- hosts: all
  gather_facts: no
  tasks:
    - ios_config:
      lines:
        - shutdown
      parents: interface GigabitEthernet1/0/11
```

## Network Best Practices for Ansible 2.5

### Overview

This document explains the best practices for using Ansible 2.5 to manage your network infrastructure.

### Audience

- This example is intended for network or system administrators who want to understand how to use Ansible to manage network devices.

### Prerequisites

This example requires the following:

- **Ansible 2.5** (or higher) installed. See [Installation Guide](#) for more information.
- One or more network devices that are compatible with Ansible.
- Basic understanding of YAML [YAML Syntax](#).
- Basic understanding of Jinja2 Templates. See [Templating \(Jinja2\)](#) for more information.
- Basic Linux command line use.
- Basic knowledge of network switch & router configurations.

### Concepts

This section explains some fundamental concepts that you should understand when working with Ansible Networking.

### Structure

The examples on this page use the following structure:

```
.
├── facts-demo.yml
└── inventory
```

### Inventory, Connections, Credentials: Grouping Devices and Variables

An `inventory` file is an INI-like configuration file that defines the mapping of hosts into groups.

In our example, the inventory file defines the groups `eos`, `ios`, `vyos` and a "group of groups" called `switches`. Further details about subgroups and inventory files can be found in the [Ansible inventory Group documentation](#).

Because Ansible is a flexible tool, there are a number of ways to specify connection information and credentials. We recommend using the `[my_group:vars]` capability in your inventory file. Here's what it would look like if you specified your ssh passwords (encrypted with Ansible Vault) among your variables:

```
[all:vars]
# these defaults can be overridden for any group in the [group:vars] section
ansible_connection=network_cli
ansible_user=ansible

[switches:children]
eos
ios
vyos

[eos]
veos01 ansible_host=veos-01.example.net
veos02 ansible_host=veos-02.example.net
veos03 ansible_host=veos-03.example.net
veos04 ansible_host=veos-04.example.net

[eos:vars]
ansible_become=yes
ansible_become_method=enable
ansible_network_os=eos
ansible_user=my_eos_user
ansible_ssh_pass= !vault |
                    $ANSIBLE_VAULT;1.1;AES256
↵373737353936366432613830663832353636643866333864323432366635333437303533361653735
↵6131363539383931353931653533356337353539373165320a316465383138636532343463633236
↵37623064393838353962386262643230303438323065356133373930646331623731656163623333
↵3431353332343530650a373038366364316135383063356531633066343434623631303166626532
                    9562
```

(continues on next page)

()

```

[ios]
ios01 ansible_host=ios-01.example.net
ios02 ansible_host=ios-02.example.net
ios03 ansible_host=ios-03.example.net

[ios:vars]
ansible_become=yes
ansible_become_method=enable
ansible_network_os=ios
ansible_user=my_ios_user
ansible_ssh_pass= !vault |
                   $ANSIBLE_VAULT;1.1;AES256
↪ 34623431313336343132373235313066376238386138316466636437653938623965383732373130
↪ 3466363834613161386538393463663861636437653866620a373136356366623765373530633735
↪ 34323262363835346637346261653137626539343534643962376139366330626135393365353739
↪ 3431373064656165320a333834613461613338626161633733343566666630366133623265303563
   8472

[vyos]
vyos01 ansible_host=vyos-01.example.net
vyos02 ansible_host=vyos-02.example.net
vyos03 ansible_host=vyos-03.example.net

[vyos:vars]
ansible_network_os=vyos
ansible_user=my_vyos_user
ansible_ssh_pass= !vault |
                   $ANSIBLE_VAULT;1.1;AES256
↪ 39336231636137663964343966653162353431333566633762393034646462353062633264303765
↪ 6331643066663534383564343537343334633031656538370a333737656236393835383863306466
↪ 62633364653238323333633337313163616566383836643030336631333431623631396364663533
↪ 3665626431626532630a353564323566316162613432373738333064366130303637616239396438
   9853

```

If you use `ssh-agent`, you do not need the `ansible_ssh_pass` lines. If you use `ssh` keys, but not `ssh-agent`, and you have multiple keys, specify the key to use for each connection in the `[group:vars]` section with `ansible_ssh_private_key_file=/path/to/correct/key`. For more information on `ansible_ssh_` options see the [List of Behavioral Inventory Parameters](#).

: Never store passwords in plain text.

The "Vault" feature of Ansible allows you to keep sensitive data such as passwords or keys in encrypted files, rather than as plain text in your playbooks or roles. These vault files can then be distributed or placed in source control. See [Using Vault in playbooks](#) for more information.

**ansible\_connection** Ansible uses the `ansible_connection` setting to determine how to connect to a remote

device. When working with Ansible Networking, set this to `network_cli` so Ansible treats the remote node as a network device with a limited execution environment. Without this setting, Ansible would attempt to use `ssh` to connect to the remote and execute the Python script on the network device, which would fail because Python generally isn't available on network devices.

**ansible\_network\_os** Informs Ansible which Network platform this hosts corresponds to. This is required when using `network_cli` or `netconf`.

**ansible\_user** The user to connect to the remote device (switch) as. Without this the user that is running `ansible-playbook` would be used. Specifies which user on the network device the connection

**ansible\_ssh\_pass** The corresponding password for `ansible_user` to log in as. If not specified SSH key will be used.

**ansible\_become** If enable mode (privilege mode) should be used, see the next section.

**ansible\_become\_method** Which type of *become* should be used, for `network_cli` the only valid choice is `enable`.

## Privilege escalation

Certain network platforms, such as `eos` and `ios`, have the concept of different privilege modes. Certain network modules, such as those that modify system state including users, will only work in high privilege states. Ansible version 2.5 added support for `become` when using `connection: network_cli`. This allows privileges to be raised for the specific tasks that need them. Adding `become: yes` and `become_method: enable` informs Ansible to go into privilege mode before executing the task, as shown here:

```
[eos:vars]
ansible_connection=network_cli
ansible_network_os=eos
ansible_become=yes
ansible_become_method=enable
```

For more information, see the [using become with network modules](#) guide.

## Jump hosts

If the Ansible Controller doesn't have a direct route to the remote device and you need to use a Jump Host, please see the [Ansible Network Proxy Command](#) guide for details on how to achieve this.

## Playbook

### Collect data

Ansible facts modules gather system information 'facts' that are available to the rest of your playbook.

Ansible Networking ships with a number of network-specific facts modules. In this example, we use the `_facts` modules `eos_facts`, `ios_facts` and `vyos_facts` to connect to the remote networking device. As the credentials are not explicitly passed via module arguments, Ansible uses the username and password from the inventory file.

Ansible's "Network Fact modules" gather information from the system and store the results in facts prefixed with `ansible_net_`. The data collected by these modules is documented in the *Return Values* section of the module docs, in this case `eos_facts` and `vyos_facts`. We can use the facts, such as `ansible_net_version` late on in the "Display some facts" task.

To ensure we call the correct mode (`*_facts`) the task is conditionally run based on the group defined in the inventory file, for more information on the use of conditionals in Ansible Playbooks see *The When Statement*.

## Example

In this example, we will create an inventory file containing some network switches, then run a playbook to connect to the network devices and return some information about them.

### Create an inventory file

First, create a file called `inventory`, containing:

```
[switches:children]
eos
ios
vyos

[eos]
eos01.example.net

[ios]
ios01.example.net

[vyos]
vyos01.example.net
```

### Create a playbook

Next, create a playbook file called `facts-demo.yml` containing the following:

```
- name: "Demonstrate connecting to switches"
  hosts: switches
  gather_facts: no

  tasks:
    ###
    # Collect data
    #
    - name: Gather facts (eos)
      eos_facts:
      when: ansible_network_os == 'eos'

    - name: Gather facts (ios)
      ios_facts:
      when: ansible_network_os == 'ios'

    - name: Gather facts (vyos)
      vyos_facts:
      when: ansible_network_os == 'vyos'

    ###
    # Demonstrate variables
    #
    - name: Display some facts
      debug:
        msg: "The hostname is {{ ansible_net_hostname }} and the OS is {{ ansible_net_
↵version }}"
```

(continues on next page)

```

- name: Facts from a specific host
  debug:
    var: hostvars['vyos01.example.net']

- name: Write facts to disk using a template
  copy:
    content: |
      #jinja2: lstrip_blocks: True
      EOS device info:
      {% for host in groups['eos'] %}
      Hostname: {{ hostvars[host].ansible_net_hostname }}
      Version: {{ hostvars[host].ansible_net_version }}
      Model: {{ hostvars[host].ansible_net_model }}
      Serial: {{ hostvars[host].ansible_net_serialnum }}
      {% endfor %}

      IOS device info:
      {% for host in groups['ios'] %}
      Hostname: {{ hostvars[host].ansible_net_hostname }}
      Version: {{ hostvars[host].ansible_net_version }}
      Model: {{ hostvars[host].ansible_net_model }}
      Serial: {{ hostvars[host].ansible_net_serialnum }}
      {% endfor %}

      VyOS device info:
      {% for host in groups['vyos'] %}
      Hostname: {{ hostvars[host].ansible_net_hostname }}
      Version: {{ hostvars[host].ansible_net_version }}
      Model: {{ hostvars[host].ansible_net_model }}
      Serial: {{ hostvars[host].ansible_net_serialnum }}
      {% endfor %}
    dest: /tmp/switch-facts
    run_once: yes

###
# Get running configuration
#

- name: Backup switch (eos)
  eos_config:
    backup: yes
    register: backup_eos_location
    when: ansible_network_os == 'eos'

- name: backup switch (vyos)
  vyos_config:
    backup: yes
    register: backup_vyos_location
    when: ansible_network_os == 'vyos'

- name: Create backup dir
  file:
    path: "/tmp/backups/{{ inventory_hostname }}"
    state: directory
    recurse: yes

- name: Copy backup files into /tmp/backups/ (eos)

```

(continues on next page)

()

```

copy:
  src: "{{ backup_eos_location.backup_path }}"
  dest: "/tmp/backups/{{ inventory_hostname }}/{{ inventory_hostname }}.bck"
  when: ansible_network_os == 'eos'

- name: Copy backup files into /tmp/backups/ (vyos)
  copy:
    src: "{{ backup_vyos_location.backup_path }}"
    dest: "/tmp/backups/{{ inventory_hostname }}/{{ inventory_hostname }}.bck"
    when: ansible_network_os == 'vyos'

```

## Running the playbook

To run the playbook, run the following from a console prompt:

```
ansible-playbook -i inventory facts-demo.yml
```

This should return output similar to the following:

```

PLAY RECAP
eos01.example.net      : ok=7    changed=2    unreachable=0    failed=0
ios01.example.net      : ok=7    changed=2    unreachable=0    failed=0
vyos01.example.net     : ok=6    changed=2    unreachable=0    failed=0

```

Next, look at the contents of the file we created containing the switch facts:

```
cat /tmp/switch-facts
```

You can also look at the backup files:

```
find /tmp/backups
```

If *ansible-playbook* fails, please follow the debug steps in *Network Debug and Troubleshooting Guide*.

## Implementation Notes

### Demo variables

Although these tasks are not needed to write data to disk, they are used in this example to demonstrate some methods of accessing facts about the given devices or a named host.

Ansible `hostvars` allows you to access variables from a named host. Without this we would return the details for the current host, rather than the named host.

For more information, see *Magic Variables, and How To Access Information About Other Hosts*.

### Get running configuration

The `eos_config` and `vyos_config` modules have a `backup:` option that when set will cause the module to create a full backup of the current `running-config` from the remote device before any changes are made. The backup file is written to the `backup` folder in the playbook root directory. If the directory does not exist, it is created.

To demonstrate how we can move the backup file to a different location, we register the result and move the file to the path stored in `backup_path`.

Note that when using variables from tasks in this way we use double quotes (") and double curly-brackets ({{ ... }}) to tell Ansible that this is a variable.

### Troubleshooting

If you receive an connection error please double check the inventory and Playbook for typos or missing lines. If the issue still occurs follow the debug steps in [Network Debug and Troubleshooting Guide](#).

:

- [Ansible for Network Automation](#)
- [Working with Inventory](#)
- [Vault best practices](#)

### Network Debug and Troubleshooting Guide

#### Introduction

Starting with Ansible version 2.1, you can now use the familiar Ansible models of playbook authoring and module development to manage heterogeneous networking devices. Ansible supports a growing number of network devices using both CLI over SSH and API (when available) transports.

This section discusses how to debug and troubleshoot network modules in Ansible 2.3.

#### How to troubleshoot

This section covers troubleshooting issues with Network Modules.

Errors generally fall into one of the following categories:

##### Authentication issues

- Not correctly specifying credentials
- Remote device (network switch/router) not falling back to other authentication methods
- SSH key issues

##### Timeout issues

- Can occur when trying to pull a large amount of data
- May actually be masking a authentication issue

##### Playbook issues

- Use of `delegate_to`, instead of `ProxyCommand`. See [network proxy guide](#) for more information.
- Not using `connection: local`



```
: unable to open shell
```

The unable to open shell message is new in Ansible 2.3, it means that the `ansible-connection` daemon has not been able to successfully talk to the remote network device. This generally means that there is an authentication issue. See the "Authentication and connection issues" section in this document for more information.

## Enabling Networking logging and how to read the logfile

**Platforms:** Any

Ansible 2.3 features improved logging to help diagnose and troubleshoot issues regarding Ansible Networking modules.

Because logging is very verbose it is disabled by default. It can be enabled via the `ANSIBLE_LOG_PATH` and `ANSIBLE_DEBUG` options on the ansible-controller, that is the machine running ansible-playbook.

Before running `ansible-playbook` run the following commands to enable logging:

```
# Specify the location for the log file
export ANSIBLE_LOG_PATH=~/.ansible.log
# Enable Debug
export ANSIBLE_DEBUG=True

# Run with 4*v for connection level verbosity
ansible-playbook -vvvv ...
```

After Ansible has finished running you can inspect the log file which has been created on the ansible-controller:

```
less $ANSIBLE_LOG_PATH

2017-03-30 13:19:52,740 p=28990 u=fred | creating new control socket for host_
↪veos01:22 as user admin
2017-03-30 13:19:52,741 p=28990 u=fred | control socket path is /home/fred/.ansible/
↪pc/ca5960d27a
2017-03-30 13:19:52,741 p=28990 u=fred | current working directory is /home/fred/
↪ansible/test/integration
2017-03-30 13:19:52,741 p=28990 u=fred | using connection plugin network_cli
...
2017-03-30 13:20:14,771 paramiko.transport userauth is OK
2017-03-30 13:20:15,283 paramiko.transport Authentication (keyboard-interactive)_
↪successful!
2017-03-30 13:20:15,302 p=28990 u=fred | ssh connection done, setting terminal
2017-03-30 13:20:15,321 p=28990 u=fred | ssh connection has completed successfully
2017-03-30 13:20:15,322 p=28990 u=fred | connection established to veos01 in 0:00:22.
↪580626
```

From the log notice:

- `p=28990` Is the PID (Process ID) of the `ansible-connection` process
- `u=fred` Is the user *running* ansible, not the remote-user you are attempting to connect as
- `creating new control socket for host veos01:22 as user admin` host:port as user
- `control socket path is` location on disk where the persistent connection socket is created
- `using connection plugin network_cli` Informs you that persistent connection is being used

- connection established to veos01 in 0:00:22.580626 Time taken to obtain a shell on the remote device

Because the log files are verbose, you can use `grep` to look for specific information. For example, once you have identified the `pid` from the creating new control socket for host line you can search for other connection log entries:

```
grep "p=28990" $ANSIBLE_LOG_PATH
```

### Isolating an error

**Platforms:** Any

As with any effort to troubleshoot it's important to simplify the test case as much as possible.

For Ansible this can be done by ensuring you are only running against one remote device:

- Using `ansible-playbook --limit switch1.example.net...`
- Using an ad-hoc `ansible` command

*ad-hoc* refers to running Ansible to perform some quick command using `/usr/bin/ansible`, rather than the orchestration language, which is `/usr/bin/ansible-playbook`. In this case we can ensure connectivity by attempting to execute a single command on the remote device:

```
ansible -m eos_command -a 'commands=?' -i inventory switch1.example.net -e 'ansible_
↪connection=local' -u admin -k
```

In the above example, we:

- connect to `switch1.example.net` specified in the inventory file `inventory`
- use the module `eos_command`
- run the command `?`
- connect using the username `admin`
- inform ansible to prompt for the ssh password by specifying `-k`

If you have SSH keys configured correctly, you don't need to specify the `-k` parameter

If the connection still fails you can combine it with the `enable_network_logging` parameter. For example:

```
# Specify the location for the log file
export ANSIBLE_LOG_PATH=~/.ansible.log
# Enable Debug
export ANSIBLE_DEBUG=True
# Run with 4*v for connection level verbosity
ansible -m eos_command -a 'commands=?' -i inventory switch1.example.net -e 'ansible_
↪connection=local' -u admin -k
```

Then review the log file and find the relevant error message in the rest of this document.

### Category "socket\_path issue"

**Platforms:** Any

The `socket_path` does not exist or cannot be found and unable to connect to socket messages are new in Ansible 2.5. These messages indicate that the socket used to communicate with the remote network device is unavailable or does not exist.

For example:

```
fatal: [spine02]: FAILED! => {
  "changed": false,
  "failed": true,
  "module_stderr": "Traceback (most recent call last):\n  File \"/tmp/ansible_
↪TSqk5J/ansible_modlib.zip/ansible/module_utils/connection.py", line 115, in _exec_
↪jsonrpc\nansible.module_utils.connection.ConnectionError: socket_path does not_
↪exist or cannot be found\n",
  "module_stdout": "",
  "msg": "MODULE FAILURE",
  "rc": 1
}
```

or

```
fatal: [spine02]: FAILED! => {
  "changed": false,
  "failed": true,
  "module_stderr": "Traceback (most recent call last):\n  File \"/tmp/ansible_
↪TSqk5J/ansible_modlib.zip/ansible/module_utils/connection.py", line 123, in _exec_
↪jsonrpc\nansible.module_utils.connection.ConnectionError: unable to connect to_
↪socket\n",
  "module_stdout": "",
  "msg": "MODULE FAILURE",
  "rc": 1
}
```

Suggestions to resolve:

Follow the steps detailed in [enable network logging](#).

If the identified error message from the log file is:

```
2017-04-04 12:19:05,670 p=18591 u=fred | command timeout triggered, timeout value is_
↪10 secs
```

or

```
2017-04-04 12:19:05,670 p=18591 u=fred | persistent connection idle timeout_
↪triggered, timeout value is 30 secs
```

Follow the steps detailed in [timeout issues](#)

## Category "Unable to open shell"

**Platforms:** Any

The `unable to open shell` message is new in Ansible 2.3. This message means that the `ansible-connection` daemon has not been able to successfully talk to the remote network device. This generally means that there is an authentication issue. It is a "catch all" message, meaning you need to enable `:ref:logging'a_note_about_logging'` to find the underlying issues.

For example:

```
TASK [prepare_eos_tests : enable cli on remote device]_
↪*****
fatal: [veos01]: FAILED! => {"changed": false, "failed": true, "msg": "unable to open_
↪shell"}
```

or:

```
TASK [ios_system : configure name_servers]_
↪*****
task path:
fatal: [ios-csr1000v]: FAILED! => {
    "changed": false,
    "failed": true,
    "msg": "unable to open shell",
}
```

Suggestions to resolve:

Follow the steps detailed in [enable\\_network\\_logging](#).

Once you've identified the error message from the log file, the specific solution can be found in the rest of this document.

### **Error: "[Errno -2] Name or service not known"**

**Platforms:** Any

Indicates that the remote host you are trying to connect to can not be reached

For example:

```
2017-04-04 11:39:48,147 p=15299 u=fred | control socket path is /home/fred/.ansible/
↪pc/ca5960d27a
2017-04-04 11:39:48,147 p=15299 u=fred | current working directory is /home/fred/git/
↪ansible-inc/stable-2.3/test/integration
2017-04-04 11:39:48,147 p=15299 u=fred | using connection plugin network_cli
2017-04-04 11:39:48,340 p=15299 u=fred | connecting to host veos01 returned an error
2017-04-04 11:39:48,340 p=15299 u=fred | [Errno -2] Name or service not known
```

Suggestions to resolve:

- If you are using the `provider`: options ensure that it's suboption `host`: is set correctly.
- If you are not using `provider`: nor top-level arguments ensure your inventory file is correct.

### **Error: "Authentication failed"**

**Platforms:** Any

Occurs if the credentials (username, passwords, or ssh keys) passed to `ansible-connection` (via `ansible` or `ansible-playbook`) can not be used to connect to the remote device.

For example:

```
<ios01> ESTABLISH CONNECTION FOR USER: cisco on PORT 22 TO ios01
<ios01> Authentication failed.
```

Suggestions to resolve:

If you are specifying credentials via `password:` (either directly or via `provider:`) or the environment variable `ANSIBLE_NET_PASSWORD` it is possible that `paramiko` (the Python SSH library that Ansible uses) is using `ssh` keys, and therefore the credentials you are specifying are being ignored. To find out if this is the case, disable "look for keys". This can be done like this:

```
export ANSIBLE_PARAMIKO_LOOK_FOR_KEYS=False
```

To make this a permanent change, add the following to your `ansible.cfg` file:

```
[paramiko_connection]
look_for_keys = False
```

### Error: "connecting to host <hostname> returned an error" or "Bad address"

This may occur if the SSH fingerprint hasn't been added to Paramiko's (the Python SSH library) known hosts file.

When using persistent connections with Paramiko, the connection runs in a background process. If the host doesn't already have a valid SSH key, by default Ansible will prompt to add the host key. This will cause connections running in background processes to fail.

For example:

```
2017-04-04 12:06:03,486 p=17981 u=fred | using connection plugin network_cli
2017-04-04 12:06:04,680 p=17981 u=fred | connecting to host veos01 returned an error
2017-04-04 12:06:04,682 p=17981 u=fred | (14, 'Bad address')
2017-04-04 12:06:33,519 p=17981 u=fred | number of connection attempts exceeded,
↪unable to connect to control socket
2017-04-04 12:06:33,520 p=17981 u=fred | persistent_connect_interval=1, persistent_
↪connect_retries=30
```

Suggestions to resolve:

Use `ssh-keyscan` to pre-populate the `known_hosts`. You need to ensure the keys are correct.

```
ssh-keyscan veos01
```

or

You can tell Ansible to automatically accept the keys

Environment variable method:

```
export ANSIBLE_PARAMIKO_HOST_KEY_AUTO_ADD=True
ansible-playbook ...
```

`ansible.cfg` method:

`ansible.cfg`

```
[paramiko_connection]
host_key_auto_add = True
```

### Error: "No authentication methods available"

For example:

```
2017-04-04 12:19:05,670 p=18591 u=fred | creating new control socket for host_
↳veos01:None as user admin
2017-04-04 12:19:05,670 p=18591 u=fred | control socket path is /home/fred/.ansible/
↳pc/ca5960d27a
2017-04-04 12:19:05,670 p=18591 u=fred | current working directory is /home/fred/git/
↳ansible-inc/ansible-workspace-2/test/integration
2017-04-04 12:19:05,670 p=18591 u=fred | using connection plugin network_cli
2017-04-04 12:19:06,606 p=18591 u=fred | connecting to host veos01 returned an error
2017-04-04 12:19:06,606 p=18591 u=fred | No authentication methods available
2017-04-04 12:19:35,708 p=18591 u=fred | connect retry timeout expired, unable to_
↳connect to control socket
2017-04-04 12:19:35,709 p=18591 u=fred | persistent_connect_retry_timeout is 15 secs
```

Suggestions to resolve:

No password or SSH key supplied

## Clearing Out Persistent Connections

**Platforms:** Any

In Ansible 2.3, persistent connection sockets are stored in `~/.ansible/pc` for all network devices. When an Ansible playbook runs, the persistent socket connection is displayed when verbose output is specified.

```
<switch> socket_path: /home/fred/.ansible/pc/f64ddfa760
```

To clear out a persistent connection before it times out (the default timeout is 30 seconds of inactivity), simple delete the socket file.

## Timeout issues

### Timeouts

Persistent connection idle timeout:

For example:

```
2017-04-04 12:19:05,670 p=18591 u=fred | persistent connection idle timeout_
↳triggered, timeout value is 30 secs
```

Suggestions to resolve:

Increase value of persistent connection idle timeout:

```
export ANSIBLE_PERSISTENT_CONNECT_TIMEOUT=60
```

To make this a permanent change, add the following to your `ansible.cfg` file:

```
[persistent_connection]
connect_timeout = 60
```

Command timeout: For example:

```
2017-04-04 12:19:05,670 p=18591 u=fred | command timeout triggered, timeout value is_
↳10 secs
```

Suggestions to resolve:

Options 1: Increase value of command timeout in configuration file or by setting environment variable. Note: This value should be less than persistent connection idle timeout ie. `connect_timeout`

```
export ANSIBLE_PERSISTENT_COMMAND_TIMEOUT=30
```

To make this a permanent change, add the following to your `ansible.cfg` file:

```
[persistent_connection]
command_timeout = 30
```

Option 2: Increase command timeout per task basis. All network modules support a timeout value that can be set on a per task basis. The timeout value controls the amount of time in seconds before the task will fail if the command has not returned.

For example:

Suggestions to resolve:

```
- name: save running-config
  ios_command:
    commands: copy running-config startup-config
    provider: "{{ cli }}"
    timeout: 30
```

Some operations take longer than the default 10 seconds to complete. One good example is saving the current running config on IOS devices to startup config. In this case, changing the timeout value from the default 10 seconds to 30 seconds will prevent the task from failing before the command completes successfully. Note: This value should be less than persistent connection idle timeout ie. `connect_timeout`

Persistent socket connect timeout: For example:

```
2017-04-04 12:19:35,708 p=18591 u=fred | connect retry timeout expired, unable to_
↪connect to control socket
2017-04-04 12:19:35,709 p=18591 u=fred | persistent_connect_retry_timeout is 15 secs
```

Suggestions to resolve:

Increase the value of the persistent connection idle timeout. Note: This value should be greater than the SSH timeout value (the timeout value under the defaults section in the configuration file) and less than the value of the persistent connection idle timeout (`connect_timeout`).

```
export ANSIBLE_PERSISTENT_CONNECT_RETRY_TIMEOUT=30
```

To make this a permanent change, add the following to your `ansible.cfg` file:

```
[persistent_connection]
connect_retry_timeout = 30
```

## Playbook issues

This section details issues are caused by issues with the Playbook itself.

**Error: "invalid connection specified, expected connection=local, got ssh"**

**Platforms:** Any

Network modules require that the connection is set to `local`. Any other connection setting will cause the playbook to fail. Ansible will now detect this condition and return an error message:

```
fatal: [nxos01]: FAILED! => {
  "changed": false,
  "failed": true,
  "msg": "invalid connection specified, expected connection=local, got ssh"
}
```

To fix this issue, set the connection value to `local` using one of the following methods:

- Set the play to use `connection: local`
- Set the task to use `connection: local`
- Run `ansible-playbook` using the `-c local` setting

### Error: "Unable to enter configuration mode"

**Platforms:** eos and ios

This occurs when you attempt to run a task that requires privileged mode in a user mode shell.

For example:

```
TASK [ios_system : configure name_servers]_
↪*****
task path:
fatal: [ios-csr1000v]: FAILED! => {
  "changed": false,
  "failed": true,
  "msg": "unable to enter configuration mode",
}
```

Suggestions to resolve:

Add `authorize: yes` to the task. For example:

```
- name: configure hostname
  ios_system:
    provider:
      hostname: foo
      authorize: yes
  register: result
```

If the user requires a password to go into privileged mode, this can be specified with `auth_pass`; if `auth_pass` isn't set, the environment variable `ANSIBLE_NET_AUTHORIZE` will be used instead.

Add `authorize: yes` to the task. For example:

```
- name: configure hostname
  ios_system:
    provider:
      hostname: foo
      authorize: yes
      auth_pass: "{{ mypasswordvar }}"
  register: result
```



## Proxy Issues

### delegate\_to vs ProxyCommand

The new connection framework for Network Modules in Ansible 2.3 that uses `cli` transport no longer supports the use of the `delegate_to` directive. In order to use a bastion or intermediate jump host to connect to network devices over `cli` transport, network modules now support the use of `ProxyCommand`.

To use `ProxyCommand`, configure the proxy settings in the Ansible inventory file to specify the proxy host.

```
[nxos]
nxos01
nxos02

[nxos:vars]
ansible_ssh_common_args='-o ProxyCommand="ssh -W %h:%p -q bastion01"'
```

With the configuration above, simply build and run the playbook as normal with no additional changes necessary. The network module will now connect to the network device by first connecting to the host specified in `ansible_ssh_common_args`, which is `bastion01` in the above example.

---

#### : Using `ProxyCommand` with passwords via variables

By design, SSH doesn't support providing passwords via environment variables. This is done to prevent secrets from leaking out, for example in `ps` output.

We recommend using SSH Keys, and if needed an `ssh-agent`, rather than passwords, where ever possible.

---

## Working with Command Output in Network Modules

### Conditionals in Networking Modules

Ansible allows you to use conditionals to control the flow of your playbooks. Ansible networking command modules use the following unique conditional statements.

- `eq` - Equal
- `neq` - Not equal
- `gt` - Greater than
- `ge` - Greater than or equal
- `lt` - Less than
- `le` - Less than or equal
- `contains` - Object contains specified item

Conditional statements evaluate the results from the commands that are executed remotely on the device. Once the task executes the command set, the `wait_for` argument can be used to evaluate the results before returning control to the Ansible playbook.

For example:

```
---
- name: wait for interface to be admin enabled
  eos_command:
    commands:
      - show interface Ethernet4 | json
    wait_for:
      - "result[0].interfaces.Ethernet4.interfaceStatus eq connected"
```

In the above example task, the command `show interface Ethernet4 | json` is executed on the remote device and the results are evaluated. If the path `(result[0].interfaces.Ethernet4.interfaceStatus)` is not equal to "connected", then the command is retried. This process continues until either the condition is satisfied or the number of retries has expired (by default, this is 10 retries at 1 second intervals).

The commands module can also evaluate more than one set of command results in an interface. For instance:

```
---
- name: wait for interfaces to be admin enabled
  eos_command:
    commands:
      - show interface Ethernet4 | json
      - show interface Ethernet5 | json
    wait_for:
      - "result[0].interfaces.Ethernet4.interfaceStatus eq connected"
      - "result[1].interfaces.Ethernet4.interfaceStatus eq connected"
```

In the above example, two commands are executed on the remote device, and the results are evaluated. By specifying the result index value (0 or 1), the correct result output is checked against the conditional.

The `wait_for` argument must always start with `result` and then the command index in `[]`, where 0 is the first command in the commands list, 1 is the second command, 2 is the third and so on.

## Platform Options

Some Ansible Network platforms support multiple connection types, privilege escalation (`enable` mode), or other options. The pages in this section offer standardized guides to understanding available options on each network platform. We welcome contributions from community-maintained platforms to this section.

## EOS Platform Options

Arista EOS supports multiple connections. This page offers details on how each connection works in Ansible 2.6 and how to use it.

### Topics

- *EOS Platform Options*
  - *Connections Available*
  - *Using CLI in Ansible 2.6*
    - \* *Example CLI group\_vars/eos.yml*
    - \* *Example CLI Task*
  - *Using eAPI in Ansible 2.6*

- \* *Enabling eAPI*
- \* *Example eAPI group\_vars/eos.yml*
- \* *Example eAPI Task*
- \* *eAPI examples with connection: local*

## Connections Available

	CLI	eAPI
Protocol	SSH	HTTP(S)
Credentials	uses SSH keys / SSH-agent if present accepts <code>-u myuser -k</code> if using password	uses HTTPS certificates if present
Indirect Access	via a bastion (jump host)	via a web proxy
Connection Settings	<code>ansible_connection:</code> <code>network_cli</code>	<code>ansible_connection:</code> <code>httpapi</code> OR <code>ansible_connection:</code> <code>local</code> with <code>transport: eapi</code> in the provider dictionary
Enable Mode (Privilege Escalation)	supported - use <code>ansible_become: yes</code> with <code>ansible_become_method:</code> <code>enable</code>	supported: <code>httpapi</code> uses <code>ansible_become: yes</code> with <code>ansible_become_method:</code> <code>enable</code> <code>local</code> uses <code>authorize: yes</code> and <code>auth_pass:</code> in the provider dictionary
Returned Data Format	<code>stdout[0].</code>	<code>stdout[0].messages[0].</code>

For legacy playbooks, EOS still supports `ansible_connection: local`. We recommend modernizing to use `ansible_connection: network_cli` or `ansible_connection: httpapi` as soon as possible.

## Using CLI in Ansible 2.6

### Example CLI `group_vars/eos.yml`

```
ansible_connection: network_cli
ansible_network_os: eos
ansible_user: myuser
ansible_ssh_pass: !vault...
ansible_become: yes
ansible_become_method: enable
ansible_become_pass: !vault...
ansible_ssh_common_args: '-o ProxyCommand="ssh -W %h:%p -q bastion01"'
```

- If you are using SSH keys (including an ssh-agent) you can remove the `ansible_ssh_pass` configuration.
- If you are accessing your host directly (not through a bastion/jump host) you can remove the `ansible_ssh_common_args` configuration.
- If you are accessing your host through a bastion/jump host, you cannot include your SSH password in the `ProxyCommand` directive. To prevent secrets from leaking out (for example in `ps` output), SSH does not support providing passwords via environment variables.

### Example CLI Task

```
- name: Backup current switch config (eos)
  eos_config:
    backup: yes
  register: backup_eos_location
  when: ansible_network_os == 'eos'
```

## Using eAPI in Ansible 2.6

### Enabling eAPI

Before you can use eAPI to connect to a switch, you must enable eAPI. To enable eAPI on a new switch via Ansible, use the `eos_eapi` module via the CLI connection. Set up `group_vars/eos.yml` just like in the CLI example above, then run a playbook task like this:

```
- name: Enable eAPI
  eos_eapi:
    enable_http: yes
    enable_https: yes
  become: true
  become_method: enable
  when: ansible_network_os == 'eos'
```

You can find more options for enabling HTTP/HTTPS connections in the `eos_eapi` module documentation.

Once eAPI is enabled, change your `group_vars/eos.yml` to use the eAPI connection.

**Example eAPI group\_vars/eos.yml**

```

ansible_connection: httpapi
ansible_network_os: eos
ansible_user: myuser
ansible_ssh_pass: !vault...
become: yes
become_method: enable
proxy_env:
  http_proxy: http://proxy.example.com:8080

```

- If you are accessing your host directly (not through a web proxy) you can remove the `proxy_env` configuration.
- If you are accessing your host through a web proxy using https, change `http_proxy` to `https_proxy`.

**Example eAPI Task**

```

- name: Backup current switch config (eos)
  eos_config:
    backup: yes
    register: backup_eos_location
    environment: "{{ proxy_env }}"
    when: ansible_network_os == 'eos'

```

In this example the `proxy_env` variable defined in `group_vars` gets passed to the `environment` option of the module in the task.

**eAPI examples with connection: local**

group\_vars/eos.yml:

```

ansible_connection: local
ansible_network_os: eos
ansible_user: myuser
ansible_ssh_pass: !vault...
eapi:
  host: "{{ inventory_hostname }}"
  transport: eapi
  authorize: yes
  auth_pass: !vault...
proxy_env:
  http_proxy: http://proxy.example.com:8080

```

eAPI task:

```

- name: Backup current switch config (eos)
  eos_config:
    backup: yes
    provider: "{{ eapi }}"
    register: backup_eos_location
    environment: "{{ proxy_env }}"
    when: ansible_network_os == 'eos'

```

In this example two variables defined in `group_vars` get passed to the module of the task:

- the `eapi` variable gets passed to the `provider` option of the module
- the `proxy_env` variable gets passed to the `environment` option of the module

: Never store passwords in plain text. We recommend using SSH keys to authenticate SSH connections. Ansible supports `ssh-agent` to manage your SSH keys. If you must use passwords to authenticate SSH connections, we recommend encrypting them with *Ansible Vault*.

### IOS Platform Options

IOS supports Enable Mode (Privilege Escalation). This page offers details on how to use Enable Mode on IOS in Ansible 2.6.

#### Topics

- *IOS Platform Options*
  - *Connections Available*
  - *Using CLI in Ansible 2.6*
    - \* *Example CLI group\_vars/ios.yml*
    - \* *Example CLI Task*

### Connections Available

	CLI
<b>Protocol</b>	SSH
<b>Credentials</b>	uses SSH keys / SSH-agent if present accepts <code>-u myuser -k</code> if using password
<b>Indirect Access</b>	via a bastion (jump host)
<b>Connection Settings</b>	<code>ansible_connection: network_cli</code>
<b>Enable Mode</b> (Privilege Escalation)	supported - use <code>ansible_become: yes</code> with <code>ansible_become_method: enable</code> and <code>ansible_become_pass:</code>
<b>Returned Data Format</b>	<code>stdout[0]</code> .

For legacy playbooks, IOS still supports `ansible_connection: local`. We recommend modernizing to use `ansible_connection: network_cli` as soon as possible.

## Using CLI in Ansible 2.6

### Example CLI `group_vars/ios.yml`

```
ansible_connection: network_cli
ansible_network_os: ios
ansible_user: myuser
ansible_ssh_pass: !vault...
ansible_become: yes
ansible_become_method: enable
ansible_become_pass: !vault...
ansible_ssh_common_args: '-o ProxyCommand="ssh -W %h:%p -q bastion01"'
```

- If you are using SSH keys (including an ssh-agent) you can remove the `ansible_ssh_pass` configuration.
- If you are accessing your host directly (not through a bastion/jump host) you can remove the `ansible_ssh_common_args` configuration.
- If you are accessing your host through a bastion/jump host, you cannot include your SSH password in the `ProxyCommand` directive. To prevent secrets from leaking out (for example in `ps` output), SSH does not support providing passwords via environment variables.

### Example CLI Task

```
- name: Backup current switch config (ios)
  ios_config:
    backup: yes
  register: backup_ios_location
  when: ansible_network_os == 'ios'
```

: Never store passwords in plain text. We recommend using SSH keys to authenticate SSH connections. Ansible supports ssh-agent to manage your SSH keys. If you must use passwords to authenticate SSH connections, we recommend encrypting them with *Ansible Vault*.

## Junos OS Platform Options

Juniper Junos OS supports multiple connections. This page offers details on how each connection works in Ansible 2.6 and how to use it.

### Topics

- *Junos OS Platform Options*
  - *Connections Available*
  - *Using CLI in Ansible 2.6*
    - \* *Example CLI inventory [junos:vars]*
    - \* *Example CLI Task*
  - *Using NETCONF in Ansible 2.6*

- \* *Enabling NETCONF*
- \* *Example NETCONF inventory [junos:vars]*
- \* *Example NETCONF Task*

## Connections Available

	<b>CLI</b> * junos_netconf & junos_command modules only	<b>NETCONF</b> * all modules except junos_netconf, which enables NETCONF
<b>Protocol</b>	SSH	XML over SSH
<b>Credentials</b>	uses SSH keys / SSH-agent if present accepts -u myuser -k if using password	uses SSH keys / SSH-agent if present accepts -u myuser -k if using password
<b>Indirect Access</b>	via a bastion (jump host)	via a bastion (jump host)
<b>Connection Settings</b>	ansible_connection: network_cli	ansible_connection: netconf
<b>Enable Mode</b> (Privilege Escalation)	not supported by Junos OS	not supported by Junos OS
<b>Returned Data Format</b>	stdout[0].	json: result[0]['software-information'][0][foo] lo0 text: result[1]. interface-information[0]. physical-interface[0]. name[0].data foo lo0 xml: result[1].rpc-reply. interface-information[0]. physical-interface[0]. name[0].data foo lo0

For legacy playbooks, Ansible still supports `ansible_connection=local` on all JUNOS modules. We recommend modernizing to use `ansible_connection=netconf` or `ansible_connection=network_cli` as soon as possible.



## Using CLI in Ansible 2.6

### Example CLI inventory [junos:vars]

```
[junos:vars]
ansible_connection=network_cli
ansible_network_os=junos
ansible_user=myuser
ansible_ssh_pass=!vault...
ansible_ssh_common_args='-o ProxyCommand="ssh -W %h:%p -q bastion01"'
```

- If you are using SSH keys (including an ssh-agent) you can remove the `ansible_ssh_pass` configuration.
- If you are accessing your host directly (not through a bastion/jump host) you can remove the `ansible_ssh_common_args` configuration.
- If you are accessing your host through a bastion/jump host, you cannot include your SSH password in the `ProxyCommand` directive. To prevent secrets from leaking out (for example in `ps` output), SSH does not support providing passwords via environment variables.

### Example CLI Task

```
- name: Retrieve Junos OS version
  junos_command:
    commands: show version
  when: ansible_network_os == 'junos'
```

## Using NETCONF in Ansible 2.6

### Enabling NETCONF

Before you can use NETCONF to connect to a switch, you must:

- install the `ncclient` python package on your control node(s) with `pip install ncclient`
- enable NETCONF on the Junos OS device(s)

To enable NETCONF on a new switch via Ansible, use the `junos_netconf` module via the CLI connection. Set up your platform-level variables just like in the CLI example above, then run a playbook task like this:

```
- name: Enable NETCONF
  connection: network_cli
  junos_netconf:
  when: ansible_network_os == 'junos'
```

Once NETCONF is enabled, change your variables to use the NETCONF connection.

### Example NETCONF inventory [junos:vars]

```
[junos:vars]
ansible_connection=netconf
ansible_network_os=junos
```

(continues on next page)

()

```
ansible_user=myuser
ansible_ssh_pass=!vault |
ansible_ssh_common_args='-o ProxyCommand="ssh -W %h:%p -q bastion01"'
```

## Example NETCONF Task

```
- name: Backup current switch config (junos)
  junos_config:
    backup: yes
  register: backup_junos_location
  when: ansible_network_os == 'junos'
```

: Never store passwords in plain text. We recommend using SSH keys to authenticate SSH connections. Ansible supports ssh-agent to manage your SSH keys. If you must use passwords to authenticate SSH connections, we recommend encrypting them with *Ansible Vault*.

## NXOS Platform Options

Cisco NXOS supports multiple connections. This page offers details on how each connection works in Ansible 2.6 and how to use it.

### Topics

- *NXOS Platform Options*
  - *Connections Available*
  - *Using CLI in Ansible 2.6*
    - \* *Example CLI group\_vars/nxos.yml*
    - \* *Example CLI Task*
  - *Using NX-API in Ansible 2.6*
    - \* *Enabling NX-API*
    - \* *Example NX-API group\_vars/nxos.yml*
    - \* *Example NX-API Task*

## Connections Available

	CLI	NX-API
Protocol	SSH	HTTP(S)
<b>Credentials</b>	uses SSH keys / SSH-agent if present accepts <code>-u myuser -k</code> if using password	uses HTTPS certificates if present
<b>Indirect Access</b>	via a bastion (jump host)	via a web proxy
<b>Connection Settings</b>	<code>ansible_connection:</code> <code>network_cli</code>	<code>ansible_connection:</code> <code>httpapi</code> OR <code>ansible_connection:</code> <code>local</code> with <code>transport: nxapi</code> in the provider dictionary
<b>Enable Mode</b> (Privilege Escalation) supported as of 2.5.3	supported - use <code>ansible_become: yes</code> with <code>ansible_become_method:</code> <code>enable</code> and <code>ansible_become_pass:</code>	not supported by NX-API
<b>Returned Data Format</b>	<code>stdout[0].</code>	<code>stdout[0].messages[0].</code>

For legacy playbooks, NXOS still supports `ansible_connection: local`. We recommend modernizing to use `ansible_connection: network_cli` or `ansible_connection: httpapi` as soon as possible.

## Using CLI in Ansible 2.6

### Example CLI `group_vars/nxos.yml`

```

ansible_connection: network_cli
ansible_network_os: nxos
ansible_user: myuser
ansible_ssh_pass: !vault...
ansible_become: yes
ansible_become_method: enable
ansible_become_pass: !vault...
ansible_ssh_common_args: '-o ProxyCommand="ssh -W %h:%p -q bastion01"'

```

- If you are using SSH keys (including an ssh-agent) you can remove the `ansible_ssh_pass` configuration.

- If you are accessing your host directly (not through a bastion/jump host) you can remove the `ansible_ssh_common_args` configuration.
- If you are accessing your host through a bastion/jump host, you cannot include your SSH password in the `ProxyCommand` directive. To prevent secrets from leaking out (for example in `ps` output), SSH does not support providing passwords via environment variables.

### Example CLI Task

```
- name: Backup current switch config (nxos)
  nxos_config:
    backup: yes
  register: backup_nxos_location
  when: ansible_network_os == 'nxos'
```

## Using NX-API in Ansible 2.6

### Enabling NX-API

Before you can use NX-API to connect to a switch, you must enable NX-API. To enable NX-API on a new switch via Ansible, use the `nxos_nxapi` module via the CLI connection. Set up `group_vars/nxos.yml` just like in the CLI example above, then run a playbook task like this:

```
- name: Enable NX-API
  nxos_nxapi:
    enable_http: yes
    enable_https: yes
  when: ansible_network_os == 'nxos'
```

To find out more about the options for enabling HTTP/HTTPS and local http see the `nxos_nxapi` module documentation.

Once NX-API is enabled, change your `group_vars/nxos.yml` to use the NX-API connection.

### Example NX-API `group_vars/nxos.yml`

```
ansible_connection: httpapi
ansible_network_os: nxos
ansible_user: myuser
ansible_ssh_pass: !vault...
proxy_env:
  http_proxy: http://proxy.example.com:8080
```

- If you are accessing your host directly (not through a web proxy) you can remove the `proxy_env` configuration.
- If you are accessing your host through a web proxy using https, change `http_proxy` to `https_proxy`.

## Example NX-API Task

```
- name: Backup current switch config (nxos)
  nxos_config:
    backup: yes
    register: backup_nxos_location
    environment: "{{ proxy_env }}"
    when: ansible_network_os == 'nxos'
```

In this example the `proxy_env` variable defined in `group_vars` gets passed to the `environment` option of the module used in the task.

: Never store passwords in plain text. We recommend using SSH keys to authenticate SSH connections. Ansible supports `ssh-agent` to manage your SSH keys. If you must use passwords to authenticate SSH connections, we recommend encrypting them with *Ansible Vault*.

## Settings by Platform

Network OS	ansible_network_os:	ansible_connection: settings available			
		network_cli	netconf	httpapi	local
Arista EOS*	eos	in v. >=2.5	N/A	in v. >=2.6	in v. >=2.4
Cisco ASA	asa	in v. >=2.5	N/A	N/A	in v. >=2.4
Cisco IOS*	ios	in v. >=2.5	N/A	N/A	in v. >=2.4
Cisco IOS XR*	iosxr	in v. >=2.5	N/A	N/A	in v. >=2.4
Cisco NX-OS*	nxos	in v. >=2.5	N/A	in v. >=2.6	in v. >=2.4
F5 BIG-IP	N/A	N/A	N/A	N/A	in v. >=2.0
F5 BIG-IQ	N/A	N/A	N/A	N/A	in v. >=2.0
Junos OS*	junos	in v. >=2.5	in v. >=2.5	N/A	in v. >=2.4
Nokia SR OS	sros	in v. >=2.5	N/A	N/A	in v. >=2.4
VyOS*	vyos	in v. >=2.5	N/A	N/A	in v. >=2.4

\* Maintained by Ansible Network Team

## 1.20 Ansible Galaxy

*Ansible Galaxy* refers to the [Galaxy](#) website where users can share roles, and to a command line tool for installing, creating and managing roles.

### Topics

- *Ansible Galaxy*
  - *The Website*
  - *The command line tool*
    - \* *Installing Roles*
    - *roles\_path*

- *version*
- *Installing multiple roles from a file*
- *Installing multiple roles from multiple files*
- *Dependencies*
- \* *Create roles*
  - *Force*
  - *Container Enabled*
  - *Using a Custom Role Skeleton*
- \* *Search for Roles*
- \* *Get more information about a role*
- \* *List installed roles*
- \* *Remove an installed role*
- \* *Authenticate with Galaxy*
- \* *Import a role*
  - *Branch*
  - *Role name*
  - *No wait*
- \* *Delete a role*
- \* *Travis integrations*
  - *List Travis integrations*
  - *Remove Travis integrations*

### 1.20.1 The Website

[Galaxy](#), is a free site for finding, downloading, and sharing community developed roles. Downloading roles from Galaxy is a great way to jumpstart your automation projects.

You can also use the site to share roles that you create. By authenticating with the site using your GitHub account, you're able to *import* roles, making them available to the Ansible community. Imported roles become available in the Galaxy search index and visible on the site, allowing users to discover and download them.

Learn more by viewing [the About page](#).

### 1.20.2 The command line tool

The `ansible-galaxy` command comes bundled with Ansible, and you can use it to install roles from Galaxy or directly from a git based SCM. You can also use it to create a new role, remove roles, or perform tasks on the Galaxy website.

The command line tool by default communicates with the Galaxy website API using the server address `https://galaxy.ansible.com`. Since the [Galaxy project](#) is an open source project, you may be running your own internal Galaxy server and wish to override the default server address. You can do this using the `-server` option or by

setting the Galaxy server value in your *ansible.cfg* file. For information on setting the value in *ansible.cfg* visit [Galaxy Settings](#).

## Installing Roles

Use the `ansible-galaxy` command to download roles from the [Galaxy website](#)

```
$ ansible-galaxy install username.role_name
```

### roles\_path

Be aware that by default Ansible downloads roles to the path specified by the environment variable `ANSIBLE_ROLES_PATH`. This can be set to a series of directories (i.e. */etc/ansible/roles:~/.ansible/roles*), in which case the first writable path will be used. When Ansible is first installed it defaults to */etc/ansible/roles*, which requires *root* privileges.

You can override this by setting the environment variable in your session, defining *roles\_path* in an *ansible.cfg* file, or by using the *-roles-path* option. The following provides an example of using *-roles-path* to install the role into the current working directory:

```
$ ansible-galaxy install --roles-path . geerlingguy.apache
```

:

*Configuring Ansible* All about configuration files

### version

You can install a specific version of a role from Galaxy by appending a comma and the value of a GitHub release tag. For example:

```
$ ansible-galaxy install geerlingguy.apache,v1.0.0
```

It's also possible to point directly to the git repository and specify a branch name or commit hash as the version. For example, the following will install a specific commit:

```
$ ansible-galaxy install git+https://github.com/geerlingguy/ansible-role-apache.git,  
→0b7cd353c0250e87a26e0499e59e7fd265cc2f25
```

## Installing multiple roles from a file

Beginning with Ansible 1.8 it is possible to install multiple roles by including the roles in a *requirements.yml* file. The format of the file is YAML, and the file extension must be either *.yml* or *.yaml*.

Use the following command to install roles included in *requirements.yml*:

```
$ ansible-galaxy install -r requirements.yml
```

Again, the extension is important. If the *.yml* extension is left off, the `ansible-galaxy` CLI assumes the file is in an older, now deprecated, "basic" format.

Each role in the file will have one or more of the following attributes:

**src** The source of the role. Use the format *username.role\_name*, if downloading from Galaxy; otherwise, provide a URL pointing to a repository within a git based SCM. See the examples below. This is a required attribute.

**scm** Specify the SCM. As of this writing only *git* or *hg* are supported. See the examples below. Defaults to *git*.

**version:** The version of the role to download. Provide a release tag value, commit hash, or branch name. Defaults to *master*.

**name:** Download the role to a specific name. Defaults to the Galaxy name when downloading from Galaxy, otherwise it defaults to the name of the repository.

Use the following example as a guide for specifying roles in *requirements.yml*:

```
# from galaxy
- src: yatesr.timezone

# from GitHub
- src: https://github.com/bennojoy/nginx

# from GitHub, overriding the name and specifying a specific tag
- src: https://github.com/bennojoy/nginx
  version: master
  name: nginx_role

# from a webserver, where the role is packaged in a tar.gz
- src: https://some.webserver.example.com/files/master.tar.gz
  name: http-role

# from Bitbucket
- src: git+http://bitbucket.org/willthames/git-ansible-galaxy
  version: vl.4

# from Bitbucket, alternative syntax and caveats
- src: http://bitbucket.org/willthames/hg-ansible-galaxy
  scm: hg

# from GitLab or other git-based scm
- src: git@gitlab.company.com:mygroup/ansible-base.git
  scm: git
  version: "0.1" # quoted, so YAML doesn't parse this as a floating-point value
```

## Installing multiple roles from multiple files

At a basic level, including requirements files allows you to break up bits of roles into smaller files. Role includes pull in roles from other files.

Use the following command to install roles includes in *requirements.yml* + *webserver.yml*

```
ansible-galaxy install -r requirements.yml
```

Content of the *requirements.yml* file:

```
# from galaxy
- src: yatesr.timezone

- include: <path_to_requirements>/webserver.yml
```



Content of the `webserver.yml` file:

```
# from github
- src: https://github.com/bennojoy/nginx

# from Bitbucket
- src: git+http://bitbucket.org/willthames/git-ansible-galaxy
  version: v1.4
```

## Dependencies

Roles can also be dependent on other roles, and when you install a role that has dependencies, those dependencies will automatically be installed.

You specify role dependencies in the `meta/main.yml` file by providing a list of roles. If the source of a role is Galaxy, you can simply specify the role in the format `username.role_name`. The more complex format used in `requirements.yml` is also supported, allowing you to provide `src`, `scm`, `version`, and `name`.

Tags are inherited *down* the dependency chain. In order for tags to be applied to a role and all its dependencies, the tag should be applied to the role, not to all the tasks within a role.

Roles listed as dependencies are subject to conditionals and tag filtering, and may not execute fully depending on what tags and conditionals are applied.

Dependencies found in Galaxy can be specified as follows:

```
dependencies:
  - geerlingguy.apache
  - geerlingguy.ansible
```

The complex form can also be used as follows:

```
dependencies:
  - src: geerlingguy.ansible
  - src: git+https://github.com/geerlingguy/ansible-role-composer.git
    version: 775396299f2da1f519f0d8885022ca2d6ee80ee8
    name: composer
```

When dependencies are encountered by `ansible-galaxy`, it will automatically install each dependency to the `roles_path`. To understand how dependencies are handled during play execution, see [Roles](#).

---

: At the time of this writing, the Galaxy website expects all role dependencies to exist in Galaxy, and therefore dependencies to be specified in the `username.role_name` format. If you import a role with a dependency where the `src` value is a URL, the import process will fail.

---

## Create roles

Use the `init` command to initialize the base structure of a new role, saving time on creating the various directories and `main.yml` files a role requires

```
$ ansible-galaxy init role_name
```

The above will create the following directory structure in the current working directory:

```
README.md
.travis.yml
defaults/
    main.yml
files/
handlers/
    main.yml
meta/
    main.yml
templates/
tests/
    inventory
    test.yml
vars/
    main.yml
```

## Force

If a directory matching the name of the role already exists in the current working directory, the `init` command will result in an error. To ignore the error use the `-force` option. Force will create the above subdirectories and files, replacing anything that matches.

## Container Enabled

If you are creating a Container Enabled role, use the `-container-enabled` option. This will create the same directory structure as above, but populate it with default files appropriate for a Container Enabled role. For instance, the `README.md` has a slightly different structure, the `.travis.yml` file tests the role using [Ansible Container](#), and the meta directory includes a `container.yml` file.

## Using a Custom Role Skeleton

A custom role skeleton directory can be supplied as follows:

```
$ ansible-galaxy init --role-skeleton=/path/to/skeleton role_name
```

When a skeleton is provided, `init` will:

- copy all files and directories from the skeleton to the new role
- any `.j2` files found outside of a templates folder will be rendered as templates. The only useful variable at the moment is `role_name`
- The `.git` folder and any `.git_keep` files will not be copied

Alternatively, the `role_skeleton` and ignoring of files can be configured via `ansible.cfg`

```
[galaxy]
role_skeleton = /path/to/skeleton
role_skeleton_ignore = ^.git$,^.*/.git_keep$
```

## Search for Roles

Search the Galaxy database by tags, platforms, author and multiple keywords. For example:

```
$ ansible-galaxy search elasticsearch --author geerlingguy
```

The search command will return a list of the first 1000 results matching your search:

```
Found 2 roles matching your search:
```

Name	Description
----	-----
geerlingguy.elasticsearch	Elasticsearch for Linux.
geerlingguy.elasticsearch-curator	Elasticsearch curator for Linux.

### Get more information about a role

Use the `info` command to view more detail about a specific role:

```
$ ansible-galaxy info username.role_name
```

This returns everything found in Galaxy for the role:

```
Role: username.role_name
  description: Installs and configures a thing, a distributed, highly available_
↪NoSQL thing.
  active: True
  commit: c01947b7bc89ebc0b8a2e298b87ab416aed9dd57
  commit_message: Adding travis
  commit_url: https://github.com/username/repo_name/commit/
↪c01947b7bc89ebc0b8a2e298b87ab
  company: My Company, Inc.
  created: 2015-12-08T14:17:52.773Z
  download_count: 1
  forks_count: 0
  github_branch:
  github_repo: repo_name
  github_user: username
  id: 6381
  is_valid: True
  issue_tracker_url:
  license: Apache
  min_ansible_version: 1.4
  modified: 2015-12-08T18:43:49.085Z
  namespace: username
  open_issues_count: 0
  path: /Users/username/projects/roles
  scm: None
  src: username.repo_name
  stargazers_count: 0
  travis_status_url: https://travis-ci.org/username/repo_name.svg?branch=master
  version:
  watchers_count: 1
```

### List installed roles

Use `list` to show the name and version of each role installed in the `roles_path`.

```
$ ansible-galaxy list

- chouseknecht.role-install_mongod, master
- chouseknecht.test-role-1, v1.0.2
- chrismeyersfsu.role-iptables, master
- chrismeyersfsu.role-required_vars, master
```

## Remove an installed role

Use `remove` to delete a role from *roles\_path*:

```
$ ansible-galaxy remove username.role_name
```

## Authenticate with Galaxy

Using the `import`, `delete` and `setup` commands to manage your roles on the Galaxy website requires authentication, and the `login` command can be used to do just that. Before you can use the `login` command, you must create an account on the Galaxy website.

The `login` command requires using your GitHub credentials. You can use your username and password, or you can create a [personal access token](#). If you choose to create a token, grant minimal access to the token, as it is used just to verify identity.

The following shows authenticating with the Galaxy website using a GitHub username and password:

```
$ ansible-galaxy login

We need your GitHub login to identify you.
This information will not be sent to Galaxy, only to api.github.com.
The password will not be displayed.

Use --github-token if you do not want to enter your password.

Github Username: dsmith
Password for dsmith:
Successfully logged into Galaxy as dsmith
```

When you choose to use your username and password, your password is not sent to Galaxy. It is used to authenticate with GitHub and create a personal access token. It then sends the token to Galaxy, which in turn verifies that your identity and returns a Galaxy access token. After authentication completes the GitHub token is destroyed.

If you do not wish to use your GitHub password, or if you have two-factor authentication enabled with GitHub, use the `-github-token` option to pass a personal access token that you create.

## Import a role

The `import` command requires that you first authenticate using the `login` command. Once authenticated you can import any GitHub repository that you own or have been granted access.

Use the following to import to role:

```
$ ansible-galaxy import github_user github_repo
```

By default the command will wait for Galaxy to complete the import process, displaying the results as the import progresses:

```

Successfully submitted import request 41
Starting import 41: role_name=myrole repo=githubuser/ansible-role-repo ref=
Retrieving GitHub repo githubuser/ansible-role-repo
Accessing branch: master
Parsing and validating meta/main.yml
Parsing galaxy_tags
Parsing platforms
Adding dependencies
Parsing and validating README.md
Adding repo tags as role versions
Import completed
Status SUCCESS : warnings=0 errors=0

```

## Branch

Use the `--branch` option to import a specific branch. If not specified, the default branch for the repo will be used.

## Role name

By default the name given to the role will be derived from the GitHub repository name. However, you can use the `--role-name` option to override this and set the name.

## No wait

If the `--no-wait` option is present, the command will not wait for results. Results of the most recent import for any of your roles is available on the Galaxy web site by visiting *My Imports*.

## Delete a role

The `delete` command requires that you first authenticate using the `login` command. Once authenticated you can remove a role from the Galaxy web site. You are only allowed to remove roles where you have access to the repository in GitHub.

Use the following to delete a role:

```
$ ansible-galaxy delete github_user github_repo
```

This only removes the role from Galaxy. It does not remove or alter the actual GitHub repository.

## Travis integrations

You can create an integration or connection between a role in Galaxy and [Travis](#). Once the connection is established, a build in Travis will automatically trigger an import in Galaxy, updating the search index with the latest information about the role.

You create the integration using the `setup` command, but before an integration can be created, you must first authenticate using the `login` command; you will also need an account in Travis, and your Travis token. Once you're ready, use the following command to create the integration:

```
$ ansible-galaxy setup travis github_user github_repo xxx-travis-token-xxx
```

The setup command requires your Travis token, however the token is not stored in Galaxy. It is used along with the GitHub username and repo to create a hash as described in [the Travis documentation](#). The hash is stored in Galaxy and used to verify notifications received from Travis.

The setup command enables Galaxy to respond to notifications. To configure Travis to run a build on your repository and send a notification, follow the [Travis getting started guide](#).

To instruct Travis to notify Galaxy when a build completes, add the following to your `.travis.yml` file:

```
notifications:
  webhooks: https://galaxy.ansible.com/api/v1/notifications/
```

### List Travis integrations

Use the `--list` option to display your Travis integrations:

```
$ ansible-galaxy setup --list
```

ID	Source	Repo
2	travis	github_user/github_repo
1	travis	github_user/github_repo

### Remove Travis integrations

Use the `--remove` option to disable and remove a Travis integration:

```
$ ansible-galaxy setup --remove ID
```

Provide the ID of the integration to be disabled. You can find the ID by using the `--list` option.

:

**Roles** All about ansible roles

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**irc.freenode.net** #ansible IRC chat channel

## 1.21 YAML Syntax

This page provides a basic overview of correct YAML syntax, which is how Ansible playbooks (our configuration management language) are expressed.

We use YAML because it is easier for humans to read and write than other common data formats like XML or JSON. Further, there are libraries available in most programming languages for working with YAML.

You may also wish to read [Working With Playbooks](#) at the same time to see how this is used in practice.

### 1.21.1 YAML Basics

For Ansible, nearly every YAML file starts with a list. Each item in the list is a list of key/value pairs, commonly called a "hash" or a "dictionary". So, we need to know how to write lists and dictionaries in YAML.

There's another small quirk to YAML. All YAML files (regardless of their association with Ansible or not) can optionally begin with `---` and end with `...`. This is part of the YAML format and indicates the start and end of a document.

All members of a list are lines beginning at the same indentation level starting with a `" - "` (a dash and a space):

```
---
# A list of tasty fruits
fruits:
  - Apple
  - Orange
  - Strawberry
  - Mango
...
```

A dictionary is represented in a simple `key: value` form (the colon must be followed by a space):

```
# An employee record
martin:
  name: Martin D'vloper
  job: Developer
  skill: Elite
```

More complicated data structures are possible, such as lists of dictionaries, dictionaries whose values are lists or a mix of both:

```
# Employee records
- martin:
  name: Martin D'vloper
  job: Developer
  skills:
    - python
    - perl
    - pascal
- tabitha:
  name: Tabitha Bitumen
  job: Developer
  skills:
    - lisp
    - fortran
    - erlang
```

Dictionaries and lists can also be represented in an abbreviated form if you really want to:

```
---
martin: {name: Martin D'vloper, job: Developer, skill: Elite}
fruits: ['Apple', 'Orange', 'Strawberry', 'Mango']
```

These are called "Flow collections".

Ansible doesn't really use these too much, but you can also specify a boolean value (true/false) in several forms:

```
create_key: yes
needs_agent: no
knows_oop: True
likes_emacs: TRUE
uses_cvs: false
```

Values can span multiple lines using `|` or `>`. Spanning multiple lines using a "Literal Block Scalar" `|` will include the newlines and any trailing spaces. Using a "Folded Block Scalar" `>` will fold newlines to spaces; it's used to make what would otherwise be a very long line easier to read and edit. In either case the indentation will be ignored. Examples are:

```
include_newlines: |
    exactly as you see
    will appear these three
    lines of poetry

fold_newlines: >
    this is really a
    single line of text
    despite appearances
```

While in the above `>` example all newlines are folded into spaces, there are two ways to enforce a newline to be kept:

```
fold_some_newlines: >
    a
    b

    c
    d
    e
    f
same_as: "a b\nc d\n e\nf\n"
```

Let's combine what we learned so far in an arbitrary YAML example. This really has nothing to do with Ansible, but will give you a feel for the format:

```
---
# An employee record
name: Martin D'vloper
job: Developer
skill: Elite
employed: True
foods:
  - Apple
  - Orange
  - Strawberry
  - Mango
languages:
  perl: Elite
  python: Elite
  pascal: Lame
education: |
  4 GCSEs
  3 A-Levels
  BSc in the Internet of Things
```

That's all you really need to know about YAML to start writing *Ansible* playbooks.

### 1.21.2 Gotchas

While you can put just about anything into an unquoted scalar, there are some exceptions. A colon followed by a space (or newline) `:` `` `` is an indicator for a mapping. A space followed by the pound sign `` ` #` starts a comment.



Because of this, the following is going to result in a YAML syntax error:

```
foo: somebody said I should put a colon here: so I did
windows_drive: c:
```

...but this will work:

```
windows_path: c:\windows
```

You will want to quote hash values using colons followed by a space or the end of the line:

```
foo: 'somebody said I should put a colon here: so I did'
windows_drive: 'c:'
```

...and then the colon will be preserved.

Alternatively, you can use double quotes:

```
foo: "somebody said I should put a colon here: so I did"
windows_drive: "c:"
```

The difference between single quotes and double quotes is that in double quotes you can use escapes:

```
foo: "a \t TAB and a \n NEWLINE"
```

The list of allowed escapes can be found in the [YAML Specification](#) under "Escape Sequences" (YAML 1.1) or "Escape Characters" (YAML 1.2).

The following is invalid YAML:

```
foo: "an escaped \' single quote"
```

Further, Ansible uses "{ { var } }" for variables. If a value after a colon starts with a "{", YAML will think it is a dictionary, so you must quote it, like so:

```
foo: "{ { variable } }"
```

If your value starts with a quote the entire value must be quoted, not just part of it. Here are some additional examples of how to properly quote things:

```
foo: "{ { variable } }/additional/string/literal"
foo2: "{ { variable } }\\backslashes\\are\\also\\special\\characters"
foo3: "even if it's just a string literal it must all be quoted"
```

Not valid:

```
foo: "E:\\path\\"rest\\of\\path"
```

In addition to ' and " there are a number of characters that are special (or reserved) and cannot be used as the first character of an unquoted scalar: [ ] { } > | \* & ! % # ` @ , .

You should also be aware of ? : -. In YAML, they are allowed at the beginning of a string if a non-space character follows, but YAML processor implementations differ, so it's better to use quotes.

In Flow Collections, the rules are a bit more strict:

```
a scalar in block mapping: this } is [ all , valid
flow mapping: { key: "you { should [ use , quotes here" }
```

Boolean conversion is helpful, but this can be a problem when you want a literal yes or other boolean values as a string. In these cases just use quotes:

```
non_boolean: "yes"
other_string: "False"
```

YAML converts certain strings into floating-point values, such as the string *1.0*. If you need to specify a version number (in a requirements.yml file, for example), you will need to quote the value if it looks like a floating-point value:

```
version: "1.0"
```

:

**Working With Playbooks** Learn what playbooks can do and how to write/run them.

**YAMLLint** YAML Lint (online) helps you debug YAML syntax if you are having problems

**Github examples directory** Complete playbook files from the github project source

**Wikipedia YAML syntax reference** A good guide to YAML syntax

**Mailing List** Questions? Help? Ideas? Stop by the list on Google Groups

**irc.freenode.net** #ansible IRC chat channel and #yaml for YAML specific questions

**YAML 1.1 Specification** The Specification for YAML 1.1, which PyYAML and libyaml are currently implementing

**YAML 1.2 Specification** For completeness, YAML 1.2 is the successor of 1.1

## 1.22 Python 3 Support

Ansible 2.5 and above have support for Python 3. Previous to 2.5, the Python 3 support was considered a tech preview. This topic discusses how to setup your controller and managed machines to use Python 3.

---

: Ansible supports Python version 3.5 and above only.

---

### 1.22.1 On the controller side

The easiest way to run **/usr/bin/ansible** under Python 3 is to install it with the Python3 version of pip. This will make the default **/usr/bin/ansible** run with Python3:

```
$ pip3 install ansible
$ ansible --version | grep "python version"
python version = 3.6.2 (default, Sep 22 2017, 08:28:09) [GCC 7.2.1 20170915 (Red_Hat 7.2.1-2)]
```

If you are running Ansible *Running From Source* and want to use Python 3 with your source checkout, run your command via python3. For example:

```
$ source ./hacking/env-setup
$ python3 $(which ansible) localhost -m ping
$ python3 $(which ansible-playbook) sample-playbook.yml
```

: Individual Linux distribution packages may be packaged for Python2 or Python3. When running from distro packages you'll only be able to use Ansible with the Python version for which it was installed. Sometimes distros will provide a means of installing for several Python versions (via a separate package or via some commands that are run after install). You'll need to check with your distro to see if that applies in your case.

### 1.22.2 Using Python 3 on the managed machines with commands and playbooks

- Set the `ansible_python_interpreter` configuration option to `/usr/bin/python3`. The `ansible_python_interpreter` configuration option is usually set as an inventory variable associated with a host or group of hosts:

```
# Example inventory that makes an alias for localhost that uses Python3
localhost-py3 ansible_host=localhost ansible_connection=local ansible_python_
↪interpreter=/usr/bin/python3

# Example of setting a group of hosts to use Python3
[py3-hosts]
ubuntu16
fedora27

[py3-hosts:vars]
ansible_python_interpreter=/usr/bin/python3
```

:

*Working with Inventory* for more information.

- Run your command or playbook:

```
$ ansible localhost-py3 -m ping
$ ansible-playbook sample-playbook.yml
```

Note that you can also use the `-e` command line option to manually set the python interpreter when you run a command. This can be useful if you want to test whether a specific module or playbook has any bugs under Python 3. For example:

```
$ ansible localhost -m ping -e 'ansible_python_interpreter=/usr/bin/python3'
$ ansible-playbook sample-playbook.yml -e 'ansible_python_interpreter=/usr/bin/python3'
↪'
```

### 1.22.3 What to do if an incompatibility is found

We have spent several releases squashing bugs and adding new tests so that Ansible's core feature set runs under both Python 2 and Python 3. However, bugs may still exist in edge cases and many of the modules shipped with Ansible are maintained by the community and not all of those may be ported yet.

If you find a bug running under Python 3 you can submit a bug report on [Ansible's GitHub project](#). Be sure to mention Python3 in the bug report so that the right people look at it.

If you would like to fix the code and submit a pull request on github, you can refer to [Ansible and Python 3](#) for information on how we fix common Python3 compatibility issues in the Ansible codebase.

## 1.23 Testing Strategies

### 1.23.1 Integrating Testing With Ansible Playbooks

Many times, people ask, "how can I best integrate testing with Ansible playbooks?" There are many options. Ansible is actually designed to be a "fail-fast" and ordered system, therefore it makes it easy to embed testing directly in Ansible playbooks. In this chapter, we'll go into some patterns for integrating tests of infrastructure and discuss the right level of testing that may be appropriate.

---

: This is a chapter about testing the application you are deploying, not the chapter on how to test Ansible modules during development. For that content, please hop over to the Development section.

---

By incorporating a degree of testing into your deployment workflow, there will be fewer surprises when code hits production and, in many cases, tests can be leveraged in production to prevent failed updates from migrating across an entire installation. Since it's push-based, it's also very easy to run the steps on the localhost or testing servers. Ansible lets you insert as many checks and balances into your upgrade workflow as you would like to have.

### 1.23.2 The Right Level of Testing

Ansible resources are models of desired-state. As such, it should not be necessary to test that services are started, packages are installed, or other such things. Ansible is the system that will ensure these things are declaratively true. Instead, assert these things in your playbooks.

```
tasks:
  - service:
      name: foo
      state: started
      enabled: yes
```

If you think the service may not be started, the best thing to do is request it to be started. If the service fails to start, Ansible will yell appropriately. (This should not be confused with whether the service is doing something functional, which we'll show more about how to do later).

### 1.23.3 Check Mode As A Drift Test

In the above setup, *-check* mode in Ansible can be used as a layer of testing as well. If running a deployment playbook against an existing system, using the *-check* flag to the *ansible* command will report if Ansible thinks it would have had to have made any changes to bring the system into a desired state.

This can let you know up front if there is any need to deploy onto the given system. Ordinarily scripts and commands don't run in check mode, so if you want certain steps to always execute in check mode, such as calls to the script module, disable check mode for those tasks:

```
roles:
  - webserver

tasks:
  - script: verify.sh
    check_mode: no
```

### 1.23.4 Modules That Are Useful for Testing

Certain playbook modules are particularly good for testing. Below is an example that ensures a port is open:

```
tasks:
  - wait_for:
      host: "{{ inventory_hostname }}"
      port: 22
      delegate_to: localhost
```

Here's an example of using the URI module to make sure a web service returns:

```
tasks:
  - action: uri url=http://www.example.com return_content=yes
    register: webpage

  - fail:
      msg: 'service is not happy'
      when: "'AWESOME' not in webpage.content"
```

It's easy to push an arbitrary script (in any language) on a remote host and the script will automatically fail if it has a non-zero return code:

```
tasks:
  - script: test_script1
  - script: test_script2 --parameter value --parameter2 value
```

If using roles (you should be, roles are great!), scripts pushed by the script module can live in the 'files/' directory of a role.

And the assert module makes it very easy to validate various kinds of truth:

```
tasks:
  - shell: /usr/bin/some-command --parameter value
    register: cmd_result

  - assert:
      that:
        - "'not ready' not in cmd_result.stderr"
        - "'gizmo enabled' in cmd_result.stdout"
```

Should you feel the need to test for existence of files that are not declaratively set by your Ansible configuration, the 'stat' module is a great choice:

```
tasks:
  - stat:
      path: /path/to/something
      register: p

  - assert:
      that:
        - p.stat.exists and p.stat.isdir
```

As mentioned above, there's no need to check things like the return codes of commands. Ansible is checking them automatically. Rather than checking for a user to exist, consider using the `user` module to make it exist.


Ansible is a fail-fast system, so when there is an error creating that user, it will stop the playbook run. You do not have to check up behind it.

### 1.23.5 Testing Lifecycle

If writing some degree of basic validation of your application into your playbooks, they will run every time you deploy.

As such, deploying into a local development VM and a staging environment will both validate that things are according to plan ahead of your production deploy.

Your workflow may be something like this:

- Use the same playbook all the time with embedded tests in development
- Use the playbook to deploy to a staging environment (with the same playbooks) that  simulates production
- Run an integration test battery written by your QA team against staging
- Deploy to production, with the same integrated tests.

Something like an integration test battery should be written by your QA team if you are a production webservice. This would include things like Selenium tests or automated API tests and would usually not be something embedded into your Ansible playbooks.

However, it does make sense to include some basic health checks into your playbooks, and in some cases it may be possible to run a subset of the QA battery against remote nodes. This is what the next section covers.

### 1.23.6 Integrating Testing With Rolling Updates

If you have read into *Delegation, Rolling Updates, and Local Actions* it may quickly become apparent that the rolling update pattern can be extended, and you can use the success or failure of the playbook run to decide whether to add a machine into a load balancer or not.

This is the great culmination of embedded tests:

```
---
- hosts: webserver
  serial: 5

  pre_tasks:

    - name: take out of load balancer pool
      command: /usr/bin/take_out_of_pool {{ inventory_hostname }}
      delegate_to: 127.0.0.1

  roles:

    - common
    - webserver
    - apply_testing_checks

  post_tasks:

    - name: add back to load balancer pool
```

(continues on next page)

()

```
command: /usr/bin/add_back_to_pool {{ inventory_hostname }}
delegate_to: 127.0.0.1
```

Of course in the above, the "take out of the pool" and "add back" steps would be replaced with a call to a Ansible load balancer module or appropriate shell command. You might also have steps that use a monitoring module to start and end an outage window for the machine.

However, what you can see from the above is that tests are used as a gate – if the "apply\_testing\_checks" step is not performed, the machine will not go back into the pool.

Read the delegation chapter about "max\_fail\_percentage" and you can also control how many failing tests will stop a rolling update from proceeding.

This above approach can also be modified to run a step from a testing machine remotely against a machine:

```
---
- hosts: webserver
  serial: 5

  pre_tasks:

    - name: take out of load balancer pool
      command: /usr/bin/take_out_of_pool {{ inventory_hostname }}
      delegate_to: 127.0.0.1

  roles:

    - common
    - webserver

  tasks:
    - script: /srv/qa_team/app_testing_script.sh --server {{ inventory_hostname }}
      delegate_to: testing_server

  post_tasks:

    - name: add back to load balancer pool
      command: /usr/bin/add_back_to_pool {{ inventory_hostname }}
      delegate_to: 127.0.0.1
```

In the above example, a script is run from the testing server against a remote node prior to bringing it back into the pool.

In the event of a problem, fix the few servers that fail using Ansible's automatically generated retry file to repeat the deploy on just those servers.

### 1.23.7 Achieving Continuous Deployment

If desired, the above techniques may be extended to enable continuous deployment practices.

The workflow may look like this:

- Write and use automation to deploy local development VMs
  - Have a CI system like Jenkins deploy to a staging environment on every code change
  - The deploy job calls testing scripts to pass/fail a build on every deploy
  - If the deploy job succeeds, it runs the same deploy playbook against production\_
- inventory (continues on next page)

---

Some Ansible users use the above approach to deploy a half-dozen or dozen times an hour without taking all of their infrastructure offline. A culture of automated QA is vital if you wish to get to this level.

If you are still doing a large amount of manual QA, you should still make the decision on whether to deploy manually as well, but it can still help to work in the rolling update patterns of the previous section and incorporate some basic health checks using modules like `'script'`, `'stat'`, `'uri'`, and `'assert'`.

### 1.23.8 Conclusion

Ansible believes you should not need another framework to validate basic things of your infrastructure is true. This is the case because Ansible is an order-based system that will fail immediately on unhandled errors for a host, and prevent further configuration of that host. This forces errors to the top and shows them in a summary at the end of the Ansible run.

However, as Ansible is designed as a multi-tier orchestration system, it makes it very easy to incorporate tests into the end of a playbook run, either using loose tasks or roles. When used with rolling updates, testing steps can decide whether to put a machine back into a load balanced pool or not.

Finally, because Ansible errors propagate all the way up to the return code of the Ansible program itself, and Ansible by default runs in an easy push-based mode, Ansible is a great step to put into a build environment if you wish to use it to roll out systems as part of a Continuous Integration/Continuous Delivery pipeline, as is covered in sections above.

The focus should not be on infrastructure testing, but on application testing, so we strongly encourage getting together with your QA team and ask what sort of tests would make sense to run every time you deploy development VMs, and which sort of tests they would like to run against the staging environment on every deploy. Obviously at the development stage, unit tests are great too. But don't unit test your playbook. Ansible describes states of resources declaratively, so you don't have to. If there are cases where you want to be sure of something though, that's great, and things like `stat/assert` are great go-to modules for that purpose.

In all, testing is a very organizational and site-specific thing. Everybody should be doing it, but what makes the most sense for your environment will vary with what you are deploying and who is using it – but everyone benefits from a more robust and reliable deployment system.

:

**all\_modules** All the documentation for Ansible modules

*Working With Playbooks* An introduction to playbooks

*Delegation, Rolling Updates, and Local Actions* Delegation, useful for working with load balancers, clouds, and locally executed steps.

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## 1.24 Frequently Asked Questions

Here are some commonly asked questions and their answers.



### 1.24.1 How can I set the PATH or any other environment variable for a task or entire playbook?

Setting environment variables can be done with the *environment* keyword. It can be used at the task or other levels in the play:

```
environment:
  PATH: "{{ ansible_env.PATH }}:/thingy/bin"
  SOME: value
```

: starting in 2.0.1 the setup task from gather\_facts also inherits the environment directive from the play, you might need to use the *default* filter to avoid errors if setting this at play level.

### 1.24.2 How do I handle different machines needing different user accounts or ports to log in with?

Setting inventory variables in the inventory file is the easiest way.

For instance, suppose these hosts have different usernames and ports:

```
[webservers]
asdf.example.com  ansible_port=5000  ansible_user=alice
jkl.example.com   ansible_port=5001  ansible_user=bob
```

You can also dictate the connection type to be used, if you want:

```
[testcluster]
localhost          ansible_connection=local
/path/to/chroot1    ansible_connection=chroot
foo.example.com     ansible_connection=paramiko
```

You may also wish to keep these in group variables instead, or file them in a `group_vars/<groupname>` file. See the rest of the documentation for more information about how to organize variables.

### 1.24.3 How do I get ansible to reuse connections, enable Kerberized SSH, or have Ansible pay attention to my local SSH config file?

Switch your default connection type in the configuration file to 'ssh', or use '-c ssh' to use Native OpenSSH for connections instead of the python paramiko library. In Ansible 1.2.1 and later, 'ssh' will be used by default if OpenSSH is new enough to support ControlPersist as an option.

Paramiko is great for starting out, but the OpenSSH type offers many advanced options. You will want to run Ansible from a machine new enough to support ControlPersist, if you are using this connection type. You can still manage older clients. If you are using RHEL 6, CentOS 6, SLES 10 or SLES 11 the version of OpenSSH is still a bit old, so consider managing from a Fedora or openSUSE client even though you are managing older nodes, or just use paramiko.

We keep paramiko as the default as if you are first installing Ansible on an EL box, it offers a better experience for new users.

### 1.24.4 How do I configure a jump host to access servers that I have no direct access to?

You can set a *ProxyCommand* in the *ansible\_ssh\_common\_args* inventory variable. Any arguments specified in this variable are added to the sftp/scp/ssh command line when connecting to the relevant host(s). Consider the following inventory group:

```
[gatewayed]
foo ansible_host=192.0.2.1
bar ansible_host=192.0.2.2
```

You can create *group\_vars/gatewayed.yml* with the following contents:

```
ansible_ssh_common_args: '-o ProxyCommand="ssh -W %h:%p -q user@gateway.example.com"'
```

Ansible will append these arguments to the command line when trying to connect to any hosts in the group *gatewayed*. (These arguments are used in addition to any *ssh\_args* from *ansible.cfg*, so you do not need to repeat global *ControlPersist* settings in *ansible\_ssh\_common\_args*.)

Note that *ssh -W* is available only with OpenSSH 5.4 or later. With older versions, it's necessary to execute *nc %h:%p* or some equivalent command on the bastion host.

With earlier versions of Ansible, it was necessary to configure a suitable *ProxyCommand* for one or more hosts in *~/.ssh/config*, or globally by setting *ssh\_args* in *ansible.cfg*.

### 1.24.5 How do I speed up management inside EC2?

Don't try to manage a fleet of EC2 machines from your laptop. Connect to a management node inside EC2 first and run Ansible from there.

### 1.24.6 How do I handle python not having a Python interpreter at */usr/bin/python* on a remote machine?

While you can write Ansible modules in any language, most Ansible modules are written in Python, including the ones central to letting Ansible work.

By default, Ansible assumes it can find a */usr/bin/python* on your remote system that is either Python2, version 2.6 or higher or Python3, 3.5 or higher.

Setting the inventory variable *ansible\_python\_interpreter* on any host will tell Ansible to auto-replace the Python interpreter with that value instead. Thus, you can point to any Python you want on the system if */usr/bin/python* on your system does not point to a compatible Python interpreter.

Some platforms may only have Python 3 installed by default. If it is not installed as */usr/bin/python*, you will need to configure the path to the interpreter via *ansible\_python\_interpreter*. Although most core modules will work with Python 3, there may be some special purpose ones which do not or you may encounter a bug in an edge case. As a temporary workaround you can install Python 2 on the managed host and configure Ansible to use that Python via *ansible\_python\_interpreter*. If there's no mention in the module's documentation that the module requires Python 2, you can also report a bug on our [bug tracker](#) so that the incompatibility can be fixed in a future release.

Do not replace the shebang lines of your python modules. Ansible will do this for you automatically at deploy time.

Also, this works for ANY interpreter, i.e ruby: *ansible\_ruby\_interpreter*, perl: *ansible\_perl\_interpreter*, etc, so you can use this for custom modules written in any scripting language and control the interpreter location.

Keep in mind that if you put *env* in your module shebang line (*#!/usr/bin/env <other>*), this facility will be ignored so you will be at the mercy of the remote *\$PATH*.

### 1.24.7 How do I handle the package dependencies required by Ansible package dependencies during Ansible installation ?

While installing Ansible, sometimes you may encounter errors such as *No package 'libffi' found* or *fatal error: Python.h: No such file or directory*. These errors are generally caused by the missing packages which are dependencies of the packages required by Ansible. For example, *libffi* package is dependency of *pynacl* and *paramiko* (Ansible -> paramiko -> pynacl -> libffi).

In order to solve these kinds of dependency issue, you may need to install required packages using the OS native package managers (e.g., *yum*, *dnf* or *apt*) or as mentioned in the package installation guide.

Please refer the documentation of the respective package for such dependencies and their installation methods.

### 1.24.8 Common Platform Issues

#### Running in a virtualenv

You can install Ansible into a virtualenv on the controller quite simply:

```
$ virtualenv ansible
$ source ./ansible/bin/activate
$ pip install ansible
```

If you want to run under Python 3 instead of Python 2 you may want to change that slightly:

```
$ virtualenv ansible
$ source ./ansible/bin/activate
$ pip3 install ansible
```

If you need to use any libraries which are not available via pip (for instance, SELinux Python bindings on systems such as Red Hat Enterprise Linux or Fedora that have SELinux enabled) then you need to install them into the virtualenv. There are two methods:

- When you create the virtualenv, specify `--system-site-packages` to make use of any libraries installed in the system's Python:

```
$ virtualenv ansible --system-site-packages
```

- Copy those files in manually from the system. For instance, for SELinux bindings you might do:

```
$ virtualenv ansible --system-site-packages
$ cp -r -v /usr/lib64/python3.*/site-packages/selinux/ ./py3-ansible/lib64/
↪python3.*/site-packages/
$ cp -v /usr/lib64/python3.*/site-packages/*selinux*.so ./py3-ansible/lib64/
↪python3.*/site-packages/
```

#### Running on BSD

:

*Working with BSD*

## Running on Solaris

By default, Solaris 10 and earlier run a non-POSIX shell which does not correctly expand the default tmp directory Ansible uses ( `~/ .ansible/tmp` ). If you see module failures on Solaris machines, this is likely the problem. There are several workarounds:

- You can set `remote_tmp` to a path that will expand correctly with the shell you are using (see the plugin documentation for C shell, fish shell, and Powershell). For example, in the ansible config file you can set:

```
remote_tmp=$HOME/.ansible/tmp
```

In Ansible 2.5 and later, you can also set it per-host in inventory like this:

```
solarisl ansible_remote_tmp=$HOME/.ansible/tmp
```

- You can set `ansible_shell_executable` to the path to a POSIX compatible shell. For instance, many Solaris hosts have a POSIX shell located at `/usr/xpg4/bin/sh` so you can set this in inventory like so:

```
solarisl ansible_shell_executable=/usr/xpg4/bin/sh
```

(bash, ksh, and zsh should also be POSIX compatible if you have any of those installed).

### 1.24.9 What is the best way to make content reusable/redistributable?

If you have not done so already, read all about "Roles" in the playbooks documentation. This helps you make playbook content self-contained, and works well with things like git submodules for sharing content with others.

If some of these plugin types look strange to you, see the API documentation for more details about ways Ansible can be extended.

### 1.24.10 Where does the configuration file live and what can I configure in it?

See *Configuring Ansible*.

### 1.24.11 How do I disable cowsay?

If cowsay is installed, Ansible takes it upon itself to make your day happier when running playbooks. If you decide that you would like to work in a professional cow-free environment, you can either uninstall cowsay, set `nocows=1` in `ansible.cfg`, or set the `ANSIBLE_NOCOWS` environment variable:

```
export ANSIBLE_NOCOWS=1
```

### 1.24.12 How do I see a list of all of the ansible\_ variables?

Ansible by default gathers "facts" about the machines under management, and these facts can be accessed in Playbooks and in templates. To see a list of all of the facts that are available about a machine, you can run the "setup" module as an ad-hoc action:

```
ansible -m setup hostname
```

This will print out a dictionary of all of the facts that are available for that particular host. You might want to pipe the output to a pager. This does NOT include inventory variables or internal 'magic' variables. See the next question if you need more than just 'facts'.

### 1.24.13 How do I see all the inventory variables defined for my host?

By running the following command, you can see inventory variables for a host:

```
ansible-inventory --list --yaml
```

### 1.24.14 How do I see all the variables specific to my host?

To see all host specific variables, which might include facts and other sources:

```
ansible -m debug -a "var=hostvars['hostname']" localhost
```

Unless you are using a fact cache, you normally need to use a play that gathers facts first, for facts included in the task above.

### 1.24.15 How do I loop over a list of hosts in a group, inside of a template?

A pretty common pattern is to iterate over a list of hosts inside of a host group, perhaps to populate a template configuration file with a list of servers. To do this, you can just access the "\$groups" dictionary in your template, like this:

```
{% for host in groups['db_servers'] %}
    {{ host }}
{% endfor %}
```

If you need to access facts about these hosts, for instance, the IP address of each hostname, you need to make sure that the facts have been populated. For example, make sure you have a play that talks to db\_servers:

```
- hosts: db_servers
  tasks:
    - debug: msg="doesn't matter what you do, just that they were talked to_
↳previously."
```

Then you can use the facts inside your template, like this:

```
{% for host in groups['db_servers'] %}
    {{ hostvars[host]['ansible_eth0']['ipv4']['address'] }}
{% endfor %}
```

### 1.24.16 How do I access a variable name programmatically?

An example may come up where we need to get the ipv4 address of an arbitrary interface, where the interface to be used may be supplied via a role parameter or other input. Variable names can be built by adding strings together, like so:

```
{{ hostvars[inventory_hostname]['ansible_' + which_interface]['ipv4']['address'] }}
```

The trick about going through hostvars is necessary because it's a dictionary of the entire namespace of variables. 'inventory\_hostname' is a magic variable that indicates the current host you are looping over in the host loop.

Also see *dynamic\_variables*.

### 1.24.17 How do I access a group variable?

Technically, you don't, Ansible does not really use groups directly. Groups are label for host selection and a way to bulk assign variables, they are not a first class entity, Ansible only cares about Hosts and Tasks.

That said, you could just access the variable by selecting a host that is part of that group, see *first\_host\_in\_a\_group* below for an example.

### 1.24.18 How do I access a variable of the first host in a group?

What happens if we want the ip address of the first webserver in the webserver group? Well, we can do that too. Note that if we are using dynamic inventory, which host is the 'first' may not be consistent, so you wouldn't want to do this unless your inventory is static and predictable. (If you are using *Ansible Tower*, it will use database order, so this isn't a problem even if you are using cloud based inventory scripts).

Anyway, here's the trick:

```
{{ hostvars[groups['webserver']][0]]['ansible_eth0']['ipv4']['address'] }}
```

Notice how we're pulling out the hostname of the first machine of the webserver group. If you are doing this in a template, you could use the Jinja2 '#set' directive to simplify this, or in a playbook, you could also use set\_fact:

```
- set_fact: headnode={{ groups['webserver']][0]] }}
- debug: msg={{ hostvars[headnode].ansible_eth0.ipv4.address }}
```

Notice how we interchanged the bracket syntax for dots – that can be done anywhere.

### 1.24.19 How do I copy files recursively onto a target host?

The "copy" module has a recursive parameter. However, take a look at the "synchronize" module if you want to do something more efficient for a large number of files. The "synchronize" module wraps rsync. See the module index for info on both of these modules.

### 1.24.20 How do I access shell environment variables?

If you just need to access existing variables ON THE CONTROLLER, use the 'env' lookup plugin. For example, to access the value of the HOME environment variable on the management machine:

```
---
# ...
vars:
  local_home: "{{ lookup('env', 'HOME') }}"
```

For environment variables on the TARGET machines, they are available via facts in the 'ansible\_env' variable:

```
{{ ansible_env.SOME_VARIABLE }}
```

If you need to set environment variables for TASK execution, see the Advanced Playbooks section about environments. There is no set way to set environment variables on your target machines, you can use template/replace/other modules to do so, but the exact files to edit vary depending on your OS and distribution and local configuration.

### 1.24.21 How do I generate crypted passwords for the user module?

The `mkpasswd` utility that is available on most Linux systems is a great option:

```
mkpasswd --method=sha-512
```

If this utility is not installed on your system (e.g. you are using macOS) then you can still easily generate these passwords using Python. First, ensure that the [Passlib](#) password hashing library is installed:

```
pip install passlib
```

Once the library is ready, SHA512 password values can then be generated as follows:

```
python -c "from passlib.hash import sha512_crypt; import getpass; print(sha512_crypt.
↳ using(rounds=5000).hash(getpass.getpass()))"
```

Use the integrated [Hashing filters](#) to generate a hashed version of a password. You shouldn't put plaintext passwords in your playbook or `host_vars`; instead, use [Using Vault in playbooks](#) to encrypt sensitive data.

In OpenBSD, a similar option is available in the base system called `encrypt(1)`:

```
encrypt
```

### 1.24.22 Ansible supports dot notation and array notation for variables. Which notation should I use?

The dot notation comes from Jinja and works fine for variables without special characters. If your variable contains dots (`.`), colons (`:`), or dashes (`-`) it is safer to use the array notation for variables.

```
item[0]['checksum:md5']
item['section']['2.1']
item['region']['Mid-Atlantic']
It is {{ temperature['Celsius']['-3'] }} outside.
```

Also array notation allows for dynamic variable composition, see [dynamic\\_variables](#).

### 1.24.23 Can I get training on Ansible?

Yes! See our [services](#) page for information on our services and training offerings. Email [info@ansible.com](mailto:info@ansible.com) for further details.

We also offer free web-based training classes on a regular basis. See our [webinar](#) page for more info on upcoming webinars.

### 1.24.24 Is there a web interface / REST API / etc?

Yes! Ansible, Inc makes a great product that makes Ansible even more powerful and easy to use. See [Ansible Tower](#).

### 1.24.25 How do I submit a change to the documentation?

Great question! Documentation for Ansible is kept in the main project git repository, and complete instructions for contributing can be found in the docs README [viewable on GitHub](#). Thanks!

### 1.24.26 How do I keep secret data in my playbook?

If you would like to keep secret data in your Ansible content and still share it publicly or keep things in source control, see *Using Vault in playbooks*.

If you have a task that you don't want to show the results or command given to it when using `-v` (verbose) mode, the following task or playbook attribute can be useful:

```
- name: secret task
  shell: /usr/bin/do_something --value={{ secret_value }}
  no_log: True
```

This can be used to keep verbose output but hide sensitive information from others who would otherwise like to be able to see the output.

The `no_log` attribute can also apply to an entire play:

```
- hosts: all
  no_log: True
```

Though this will make the play somewhat difficult to debug. It's recommended that this be applied to single tasks only, once a playbook is completed. Note that the use of the `no_log` attribute does not prevent data from being shown when debugging Ansible itself via the `ANSIBLE_DEBUG` environment variable.

### 1.24.27 When should I use {{ }}? Also, how to interpolate variables or dynamic variable names

A steadfast rule is 'always use `{{ }}` except when `when:`'. Conditionals are always run through Jinja2 as to resolve the expression, so `when:`, `failed_when:` and `changed_when:` are always templated and you should avoid adding `{{ }}`.

In most other cases you should always use the brackets, even if previously you could use variables without specifying (like `loop` or `with_` clauses), as this made it hard to distinguish between an undefined variable and a string.

Another rule is 'moustaches don't stack'. We often see this:

```
{{ somevar_{{other_var}} }}
```

The above DOES NOT WORK as you expect, if you need to use a dynamic variable use the following as appropriate:

```
{{ hostvars[inventory_hostname]['somevar_' + other_var] }}
```

For 'non host vars' you can use the vars lookup plugin:

```
{{ lookup('vars', 'somevar_' + other_var) }}
```

### 1.24.28 Why don't you ship in X format?

Several reasons, in most cases it has to do with maintainability, there are tons of ways to ship software and it is a herculean task to try to support them all. In other cases there are technical issues, for example, for python wheels, our dependencies are not present so there is little to no gain.



### 1.24.29 I don't see my question here

Please see the section below for a link to IRC and the Google Group, where you can ask your question there.

:

*Working With Playbooks* An introduction to playbooks

*Best Practices* Best practices advice

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## 1.25 Glossary

The following is a list (and re-explanation) of term definitions used elsewhere in the Ansible documentation.

Consult the documentation home page for the full documentation and to see the terms in context, but this should be a good resource to check your knowledge of Ansible's components and understand how they fit together. It's something you might wish to read for review or when a term comes up on the mailing list.

**Action** An action is a part of a task that specifies which of the modules to run and which arguments to pass to that module. Each task can have only one action, but it may also have other parameters.

**Ad Hoc** Refers to running Ansible to perform some quick command, using `/usr/bin/ansible`, rather than the *orchestration* language, which is `/usr/bin/ansible-playbook`. An example of an ad hoc command might be rebooting 50 machines in your infrastructure. Anything you can do ad hoc can be accomplished by writing a *playbook* and playbooks can also glue lots of other operations together.

**Async** Refers to a task that is configured to run in the background rather than waiting for completion. If you have a long process that would run longer than the SSH timeout, it would make sense to launch that task in async mode. Async modes can poll for completion every so many seconds or can be configured to "fire and forget", in which case Ansible will not even check on the task again; it will just kick it off and proceed to future steps. Async modes work with both `/usr/bin/ansible` and `/usr/bin/ansible-playbook`.

**Callback Plugin** Refers to some user-written code that can intercept results from Ansible and do something with them. Some supplied examples in the GitHub project perform custom logging, send email, or even play sound effects.

**Check Mode** Refers to running Ansible with the `--check` option, which does not make any changes on the remote systems, but only outputs the changes that might occur if the command ran without this flag. This is analogous to so-called "dry run" modes in other systems, though the user should be warned that this does not take into account unexpected command failures or cascade effects (which is true of similar modes in other systems). Use this to get an idea of what might happen, but do not substitute it for a good staging environment.

**Connection Plugin** By default, Ansible talks to remote machines through pluggable libraries. Ansible supports native OpenSSH (*SSH (Native)*) or a Python implementation called *paramiko*. OpenSSH is preferred if you are using a recent version, and also enables some features like Kerberos and jump hosts. This is covered in the *getting started section*. There are also other connection types like `accelerate` mode, which must be bootstrapped over one of the SSH-based connection types but is very fast, and `local` mode, which acts on the local system. Users can also write their own connection plugins.

**Conditionals** A conditional is an expression that evaluates to true or false that decides whether a given task is executed on a given machine or not. Ansible's conditionals are powered by the 'when' statement, which are discussed in the *Working With Playbooks*.

**Declarative** An approach to achieving a task that uses a description of the final state rather than a description of the sequence of steps necessary to achieve that state. For a real world example, a declarative specification of a

task would be: "put me in California". Depending on your current location, the sequence of steps to get you to California may vary, and if you are already in California, nothing at all needs to be done. Ansible's Resources are declarative; it figures out the steps needed to achieve the final state. It also lets you know whether or not any steps needed to be taken to get to the final state.

**Diff Mode** A `--diff` flag can be passed to Ansible to show what changed on modules that support it. You can combine it with `--check` to get a good 'dry run'. File diffs are normally in unified diff format.

**Executor** A core software component of Ansible that is the power behind `/usr/bin/ansible` directly – and corresponds to the invocation of each task in a *playbook*. The Executor is something Ansible developers may talk about, but it's not really user land vocabulary.

**Facts** Facts are simply things that are discovered about remote nodes. While they can be used in *playbooks* and templates just like variables, facts are things that are inferred, rather than set. Facts are automatically discovered by Ansible when running plays by executing the internal setup module on the remote nodes. You never have to call the setup module explicitly, it just runs, but it can be disabled to save time if it is not needed or you can tell ansible to collect only a subset of the full facts via the `gather_subset :` option. For the convenience of users who are switching from other configuration management systems, the fact module will also pull in facts from the **ohai** and **facter** tools if they are installed. These are fact libraries from Chef and Puppet, respectively. (These may also be disabled via `gather_subset :`)

**Filter Plugin** A filter plugin is something that most users will never need to understand. These allow for the creation of new *Jinja2* filters, which are more or less only of use to people who know what Jinja2 filters are. If you need them, you can learn how to write them in the *API docs section*.

**Forks** Ansible talks to remote nodes in parallel and the level of parallelism can be set either by passing `--forks` or editing the default in a configuration file. The default is a very conservative five (5) forks, though if you have a lot of RAM, you can easily set this to a value like 50 for increased parallelism.

**Gather Facts (Boolean)** *Facts* are mentioned above. Sometimes when running a multi-play *playbook*, it is desirable to have some plays that don't bother with fact computation if they aren't going to need to utilize any of these values. Setting `gather_facts: False` on a playbook allows this implicit fact gathering to be skipped.

**Globbering** Globbing is a way to select lots of hosts based on wildcards, rather than the name of the host specifically, or the name of the group they are in. For instance, it is possible to select `ww*` to match all hosts starting with `www`. This concept is pulled directly from **Func**, one of Michael DeHaan's (an Ansible Founder) earlier projects. In addition to basic globbing, various set operations are also possible, such as 'hosts in this group and not in another group', and so on.

**Group** A group consists of several hosts assigned to a pool that can be conveniently targeted together, as well as given variables that they share in common.

**Group Vars** The `group_vars/` files are files that live in a directory alongside an inventory file, with an optional filename named after each group. This is a convenient place to put variables that are provided to a given group, especially complex data structures, so that these variables do not have to be embedded in the *inventory* file or *playbook*.

**Handlers** Handlers are just like regular tasks in an Ansible *playbook* (see *Tasks*) but are only run if the Task contains a `notify` directive and also indicates that it changed something. For example, if a config file is changed, then the task referencing the config file templating operation may notify a service restart handler. This means services can be bounced only if they need to be restarted. Handlers can be used for things other than service restarts, but service restarts are the most common usage.

**Host** A host is simply a remote machine that Ansible manages. They can have individual variables assigned to them, and can also be organized in groups. All hosts have a name they can be reached at (which is either an IP address or a domain name) and, optionally, a port number, if they are not to be accessed on the default SSH port.

**Host Specifier** Each *Play* in Ansible maps a series of *tasks* (which define the role, purpose, or orders of a system) to a set of systems.

This `hosts:` directive in each play is often called the hosts specifier.

It may select one system, many systems, one or more groups, or even some hosts that are in one group and explicitly not in another.

**Host Vars** Just like *Group Vars*, a directory alongside the inventory file named `host_vars/` can contain a file named after each hostname in the inventory file, in *YAML* format. This provides a convenient place to assign variables to the host without having to embed them in the *inventory* file. The Host Vars file can also be used to define complex data structures that can't be represented in the inventory file.

**Idempotency** An operation is idempotent if the result of performing it once is exactly the same as the result of performing it repeatedly without any intervening actions.

**Includes** The idea that *playbook* files (which are nothing more than lists of *plays*) can include other lists of plays, and task lists can externalize lists of *tasks* in other files, and similarly with *handlers*. Includes can be parameterized, which means that the loaded file can pass variables. For instance, an included play for setting up a WordPress blog may take a parameter called `user` and that play could be included more than once to create a blog for both `alice` and `bob`.

**Inventory** A file (by default, Ansible uses a simple INI format) that describes *Hosts* and *Groups* in Ansible. Inventory can also be provided via an *Inventory Script* (sometimes called an "External Inventory Script").

**Inventory Script** A very simple program (or a complicated one) that looks up *hosts*, *group* membership for hosts, and variable information from an external resource – whether that be a SQL database, a CMDB solution, or something like LDAP. This concept was adapted from Puppet (where it is called an "External Nodes Classifier") and works more or less exactly the same way.

**Jinja2** Jinja2 is the preferred templating language of Ansible's template module. It is a very simple Python template language that is generally readable and easy to write.

**JSON** Ansible uses JSON for return data from remote modules. This allows modules to be written in any language, not just Python.

**Lazy Evaluation** In general, Ansible evaluates any variables in *playbook* content at the last possible second, which means that if you define a data structure that data structure itself can define variable values within it, and everything "just works" as you would expect. This also means variable strings can include other variables inside of those strings.

**Library** A collection of modules made available to `/usr/bin/ansible` or an Ansible *playbook*.

**Limit Groups** By passing `--limit somegroup` to `ansible` or `ansible-playbook`, the commands can be limited to a subset of *hosts*. For instance, this can be used to run a *playbook* that normally targets an entire set of servers to one particular server.

**Local Action** A `local_action` directive in a *playbook* targeting remote machines means that the given step will actually occur on the local machine, but that the variable `{{ ansible_hostname }}` can be passed in to reference the remote hostname being referred to in that step. This can be used to trigger, for example, an `rsync` operation.

**Local Connection** By using `connection: local` in a *playbook*, or passing `-c local` to `/usr/bin/ansible`, this indicates that we are managing the local host and not a remote machine.

**Lookup Plugin** A lookup plugin is a way to get data into Ansible from the outside world. Lookup plugins are an extension of Jinja2 and can be accessed in templates, e.g., `{{ lookup('file', '/path/to/file') }}`. These are how such things as `with_items`, are implemented. There are also lookup plugins like `file` which loads data from a file and ones for querying environment variables, DNS text records, or key value stores.

**Loops** Generally, Ansible is not a programming language. It prefers to be more declarative, though various constructs like `loop` allow a particular task to be repeated for multiple items in a list. Certain modules, like `yum` and `apt`, actually take lists directly, and can install all packages given in those lists within a single transaction, dramatically speeding up total time to configuration, so they can be used without loops.

**Modules** Modules are the units of work that Ansible ships out to remote machines. Modules are kicked off by either `/usr/bin/ansible` or `/usr/bin/ansible-playbook` (where multiple tasks use lots of different modules in conjunction). Modules can be implemented in any language, including Perl, Bash, or Ruby – but can leverage some useful communal library code if written in Python. Modules just have to return *JSON*. Once modules are executed on remote machines, they are removed, so no long running daemons are used. Ansible refers to the collection of available modules as a *library*.

**Multi-Tier** The concept that IT systems are not managed one system at a time, but by interactions between multiple systems and groups of systems in well defined orders. For instance, a web server may need to be updated before a database server and pieces on the web server may need to be updated after *THAT* database server and various load balancers and monitoring servers may need to be contacted. Ansible models entire IT topologies and workflows rather than looking at configuration from a "one system at a time" perspective.

**Notify** The act of a *task* registering a change event and informing a *handler* task that another *action* needs to be run at the end of the *play*. If a handler is notified by multiple tasks, it will still be run only once. Handlers are run in the order they are listed, not in the order that they are notified.

**Orchestration** Many software automation systems use this word to mean different things. Ansible uses it as a conductor would conduct an orchestra. A datacenter or cloud architecture is full of many systems, playing many parts – web servers, database servers, maybe load balancers, monitoring systems, continuous integration systems, etc. In performing any process, it is necessary to touch systems in particular orders, often to simulate rolling updates or to deploy software correctly. Some system may perform some steps, then others, then previous systems already processed may need to perform more steps. Along the way, emails may need to be sent or web services contacted. Ansible orchestration is all about modeling that kind of process.

**paramiko** By default, Ansible manages machines over SSH. The library that Ansible uses by default to do this is a Python-powered library called paramiko. The paramiko library is generally fast and easy to manage, though users desiring Kerberos or Jump Host support may wish to switch to a native SSH binary such as OpenSSH by specifying the connection type in their *playbooks*, or using the `-c ssh` flag.

**Playbooks** Playbooks are the language by which Ansible orchestrates, configures, administers, or deploys systems. They are called playbooks partially because it's a sports analogy, and it's supposed to be fun using them. They aren't workbooks :)

**Plays** A *playbook* is a list of plays. A play is minimally a mapping between a set of *hosts* selected by a host specifier (usually chosen by *groups* but sometimes by hostname *globs*) and the *tasks* which run on those hosts to define the role that those systems will perform. There can be one or many plays in a playbook.

**Pull Mode** By default, Ansible runs in *push mode*, which allows it very fine-grained control over when it talks to each system. Pull mode is provided for when you would rather have nodes check in every N minutes on a particular schedule. It uses a program called **ansible-pull** and can also be set up (or reconfigured) using a push-mode *playbook*. Most Ansible users use push mode, but pull mode is included for variety and the sake of having choices.

**ansible-pull** works by checking configuration orders out of git on a crontab and then managing the machine locally, using the *local connection* plugin.

**Push Mode** Push mode is the default mode of Ansible. In fact, it's not really a mode at all – it's just how Ansible works when you aren't thinking about it. Push mode allows Ansible to be fine-grained and conduct nodes through complex orchestration processes without waiting for them to check in.

**Register Variable** The result of running any *task* in Ansible can be stored in a variable for use in a template or a conditional statement. The keyword used to define the variable is called `register`, taking its name from the idea of registers in assembly programming (though Ansible will never feel like assembly programming). There are an infinite number of variable names you can use for registration.

**Resource Model** Ansible modules work in terms of resources. For instance, the file module will select a particular file and ensure that the attributes of that resource match a particular model. As an example, we might wish to change the owner of `/etc/motd` to `root` if it is not already set to `root`, or set its mode to `0644` if it is

not already set to 0644. The resource models are *idempotent* meaning change commands are not run unless needed, and Ansible will bring the system back to a desired state regardless of the actual state – rather than you having to tell it how to get to the state.

**Roles** Roles are units of organization in Ansible. Assigning a role to a group of *hosts* (or a set of *groups*, or *host patterns*, etc.) implies that they should implement a specific behavior. A role may include applying certain variable values, certain *tasks*, and certain *handlers* – or just one or more of these things. Because of the file structure associated with a role, roles become redistributable units that allow you to share behavior among *playbooks* – or even with other users.

**Rolling Update** The act of addressing a number of nodes in a group N at a time to avoid updating them all at once and bringing the system offline. For instance, in a web topology of 500 nodes handling very large volume, it may be reasonable to update 10 or 20 machines at a time, moving on to the next 10 or 20 when done. The `serial:` keyword in an Ansible *playbooks* control the size of the rolling update pool. The default is to address the batch size all at once, so this is something that you must opt-in to. OS configuration (such as making sure config files are correct) does not typically have to use the rolling update model, but can do so if desired.

**Serial**

:

*Rolling Update*

**Sudo** Ansible does not require root logins, and since it's daemonless, definitely does not require root level daemons (which can be a security concern in sensitive environments). Ansible can log in and perform many operations wrapped in a sudo command, and can work with both password-less and password-based sudo. Some operations that don't normally work with sudo (like scp file transfer) can be achieved with Ansible's copy, template, and fetch modules while running in sudo mode.

**SSH (Native)** Native OpenSSH as an Ansible transport is specified with `-c ssh` (or a config file, or a directive in the *playbook*) and can be useful if wanting to login via Kerberized SSH or using SSH jump hosts, etc. In 1.2.1, `ssh` will be used by default if the OpenSSH binary on the control machine is sufficiently new. Previously, Ansible selected `paramiko` as a default. Using a client that supports `ControlMaster` and `ControlPersist` is recommended for maximum performance – if you don't have that and don't need Kerberos, jump hosts, or other features, `paramiko` is a good choice. Ansible will warn you if it doesn't detect `ControlMaster/ControlPersist` capability.

**Tags** Ansible allows tagging resources in a *playbook* with arbitrary keywords, and then running only the parts of the playbook that correspond to those keywords. For instance, it is possible to have an entire OS configuration, and have certain steps labeled `ntp`, and then run just the `ntp` steps to reconfigure the time server information on a remote host.

**Task** *Playbooks* exist to run tasks. Tasks combine an *action* (a module and its arguments) with a name and optionally some other keywords (like *looping directives*). *Handlers* are also tasks, but they are a special kind of task that do not run unless they are notified by name when a task reports an underlying change on a remote system.

**Tasks** A list of *Task*.

**Templates** Ansible can easily transfer files to remote systems but often it is desirable to substitute variables in other files. Variables may come from the *inventory* file, *Host Vars*, *Group Vars*, or *Facts*. Templates use the *Jinja2* template engine and can also include logical constructs like loops and if statements.

**Transport** Ansible uses `:term:Connection Plugins` to define types of available transports. These are simply how Ansible will reach out to managed systems. Transports included are *paramiko*, *ssh* (using OpenSSH), and *local*.

**When** An optional conditional statement attached to a *task* that is used to determine if the task should run or not. If the expression following the `when:` keyword evaluates to false, the task will be ignored.

**Vars (Variables)** As opposed to *Facts*, variables are names of values (they can be simple scalar values – integers, booleans, strings) or complex ones (dictionaries/hashes, lists) that can be used in templates and *playbooks*.

They are declared things, not things that are inferred from the remote system's current state or nature (which is what Facts are).

**YAML** Ansible does not want to force people to write programming language code to automate infrastructure, so Ansible uses YAML to define *playbook* configuration languages and also variable files. YAML is nice because it has a minimum of syntax and is very clean and easy for people to skim. It is a good data format for configuration files and humans, but also machine readable. Ansible's usage of YAML stemmed from Michael DeHaan's first use of it inside of Cobbler around 2006. YAML is fairly popular in the dynamic language community and the format has libraries available for serialization in many languages (Python, Perl, Ruby, etc.).

:

*Frequently Asked Questions* Frequently asked questions

*Working With Playbooks* An introduction to playbooks

*Best Practices* Best practices advice

**User Mailing List** Have a question? Stop by the google group!

**irc.freenode.net** #ansible IRC chat channel

## 1.26 Ansible Tower

*Ansible Tower* (formerly 'AWX') is a web-based solution that makes Ansible even more easy to use for IT teams of all kinds. It's designed to be the hub for all of your automation tasks.

Tower allows you to control access to who can access what, even allowing sharing of SSH credentials without someone being able to transfer those credentials. Inventory can be graphically managed or synced with a wide variety of cloud sources. It logs all of your jobs, integrates well with LDAP, and has an amazing browsable REST API. Command line tools are available for easy integration with Jenkins as well. Provisioning callbacks provide great support for autoscaling topologies.

Find out more about Tower features and how to download it on the [Ansible Tower webpage](#). Tower is free for usage for up to 10 nodes, and comes bundled with amazing support from Ansible, Inc. As you would expect, Tower is installed using Ansible playbooks!

## 1.27 Ansible Roadmap

The Ansible team develops a roadmap for each major Ansible release. The latest roadmap shows current work; older roadmaps provide a history of the project.

### 1.27.1 Ansible 2.1

**Target: April**

#### Topics

- *Ansible 2.1*
  - *Windows*
  - *Network*



- *VMware*
- *Azure*
- *Docker*
- *Cloud*
- *Ansiballz*
- *Diff-support*
- *Other*
- *Community*

## Windows

- General
  - Figuring out privilege escalation (runas w/ username/password)
  - Implement kerberos encryption over http
  - pywinrm conversion to requests (Some mess here on pywinrm/requests. will need docs etc.)
  - NTLM support
- Modules
  - Finish cleaning up tests and support for post-beta release
  - Strict mode cleanup (one module in core)
  - Domain user/group management
  - Finish win\_host and win\_rm in the domain/workgroup modules.
    - \* Close 2 existing PRs (These were deemed insufficient)
  - Replicate python module API in PS/C# (deprecate hodgepodge of stuff from module\_utils/powershell.ps1)

## Network

- Cisco modules (ios, iosxr, nxos, iosxe)
- Arista modules (eos)
- Juniper modules (junos)
- OpenSwitch
- Cumulus
- Dell (os10) - At risk
- Netconf shared module
- Hooks for supporting Tower credentials

### VMware

This one is a little at risk due to staffing. We're investigating some community maintainers and shifting some people at Ansible around, but it is a VERY high priority.

- vsphere\_guest brought to parity with other vmware modules (vs Viasat and 'whereismyjetpack' provided modules)
- VMware modules moved to official pyvmomi bindings
- VMware inventory script updates for pyvmomi, adding tagging support

### Azure

This is on hold until Microsoft swaps out the code generator on the Azure Python SDK, which may introduce breaking changes. We have basic modules working against all of these resources at this time. Could ship it against current SDK, but may break. Or should the version be pinned?) - Minimal Azure coverage using new ARM api - Resource Group - Virtual Network - Subnet - Public IP - Network Interface - Storage Account - Security Group - Virtual Machine - Update of inventory script to use new API, adding tagging support

### Docker

- Start Docker module refactor
- Update to match current docker CLI capabilities
- Docker exec support

### Cloud

Upgrade other cloud modules or work with community maintainers to upgrade. (In order)

- AWS (Community maintainers)
- Openstack (Community maintainers)
- Google (Google/Community)
- Digital Ocean (Community)

### Ansiballz

Renamed from Ziploader

- Write code to create the zipfile that gets passed across the wire to be run on the remote python
- Port most of the functionality in module\_utils to be usage in ansiballz instead
- Port a few essential modules to use ansiballz instead of module-replacer as proof of concept
- New modules will be able to use ansiballz. Old modules will need to be ported in future releases (Some modules will not need porting but others will)
- Better testing of modules, caching of modules clientside(Have not yet arrived at an architecture for this that we like), better code sharing between ansible/ansible and modules
- ansiballz is a helpful building block for: python3 porting(high priority), better code sharing between modules(medium priority)



- ansiballz is a good idea before: enabling users to have custom module\_utils directories

## Diff-support

Expand module diff support (already in progress in devel)

- Framework done. Need to add to modules, test etc.
- Coordinate with community to update their modules

## Other

Things being kicking down the road that we said we'd do

- NOT remerging core with ansible/ansible this release cycle

## Community

- Define the process/ETA for reviewing PR's from community
- Publish better docs and how-tos for submitting code/features/fixes

## 1.27.2 Ansible 2.2

**Target: September 2016**

### Topics

- *Ansible 2.2*
  - *Docker*
  - *Extras split from Core*
  - *Tweaks/Fixes*
  - *AWS*
  - *Google*
  - *OpenStack*
  - *Azure load balancer*
  - *VMware*
  - *Windows*
  - *Network*
  - *Role revamp*
  - *Vault*
  - *Python3*
  - *Infrastructure Buildout and Changes*

### Docker

Lead by Chris Houseknecht

- Docker\_network: **done**
- Docker\_volume: Not in this release
- Docker\_file: Not in this release.
- Openshift: oso\_deployment, oso\_route, oso\_service, oso\_login (...and possibly others. These are modules being developed to support [ansible-container](#).): Deferred for later release
- Kubernetes: kube\_deployment, kube\_service, kube\_login (...and possibly others. These too are modules being developed to support [ansible-container](#)): Deferred for later release

### Extras split from Core

Lead by Jason M and Jimi-c (Targeting 2.2, could move into 2.3).

Targeted towards the 2.2 release or shortly after, we are planning on splitting Extras out of the "Ansible Core" project. That means that modules that are shipped with Ansible by default are **only** the modules in `ansible-modules-core`. Ansible extras will become a separate project, managed by the community standard. Over the next few months we're going to have a lot of work to do on getting all of the modules in the right places for this to work.

- Create proposal (Jason or Jimi)
- Review modules for correct location (extras v core)
- Extras is a completely different package (does not install with ansible)
- Library dependencies
- Decide and implement release schedules between Ansible Core and Extras to ensure compatibility and longevity for modules and versions of Ansible.

### Tweaks/Fixes

- Connection handling stuff. (Toshio K. and Brian C.): This is a stretch goal for 2.2. **This work got pushed out**
  - Change connection polling to avoid resource limitations, see <https://github.com/ansible/ansible/issues/14143>
  - <https://docs.python.org/3/library/selectors.html#module-selectors>
  - Code: [https://github.com/kail1/ansible/blob/fix/select\\_fd\\_out\\_of\\_range\\_wip/lib/ansible/plugins/connection/ssh.py](https://github.com/kail1/ansible/blob/fix/select_fd_out_of_range_wip/lib/ansible/plugins/connection/ssh.py)

### AWS

Lead by Ryan Brown

- Pagination for all AWS modules (generic pagination exists, but isn't used everywhere) (bumped to 2.3)
- Refactoring `ec2.py` to be more digestible (bumped to 2.3)
- Fix inconsistencies with different authentication methods (STS, environment creds, `~/.aws/credentials`) (done)
- AWS Lambda modules (`lambda_execute` done, others pending)
- Ryan Brown and Robyn Bergeron work on bug/PR triage to reduce backlog (reduced - continuing to work on it)

## Google

Lead by Ryan Brown and Tom Melendez

- Add support for Google Cloud DNS
- Add support for Google Cloud managed instance groups (done)
- Support restoring instances from snapshots
- Improved handling of scratch disks on instances (done)

## OpenStack

Lead by Ryan Brown

Stretch goal for this release

- Ryan with some help from David Shrewsbury (Zuul/Openstack at RedHat).
- Support Heat stack resources (done)
- Support LBaaS load balancers

## Azure load balancer

- Feature parity for AWS ELB (Stretch Goal)

## VMware

Lead by Brian, Jtanner

- *module/inventory script: port to pyvmomi (jtanner, bcoca)* **done:** <https://github.com/ansible/ansible/pull/15967>
- *inventory script: allow filtering ala ec2 (jtanner) (undergoing PR process)* **done:** <https://github.com/ansible/ansible/pull/15967>
- vsphere: feature parity with whereismyjetpack and viasat modules

## Windows

Lead by Matt D

- Feature parity
  - PS module API (mirror Python module API where appropriate). Note: We don't necessarily like the current python module API (AnsibleModule is a huge class with many unrelated utility functions. Maybe we should redesign both at the same time?) (bumped to 2.3+ due to "moving target" uncertainty)
  - Environment keyword support (done)
  - win\_shell/win\_command (done)
  - Async support (done)
  - (stretch goal) Pipelining (bumped to 2.3+)
- Windows-specific enhancements
  - Multiple Kerberos credential support (done)

- Server 2016 testing/fixes (done, awaiting next TP/RTM)
- (stretch goal) Nano Server connection + module\_utils working (bumped to 2.3)
- (stretch goal) Encrypted kerberos support in pywinrm (bumped to 2.3)

### Network

Lead by Nate C, Peter S

- **Done:** Unify NetworkModules (module\_utils/network.py) as much as possible
- **Done:** Add support for config diff and replace on supported platforms (2 weeks)
- **Done:** Support for VyOS network operating system
- **Done:** Add support for RestConf for IOS/XE
- **Done:** Support for Dell Networking OS10
- **Done:** Add support for Nokia SR OS modules
- **Done:** Network facts modules (dellos, eos, ios, iosxr, junos, nxos, openswitch, vyos)
- **Deferred:** Network facts modules (cumulus, netvisor, sros)
- **Deferred:** Add support for NetConf for IOS/XE
- **Deferred:** (stretch goal) Quagga modules
- **Deferred:** (stretch goal) Bird modules
- **Deferred:** (stretch goal) GoBGP modules

### Role revamp

- Implement 'role revamp' proposal to give users more control on role/task execution (Brian)
  - [https://github.com/ansible/proposals/blob/master/roles\\_revamp.md](https://github.com/ansible/proposals/blob/master/roles_revamp.md)

### Vault

Lead by Jtanner, Adrian

- *Extend 'transparent vault file usage' to other action plugins other than 'copy'* (<https://github.com/ansible/ansible/issues/7298>) **done:** <https://github.com/ansible/ansible/pull/16957>
- Add 'per variable' vault support (!vault YAML directive, existing PR already) <https://github.com/ansible/ansible/issues/13287> <https://github.com/ansible/ansible/issues/14721>
- Add vault/unvault filters <https://github.com/ansible/ansible/issues/12087> (deferred to 2.3)
- Add vault support to lookups (likely deferred to 2.3 or until lookup plugins are revamped)
- Allow for multiple vault secrets <https://github.com/ansible/ansible/issues/13243>
- Config option to turn 'unvaulting' failures into warnings <https://github.com/ansible/ansible/issues/13244>

## Python3

Lead by Toshio

A note here from Jason M: Getting to complete, tested Python 3 is both a critical task and one that has so much work and so many moving parts that we don't expect this to be complete by the 2.2 release. Toshio will lead this overall effort.

- Motivation: - Ubuntu LTS (16.04) already ships without python2. RHEL8 is coming which is also expected to be python3 based. These considerations make this high priority. - Ansible users are getting restless: <https://groups.google.com/forum/#!topic/ansible-project/DUKzTho3OCI> - This is probably going to take multiple releases to complete; need to get started now
- Baselines: - We're targeting Python-3.5 and above.
- Goals for 2.2:
  - Tech preview level of support
  - Controller-side code can run on Python3
  - Update: Essential features have been shown to work on Python3. Currently all unittests and all but three integration tests are passing on Python3. Code has not been line-by-line audited so bugs remain but can be treated as bugs, not as massive, invasive new features.
  - Almost all of our deps have been ported:
    - \* The base deps in setup.py are ported: ['paramiko', 'jinja2', 'PyYAML', 'setuptools', 'pypcrypto']
    - \* python-six from the rpm spec file has been ported
    - \* Python-keyczar from the rpm spec file is not.
    - \* Strategy: removing keyczar when we drop accelerate for 2.3. Print deprecation in 2.1.
  - Module\_utils ported to dual python3/python2(2.4 for much of it, python2.6 for specific things) **Mostly done:** Also not line-by-line audited but the unittests and integration tests do show that the most use functionality is working.
  - Add module\_utils files to help port
    - \* Update: copy of the six library (v1.4.1 for python2.4 compat) and unicode helpers are here (ansible.module\_utils.\_text.{to\_bytes,to\_text,to\_native})
  - A few basic modules ported to python3
    - \* Stat module best example module since it's essential.
    - \* Update:
      - A handful of modules like stat have been line-by-line ported. They should work reliably with few python3-specific bugs. All but three integration tests pass which means that most essential modules are working to some extent on Python3.
      - The three failing tests are: service, hg, and uri.
      - **Note, large swaths of the modules are not tested. The status of** these is unknown
  - All code should compile under Python3. - lib/ansible/\* and all modules now compile under Python-3.5
  - Side work to do: - Figure out best ways to run unit-tests on modules. Start unit-testing modules. This is going to become important so we don't regress python3 or python2.4 support in modules (Going to largely punt on this for 2.2. Matt Clay is working on building us a testing foundation for the first half of 2.2 development so we'll re-evaluate towards the middle of the dev cycle). - More unit tests of

module\_utils - More integration tests. Currently integration tests are the best way to test ansible modules so we have to rely on those.

– Goals for 2.3:

- \* Bugfixing, bugfixing, bugfixing. We need community members to test, submit bugs, and add new unit and integration tests. I'll have some time allocated both to review any Python3 bugfixes that they submit and to work on bug reports without PRs. The overall goal is to make the things that people do in production with Ansible work on Python 3.

### Infrastructure Buildout and Changes

Lead by Matt Clay

Another note from Jason M: A lot of this work is to ease the burden of CI, CI performance, increase our testing coverage and all of that sort of thing. It's not necessarily feature work, but it's **critical** to growing our product and our ability to get community changes in more securely and quickly.

- **CI Performance** Reduce time spent waiting on CI for PRs. Combination of optimizing existing Travis setup and offloading work to other services. Will be impacted by available budget.

**Done:** Most tests have been migrated from Travis to Shippable.

- **Core Module Test Organization** Relocate core module tests to ansible-modules-core to encourage inclusion of tests in core module PRs.

**Deferred:** Relocation of core module tests has been deferred due to proposed changes in [modules management](#).

- **Documentation** Expand documentation on setting up a development and test environment, as well as writing tests. The goal is to ease development for new contributors and encourage more testing, particularly with module contributions.

- **Test Coverage**

- Expand test coverage, particularly for CI. Being testing, this is open ended. Will be impacted by available budget.

**Done:** Module PRs now run integration tests for the module(s) being changed.

- Python 3 - Run integration tests using Python 3 on CI with tagging for those which should pass, so we can track progress and detect regressions.

**Done:** Integration tests now run on Shippable using a Ubuntu 16.04 docker image with only Python 3 installed.

- Windows - Create framework for running Windows integration tests, ideally both locally and on CI.

**Done:** Windows integration tests now run on Shippable.

- FreeBSD - Include FreeBSD in CI coverage. Not originally on the roadmap, this is an intermediary step for CI coverage for macOS.

**Done:** FreeBSD integration tests now run on Shippable.

- macOS - Include macOS in CI coverage.

**Done:** macOS integration tests now run on Shippable.

### 1.27.3 Ansible 2.3

**Target: Mid April 2017**

## Topics

- *Ansible 2.3*
  - *General Comments from the Core Team*
  - *Repo Merge*
  - *Metadata*
  - *Documentation*
  - *Windows*
  - *Azure*
  - *Networking*
  - *Python3*
  - *Testing and CI*
  - *Amazon*
  - *Plugin Loader*
  - *ansible-ssh*

## General Comments from the Core Team

- The 2.3 Ansible Core is just a little different than the past two major releases we've done. In addition to feature work, we're using part of the time for this release to reduce some of our backlog in other areas than pure development.
- *Administration:* Clean up our GitHub repos and move to one repo so that contributions, tickets, submissions, etc are centralized and easier for both the community and the Core Team to manage.
- *Metadata:* Move to a Metadata based system for modules. This has been discussed here: <https://github.com/ansible/proposals/blob/master/modules-management.md>
- *Documentation:* We're aware that Docs have issues. Scott Butler, aka Dharmabumstead will be leading the charge on how he and we as a community can clean them up.
- *Backlog & Stability:* We're spending some of the cycles for 2.3 trying to reduce our ticket/PR backlog, and clean up some particular areas of the project that the community has expressed particular frustrations about.
- *Python 3:* The community and Toshio have done TONS of work getting Python 3 working. Still more to go...
- *Features:* We still have some cool stuff coming. Check it out below. For people on the Networking side of the world, the Persistent Connection Manager will be a *huge* feature and performance gain.

## Repo Merge

- Script that a submitter can run to migrate their PR (**done**)
- Script that a committer can run to fork a PR and then merge to ansible/ansible (**mostly done**)
- Move all the issues (remove old ones that can be removed) (**done**)
- Enhance ansibullbot to accommodate the changes (jctanner) (**in progress, going well**)

### Metadata

- Add metadata to the modules we ship (**done**)
- Write code to use metadata in docs (**done**)
- If needed for python2/3 write code to use metadata in module\_common or pluginloader (**not needed**)

### Documentation

- Update developing\_modules (**in progress, will continue in 2.4**)
- Set up rst skeleton for module\_utils docs.
- Plugin development docs
- Speed up *make webdocs* <https://github.com/ansible/ansible/issues/17406> (**done**)

### Windows

Lead by nitzmahone

- **Platform**
  - Pipelining support (**done**)
  - Become support (**done/experimental**)
  - Integrated kerberos ticket management (via ansible\_user/ansible\_password) (**done**)
  - Switch PS input encoding to BOM-less UTF8 (**done**)
  - Server 2016 support/testing (now RTM'd) (**partial**)
  - Modularize Windows module\_utils (allow N files) (**partial**)
  - Declarative argspec for PS / .NET (**bumped to 2.4**)
  - Kerberos encryption (via notting, pywinrm/requests\_kerberos/pykerberos) (**in progress, available in py-winrm post 2.3 release**)
  - Fix plugin-specific connection var lookup/delegation (either registered explicitly by plugins or ansible\_(plugin)\_\*) (**bumped to 2.4**)
- **Modules**
  - win\_domain module (**done**)
  - win\_domain\_membership module (**done**)
  - win\_domain\_controller module (**done**)
  - win\_dns\_client module (**done**)
  - win\_wait\_for module
  - win\_disk\_image module (**done**)
  - Updates to win\_chocolatey, adopt to core (stretch) (**bump to 2.4**)
  - Updates/rewrite to win\_unzip, adopt to core (stretch) (**bump to 2.4**)
  - Updates to win\_updates, adopt to core (stretch) (**bump to 2.4**)
  - Updates to win\_package, adopt to core (+ deprecate win\_msi) (stretch) (**bump to 2.4**)



## Azure

Lead by nitzmahone, mattclay

- Ensure Azure SDK rc6/RTM work (**done**)
- Move tests from ansible/azure\_rm repo to ansible/ansible (**bump to 2.4, no CI resources**)
- Update/enhance tests (**bump to 2.4, no CI resources**)
- Expose endpoint overrides (support AzureChinaCloud, Azure Stack) (**bump to 2.4**)
- Get Azure tests running in CI (stretch, depends on availability of sponsored account) (**bump to 2.4, no CI resources**)
- azure\_rm\_loadbalancer module (stretch) (**bump to 2.4**)

## Networking

- Code stability and tidy up (**done**)
- Extend testing (**done**)
- User facing documentation
- Persistent connection manager (**done**)
- Netconf/YANG implementation (only feature) (**done**)
- Deferred from 2.2: Network facts modules (sros)

## Python3

- For 2.3:
  - We want all tests to pass
    - \* Just the mercurial tests left because we haven't created an image with both python2 and python3 to test it on yet.
    - \* Check by doing `grep skip/python3 test/integration/targets/*/aliases`
  - If users report bugs on python3, these should be fixed and will prioritize our work on porting other modules.
- Still have to solve the python3-only and python2-only modules. Thinking of doing this via metadata. Will mean we have to use metadata at the module\_common level. Will also mean we don't support py2-only or py3-only old style python modules.
- Note: Most of the currently tested ansible features now run. But there's still a lot of code that's untested.

## Testing and CI

Lead by mattclay

- *Static Code Analysis*: Create custom pylint extensions to automate detection of common Ansible specific issues reported during code review. Automate feedback on PRs for new code only to avoid noise from existing code which does not pass.

**Ongoing:** Some static code analysis is now part of the CI process:

- pep8 is now being run by CI, although not all PEP 8 rules are being enforced.

- pylint is now being run by CI, but currently only on the ansible-test portion of codebase.

- *Test Reliability:* Eliminate transient test failures by fixing unreliable tests. Reduce network dependencies by moving network resources into httptester.

**Ongoing:** Many of the frequent sources of test instability have been resolved. However, more work still remains. Some new issues have also appeared, which are currently being worked on.

- *Enable Remaining Tests:* Implement fixes for macOS, FreeBSD and Python 3 to enable the remaining blacklisted tests for CI.

**Ongoing:** More tests have been enabled for macOS, FreeBSD and Python 3. However, work still remains to enable more tests.

- *Windows Server 2016:* Add Windows Server 2016 to CI when official AMIs become available.

**Delayed:** Integration tests pass on Windows Server 2016. However, due to intermittent WinRM issues, the tests have been disabled.

Once the issues with WinRM have been resolved, the tests will be re-enabled.

- *Repository Consolidation:* Update CI to maintain and improve upon existing functionality after repository consolidation.

**Done:** A new test runner, ansible-test, has been deployed to manage CI jobs on Shippable.

Tests executed on PRs are based on the changes made in the PR, for example:

- Changes to a module will only run tests appropriate for that module.
- Changes to Windows modules or the Windows connection plugin run tests on Windows.
- Changes to network modules run tests on the appropriate virtual network device (currently supporting VyOS and IOS).

Tests executed on merges are based on changes since the last successful merge test.

## Amazon

Lead by ryansb

- Improve ec2.py integration tests (**partial, more to do in 2.4**)
- ELB version 2 (**pushed - needs\_revision**) [PR](#)
- CloudFormation YAML, cross-stack reference, and roles support (**done**)
- ECS module refactor (**done**)
- AWS module unit testing w/ placebo (boto3 only) (**pushed 2.4**)

## Plugin Loader

- Add module\_utils to the plugin loader (feature) [done]
- Split plugin loader: Plugin\_search, plugin\_loader (modules only use first) [pushed to 2.4]

**ansible-ssh**

- Add a 'ansible-ssh' convenience and debugging tool (will slip to 2.4)
- Tool to invoke an interactive ssh to a host with the same args/env/config that ansible would.
- There are at least three external versions
  - <https://github.com/2ndQuadrant/ansible-ssh>
  - <https://github.com/haad/ansible-ssh>
  - <https://github.com/mlvnd/ansible-ssh>

**1.27.4 Ansible 2.4**

**Core Engine Freeze and Module Freeze: 15 August 2017**

**Core and Curated Module Freeze: 15 August 2017**

**Community Module Freeze: 29 August 2017**

**Release Candidate 1 will be 06 September, 2017**

**Target: Mid-September 2017**

**Topics**

- *Ansible 2.4*
  - *Administrivia and Process*
  - *Python 2.4 and 2.5 support discontinuation*
  - *Python 3*
  - *Ansible-Config*
  - *Inventory*
  - *Facts*
  - *PluginLoader*
  - *Static Loop Keyword*
  - *Vault*
  - *Globalize Callbacks*
  - *Plugins*
  - *Group Priorities*
  - *Runtime Check on Modules for Blacklisting*
  - *Disambiguate Includes*
  - *Windows*
  - *AWS*
  - *Azure*
  - *Google Cloud Platform*

- *Network Roadmap*
- *Contributor Quality of Life*

### Administrivia and Process

- Starting with 2.4, all items that are deprecated will be removed in 4 major releases unless otherwise stated.
  - For example: A module that is deprecated in 2.4 will be removed in 2.8

### Python 2.4 and 2.5 support discontinuation

- Ansible will not support Python 2.4 nor 2.5 on the target hosts anymore. Going forward, Python 2.6+ will be required on targets, as already is the case on the controller.

### Python 3

- Ansible Core Engine and Core modules will be tested on Python 3
  - All Core modules now have at least a smoketest integration test. Additional coverage is welcomed to find more bugs and prevent regressions.
- Communicate with Linux distros to provide Ansible running on Python 3
  - Python3 based Ansible packages are now available to run on Fedora Linux

### Ansible-Config

- Proposal found in ansible/proposals issue [#35](#).
- Initial PR of code found in ansible/ansible PR [#12797](#). **(done)**
- Per plugin configuration (depends on plugin docs below). **(WIP)**
- New yaml format for config **(possibly pushed to future roadmap)**
- Extend the ability of the current config system by adding an `ansible-config` command and add the following:
  - Dump existing config settings **(working, fine tuning)**
  - Update / write a config entry **(pushed to future roadmap)**
  - Show available options (ini entry, yaml, env var, etc) **(working, fine tuning)**

### Inventory

**(done, needs docs)** - Proposal found in ansible/proposals issue [#41](#). - Current inventory is overly complex, non modular and mostly still a legacy from inception.

## Facts

- Configurable list of 'fact modules' for `gather_facts` **(done)**
- Fact gathering policy finer grained **(done)**
- Make `setup.py/facts` more pluggable **(done)**
- Improve testing of `setup.py/facts.py` **(done)**
- Namespacing fact variables (via a config option) implemented in [ansible/ansible PR #18445](#). **(done)** Proposal found in [ansible/proposals issue #17](#).

## PluginLoader

**(pushed out to future release)** - Over the past couple releases we've had some thoughts about how PluginLoader might be better structured

- Load the loaders via an initialization function(), not when importing the module. (stretch goal, doesn't impact the CLI)
- Separate duties of `PluginLoader` from `PluginFinder`. Most plugins need both but `Modules` and `Module_utils` only need a `PluginFinder`
- Write different `PluginFinder` subclasses for `module_utils` and perhaps `Modules`. Most Plugin types have a flattened namespace and are single python files. `Modules` include code that is not written in python. `Module_utils` are vastly different from the other Plugins as they maintain a hierarchical namespace and are multi-file.
- Potentially split `module_utils` loader for python from `module_utils` loader for powershell. Currently we only support generic `module_utils` for python modules. The powershell modules always include a single, hardcoded powershell `module_utils` file. If we add generic `module_utils` for powershell, we'll need to decide how to organize the code.

## Static Loop Keyword

- **Pushed to 2.5**
- Deprecate (not on standard deprecation cycle) `with_` in favor of `loop:`
- This `loop:` will take only a list
- Remove complexity from loops, lookups are still available to users
- Less confusing having a static directive vs a one that is dynamic depending on plugins loaded.

## Vault

- Support for multiple vault passwords. **(done)**
  - Each decrypted item should know which secret to request **(done)**
  - Support requesting credentials (password prompt) as callbacks
- Ability to open and edit file with encrypted vars deencrypted, and encrypt/format on save

## Globalize Callbacks

**(pushed out to future release)** - Make `send_callback` available to other code that cannot use it. - Would allow for 'full formatting' of output (see JSON callback) - Fixes static 'include' display problem

### Plugins

- Allow plugins to have embedded docs (like modules) (**done**)
- Update ansible-doc and website to generate docs from these ansible/ansible PR #22796. (**ansible-doc working, todo:website**)

### Group Priorities

(**done**) - Start using existing group priority variable to sort/merge group vars - Implementation for this in ansible/ansible PR #22580. - Documentation of group priority variable

### Runtime Check on Modules for Blacklisting

(**pushed out to future release**) - Filter on things like "supported\_by" in module metadata - Provide users with an option of "warning, error or allow/ignore" - Configurable via ansible.cfg and environment variable

### Disambiguate Includes

- Create import\_x for 'static includes' (import\_task, import\_playbook, import\_role)
  - Any directives are applied to the 'imported' tasks
- Create include\_x for 'dynamic includes' (include\_task, include\_role)
  - Any directives apply to the 'include' itself

### Windows

- New PS/.NET module API (**in progress**)
- Windows Nano Server support
- Windows module\_utils pluginloader (**done**)
- Refactor duplicated module code into new module\_utils files (**in progress**)
- Evaluate #Requires directives (existing and new: PS version, OS version, etc)
- Improve module debug support/persistence (**done**)
- Explore official DSC support (**done**)
- Explore module intermediate output
- Explore Powershell module unit testing (**in progress**)
- Explore JEA support (stretch)
- Extended become support with network/service/batch logon types
- Module updates
  - Split "Windows" category into multiple subs
  - Domain user/group management modules (**done**)
  - win\_mapped\_drive module (**done**)
  - win\_hotfix (**done**)

- win\_updates rewrite to require become
- win\_package changes required to deprecate win\_msi **(done)**
- win\_copy re-write **(done)**

## AWS

- Focus on pull requests for various modules
- Triage existing merges for modules
- Module work
  - elb-target-groups #19492, #24583. **(done)**
  - alb\* #19491, #24584. **(done)**
  - ecs #20618. **(in review process)**
  - Data Pipelines #22878. **(in review process)**
  - VPN #24385. **(in review process)**
  - DirectConnect #26152. **(connection module in review process, several more to come)**

## Azure

- Expose endpoint overrides **(done)**
- Reformat/document module output to collapse internal API structures and surface important data (eg, public IPs, NICs, data disks) **(pushed to future)**
- Add load balancer module **(in progress)**
- Add Azure Functions module **(in progress)**

## Google Cloud Platform

- New Module: DataProc
- Support for Cross-Region HTTP Load Balancing
- New Module: GKE

## Network Roadmap

- Removal of \*\_template modules **(done)**
- Distributed Continuous Integration Infrastructure **(done)**
- RPC Connection Plugin **(done)**
- Module Work
  - Declarative intent modules **(done)**
  - OpenVSwitch **(done)**
  - Minimal Viable Platform Agnostic Modules **(done)**

## Contributor Quality of Life

- All Core and Curated modules will work towards having unit testing. (**edit: integration and/or unit tests**)
- More bot improvements!
  - Bot comments on PRs with details of test failures. (**done**)
- Test Infrastructure changes
  - Shippable + Bot Integration
    - \* Provide verified test results to the bot from Shippable so the bot can comment on PRs with CI failures. (**done, compile and sanity tests only**)
    - \* Enable the bot to mark PRs with `ci_verified` if all CI failures are verified. (**done**)
  - Windows Server 2016 Integration Tests
    - \* Restore Windows Server 2016 integration tests on Shippable.
      - Originally enabled during the 2.3 release cycle, but later disabled due to intermittent WinRM issues.
      - Depends on resolution of WinRM connection issues.
  - Windows Server Nano Integration Tests (**pushed to future roadmap**)
    - \* Add support to ansible-core-ci for Windows Server 2016 Nano and enable on Shippable.
    - \* This will use a subset of the existing Windows integration tests.
    - \* Depends on resolution of WinRM connection issues.
  - Windows + Python 3 Tests
    - \* Run basic Windows tests using Python 3 as the controller. (**partially done, not all planned tests running yet**)
    - \* Depends on resolution of WinRM Python 3 issues.
  - Cloud Integration Tests
    - \* Run existing cloud integration tests as part of CI for:
      - AWS (**done**)
      - Azure (**done**)
      - GCP (**pushed to future roadmap**)
    - \* Tests to be run only on cloud module (and module\_utils) PRs and merges for the relevant cloud provider. (**done**)
  - Test Reliability
    - \* Further improve test reliability to reduce false positives on Shippable. (**ongoing**)
    - \* This continues work from the 2.3 release cycle.
  - Static Code Analysis
    - \* Further expand the scope and coverage of static analysis. (**ongoing**)
    - \* This continues work from the 2.3 release cycle.



## 1.27.5 Ansible 2.5

**Core Engine Freeze and Module Freeze: 22 January 2018**

**Core and Curated Module Freeze: 22 January 2018**

**Community Module Freeze: 7 February 2018**

**Release Candidate 1 will be 21 February, 2018**

**Target: March 2018**

### Topics

- *Ansible 2.5*
  - *Engine improvements*
  - *Ansible-Config*
  - *Inventory*
  - *Facts*
  - *Static Loop Keyword*
  - *Vault*
  - *Runtime Check on Modules for Blacklisting*
  - *Windows*
  - *General Cloud*
  - *AWS*
  - *Azure*
  - *Network Roadmap*
  - *Documentation*
  - *Contributor Quality of Life*

### Engine improvements

- Assemble module improvements - assemble just skips when in check mode, it should be able to test if there is a difference and changed=true/false. - The same with diff, it should work as template modules does
- Handle Password reset prompts cleaner
- Tasks stats for rescues and ignores
- Normalize temp dir usage across all subsystems
- Add option to set playbook dir for adhoc, inventory and console to allow for 'relative path loading'

### Ansible-Config

- Extend config to more plugin types and update plugins to support the new config

### Inventory

- ansible-inventory option to output group variable assignment and data (`--export`)
- Create inventory plugins for: - aws

### Facts

- Namespacing fact variables (via a config option) implemented in ansible/ansible PR [#18445](#). Proposal found in ansible/proposals issue [#17](#).
- Make fact collectors and `gather_subset` specs finer grained
- Eliminate unneeded deps between fact collectors
- Allow fact collectors to indicate if they need information from another fact collector to be gathered first.

### Static Loop Keyword

- A simpler alternative to `with_`, `loop`: only takes a list
- Remove complexity from loops, lookups are still available to users
- Less confusing having a static directive vs a one that is dynamic depending on plugins loaded.

### Vault

- Vault secrets client inc new 'keyring' client

### Runtime Check on Modules for Blacklisting

- Filter on things like "supported\_by" in module metadata
- Provide users with an option of "warning, error or allow/ignore"
- Configurable via `ansible.cfg` and environment variable

### Windows

- Implement `gather_subset` on Windows facts
- Fix Windows `async` + `become` to allow them to work together
- Implement Windows `become` flags for controlling various modes (**done**) - `logontype` - elevation behavior
- Convert `win_updates` to action plugin for auto reboot and extra features (**done**)
- Spike out changing the connection over to PSRP instead of WSMV (**done- it's possible**)
- Module updates
  - `win_updates` (**done**)
    - \* Fix `win_updates` to detect (or request) `become`
    - \* Add whitelist/blacklist features to `win_updates`
  - `win_dsc` further improvements (**done**)

## General Cloud

- Make multi-cloud provisioning easier
- Diff mode will output provisioning task results of ansible-playbook runs
- Terraform module

## AWS

- Focus on pull requests for various modules
- Triage existing merges for modules
- Module work
  - ec2\_instance
  - ec2\_vpc: Allow the addition of secondary IPv4 CIDRS to existing VPCs.
  - AWS Network Load Balancer support (NLB module, ASG support, etc)
  - rds\_instance

## Azure

- Azure CLI auth (**done**)
- Fix Azure module results to have "high-level" output instead of raw REST API dictionary (**partial, more to come in 2.6**)
- Deprecate Azure automatic storage accounts in azure\_rm\_virtualmachine (**breaks on Azure Stack, punted until AS supports managed disks**)

## Network Roadmap

- Refactor common network shared code into package (**done**)
- Convert various nxos modules to leverage declarative intent (**done**)
- Refactor various modules to leverage the cliconf plugin (**done**)
- Add various missing declarative modules for supported platforms and functions (**done**)
- Implement a feature that handles platform differences and feature unavailability (**done**)
- netconf-config.py should provide control for deployment strategy
- Create netconf connection plugin (**done**)
- Create netconf fact module
- Turn network\_cli into a usable connection type (**done**)
- Implements jsonrpc message passing for ansible-connection (**done**)
- Improve logging for ansible-connection (**done**)
- Improve stdout output for failures whilst using persistent connection (**done**)
- Create IOS-XR NetConf Plugin and refactor iosxr modules to leverage netconf plugin (**done**)
- Refactor junos modules to use netconf plugin (**done**)

- Filters: Add a filter to convert XML response from a network device to JSON object **(done)**

### Documentation

- Extend documentation to more plugins
- Document vault-password-client scripts.
- Network Documentation
  - New landing page (to replace intro\_networking) **(done)**
  - Platform specific guides **(done)**
  - Walk through: Getting Started **(done)**
  - Networking and become **(done)**
  - Best practice **(done)**

### Contributor Quality of Life

- Finish PSScriptAnalyser integration with ansible-test (for enforcing Powershell style) **(done)**
- Resolve issues requiring skipping of some integration tests on Python 3.

## 1.27.6 Ansible 2.6

### Topics

- *Ansible 2.6*
  - *Release Schedule*
    - \* *Actual*
    - \* *Expected*
  - *Engine improvements*
  - *Core Modules*
  - *Cloud Modules*
  - *Network*
    - \* *Connection work*
    - \* *Modules*
    - \* *Other Features*
  - *Windows*

### Release Schedule

#### Actual

- 2018-05-17 Core Freeze (Engine and Core Modules/Plugins)

- 2018-05-21 Alpha Release 1
- 2018-05-25 Community Freeze (Non-Core Modules/Plugins)
- 2018-05-25 Branch stable-2.6
- 2018-05-30 Alpha Release 2
- 2018-06-05 Release Candidate 1
- 2018-06-08 Release Candidate 2
- 2018-06-18 Release Candidate 3
- 2018-06-25 Release Candidate 4
- 2018-06-26 Release Candidate 5

### Expected

- 2018-06-28 Final Release

### Engine improvements

- Version 2.6 is largely going to be a stabilization release for Core code.
- Some of the items covered in this release, but are not limited to are the following:
  - `ansible-inventory`
  - `import_*`
  - `include_*`
  - Test coverage
  - Performance Testing

### Core Modules

- Adopt-a-module Campaign
  - Review current status of all Core Modules
  - Reduce backlog of open issues against these modules

### Cloud Modules

### Network

### Connection work

- New connection plugin: eAPI [proposal#102](#)
- New connection plugin: NX-API
- Support for configurable options for `network_cli` & `netconf`

### Modules

- New `net_get` - platform agnostic module for pulling configuration via SCP/SFTP over `network_cli`
- New `net_put` - platform agnostic module for pushing configuration via SCP/SFTP over `network_cli`
- New `netconf_get` - Netconf module to fetch configuration and state data [proposal#104](#)

### Other Features

- Stretch & tech preview: Configuration caching for `network_cli`. Opt-in feature to avoid `show running` performance hit

### Windows

---

## A

Action, [501](#)  
Ad Hoc, [501](#)  
ANSIBLE\_DEBUG, [294](#), [453](#), [500](#)  
ANSIBLE\_HOST\_KEY\_CHECKING, [45](#)  
ANSIBLE\_HOSTS, [23](#)  
ANSIBLE\_INVENTORY, [23](#)  
ANSIBLE\_INVENTORY\_IGNORE, [65](#)  
ANSIBLE\_LIBRARY, [287](#)  
ANSIBLE\_LOG\_PATH, [453](#)  
ANSIBLE\_NOCOWS, [496](#)  
ANSIBLE\_NULL\_REPRESENTATION, [11](#)  
ANSIBLE\_ROLES\_PATH, [475](#)  
ANSIBLE\_VAULT\_PASSWORD\_FILE, [201](#)  
Async, [501](#)

## C

Callback Plugin, [501](#)  
Check Mode, [501](#)  
Conditionals, [501](#)  
Connection Plugin, [501](#)

## D

Declarative, [501](#)  
Diff Mode, [502](#)

## E

Executor, [502](#)

## F

Facts, [502](#)  
Filter Plugin, [502](#)  
Forks, [502](#)

## G

Gather Facts (Boolean), [502](#)  
Globbing, [502](#)  
Group, [502](#)  
Group Vars, [502](#)

## H

Handlers, [502](#)  
Host, [502](#)  
Host Specifier, [502](#)  
Host Vars, [503](#)

## I

Idempotency, [503](#)  
Includes, [503](#)  
Inventory, [503](#)  
Inventory Script, [503](#)

## J

Jinja2, [503](#)  
JSON, [503](#)

## L

Lazy Evaluation, [503](#)  
Library, [503](#)  
Limit Groups, [503](#)  
Local Action, [503](#)  
Local Connection, [503](#)  
Lookup Plugin, [503](#)  
Loops, [503](#)

## M

Modules, [504](#)  
Multi-Tier, [504](#)

## N

Notify, [504](#)

## O

Orchestration, [504](#)

## P

paramiko, [504](#)  
Playbooks, [504](#)  
Plays, [504](#)  
Pull Mode, [504](#)

Push Mode, [504](#)

## R

Register Variable, [504](#)

Resource Model, [504](#)

Roles, [505](#)

Rolling Update, [505](#)

## S

Serial, [505](#)

SSH (Native), [505](#)

Sudo, [505](#)

## T

Tags, [505](#)

Task, [505](#)

Tasks, [505](#)

Templates, [505](#)

Transport, [505](#)

## V

Vars (Variables), [505](#)

## W

When, [505](#)

## Y

YAML, [506](#)