Ansible Documentation

Release v2.2.0.0-1-54-gf6b47c5

Ansible, Inc

November 18, 2016

Part I

About Ansible

Welcome to the Ansible documentation!

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 an accelerated socket mode 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 (v2.2.0.0-1-54-gf6b47c5) and also some development version features (**|versiondev|**). For recent features, we note in each section the version of Ansible where the feature was added.

Ansible, Inc. 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, typically adding 20 or so new modules in each release.

CHAPTER

ONE

INTRODUCTION

Before we dive into the really fun parts – playbooks, configuration management, deployment, and orchestration – we'll learn how to get Ansible installed and cover some basic concepts. We'll also go over how to execute ad-hoc commands in parallel across your nodes using /usr/bin/ansible, and see what sort of modules are available in Ansible's core (you can also write your own, which is covered later).

Installation

Topics

Getting Ansible

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.

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.

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.

Control Machine Requirements

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

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

Note: As of 2.0 ansible uses a few more file handles to manage its forks. OS X has a very low setting 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.

Warning: 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.

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 2.4 or later. If you are running less than Python 2.5 on the remotes, you will also need:

- `python-simplejson`

Note: Ansible's "raw" module (for executing commands in a quick and dirty way) and the script module don't even need that. So technically, you can use Ansible to install python-simplejson using the raw module, which then allows you to use everything else. (That's jumping ahead though.)

Note: 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 of course still use the yum module in Ansible to install this package on remote systems that do not have it.

Note: Python 3 is a slightly different language than Python 2 and some Python programs (including Ansible) are not switching over yet. Ansible uses Python 2 in order to maintain compability with older distributions such as RHEL 5 and RHEL 6. However, some Linux distributions (Gentoo, Arch) may not have a Python 2.X interpreter installed by default. On those systems, you should install one, and set the 'ansible_python_interpreter' variable in inventory (see *Inventory*) to point at your 2.X Python. Distributions like Red Hat Enterprise Linux, CentOS, Fedora, and Ubuntu all have a 2.X interpreter installed by default and this does not apply to those distributions. This is also true of nearly all Unix systems.

If you need to bootstrap these remote systems by installing Python 2.X, using the 'raw' module will be able to do it remotely. For example, `ansible myhost --sudo -m raw -a "yum install -y python2 python-simplejson"` would install Python 2.X and the simplejson module needed to run ansible and its modules.

Installing the Control Machine

Running From Source

Ansible is trivially easy to run from a checkout, root permissions are not required to use it and there is no software to actually install for Ansible itself. 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 also easily contribute to the project. Because there is nothing to install, following the development version is significantly easier than most open source projects.

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

To install from source.

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

Using Bash:

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

Using Fish:

```
$ . ./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 [1]:

[1] If you have issues with the "pycrypto" package install on Mac OSX, which is included as a dependency for paramiko, then you may need to try "CC=clang sudo -E pip install pycrypto".

```
$ sudo pip install paramiko PyYAML Jinja2 httplib2 six
```

Note when updating ansible, be sure to not only update the source tree, but also the "submodules" in git which point at Ansible's own modules (not the same kind of modules, alas).

```
$ git pull --rebase
$ 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 *Inventory*) other than /etc/ansible/hosts:

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

Note: 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" if you wish.

Latest Release Via Yum

RPMs are available from yum for EPEL 6, 7, and currently supported Fedora distributions.

Ansible itself can manage earlier operating systems that contain Python 2.4 or higher (so also EL5).

Fedora users can install Ansible directly, though if you are using RHEL or CentOS and have not already done so, configure EPEL

```
# install the epel-release RPM if needed on CentOS, RHEL, or Scientific Linux
$ sudo yum install ansible
```

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. Make sure you have `rpm-build`, `make`, `asciidoc`, `git`, `python-setuptools` and `python2-devel` installed.

```
$ git clone git://github.com/ansible/ansible.git --recursive
$ 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 install software-properties-common
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install ansible
```

Note: For the older version 1.9 we use this ppa:ansible/ansible-1.9

Note: 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 above.

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
```

Note: If you have Python 3 as a default Python slot on your Gentoo nodes (default setting), then you must set `ansible_python_interpreter = /usr/bin/python2` in your group or inventory variables.

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.

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.

Note: If you have Python 3 as a default Python slot on your Arch nodes (default setting), then you must set `ansible_python_interpreter = /usr/bin/python2` in your group or inventory variables.

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 [1]:

```
$ sudo pip install ansible
```

Or if you are looking for the latest development version:

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

If you are installing on OS X 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.

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.

See also:

Introduction To Ad-Hoc Commands Examples of basic commands

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

Getting Started

Topics

Foreword

Now that you've read *Installation* and installed Ansible, it's time to dig in and get started with some 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 going. Once you have these concepts down, read *Introduction To Ad-Hoc Commands* for some more detail, and then you'll be ready to dive into 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 1.3 and later 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, OS X, or Ubuntu as your control machine until a newer version of OpenSSH is available for your platform – or engage 'accelerated mode' in Ansible. See *Accelerated Mode*.

In releases up to and including Ansible 1.2, the default was strictly paramiko. Native SSH had to be explicitly selected with the `-c` ssh option or set in the configuration file.

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 *Configuration file*.

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).

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: *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 *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!

Host Key Checking

Ansible 1.2.1 and later have 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 an 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 *Configuration file* 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.

See also:

Inventory More information about inventory

Introduction To Ad-Hoc Commands Examples of basic commands

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

Inventory

Topics

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 file, 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 (explained below) and also pull inventory from dynamic or cloud sources, as described in *Dynamic Inventory*.

Hosts and Groups

The format for `/etc/ansible/hosts` is an INI-like format and looks like this:

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

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

The things 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.

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, and 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 like this:

```
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. We're just getting started.

Adding a lot of hosts? If you have 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
```

Note: Ansible 2.0 has deprecated the "ssh" from `ansible_ssh_user`, `ansible_ssh_host`, and `ansible_ssh_port` to become `ansible_user`, `ansible_host`, and `ansible_port`. If you are using a version of Ansible prior to 2.0, you should continue using the older style variables (`ansible_ssh_*`). These shorter variables are ignored, without warning, in older versions of Ansible.

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 alluded to 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:

```
[atlanta]
host1
host2

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

Groups of Groups, and Group Variables

It is also possible to make groups of groups using the `:children` suffix. Just like above, you can apply variables using `: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
```

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.

Splitting Out Host and Group Specific Data

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

In addition to storing variables directly in the INI file, host and group variables can be stored in individual files relative to the inventory file.

These variable files are in YAML format. Valid file extensions include '.yml', '.yaml', '.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 'webservers', variables in YAML files at the following locations will be made available to the host:

```
/etc/ansible/group_vars/raleigh # can optionally end in '.yml', '.yaml', or '.json'
/etc/ansible/group_vars/webservers
/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 ok 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. 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. Note that this only works on Ansible 1.4 or later.

Tip: In Ansible 1.2 or later 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.

List of Behavioral Inventory Parameters

As alluded to above, setting the following variables controls how ansible interacts with remote hosts.

Host connection:

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.

Note: Ansible 2.0 has deprecated the "ssh" from `ansible_ssh_user`, `ansible_ssh_host`, and `ansible_ssh_port` to become `ansible_user`, `ansible_host`, and `ansible_port`. If you are using a version of Ansible prior to 2.0, you should continue using the older style variables (`ansible_ssh_*`). These shorter variables are ignored, without warning, in older versions of Ansible.

SSH connection:

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.

ansible_ssh_pass The ssh password to use (this is insecure, we strongly recommend using `--ask-pass` or SSH keys)

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`.

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 (this is insecure, we strongly recommend using `--ask-become-pass` or SSH keys)

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.

New in version 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 a 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 is executing playbooks over SSH but is not limited to. With the host specific parameter `ansible_connection=<connector>` the connection type can be changed. 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. 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:
    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

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

See also:

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

Introduction To Ad-Hoc Commands Examples of basic commands

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

Dynamic Inventory

Topics

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 *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 enterprisey 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_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"
```

Note: 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, this option is not supported by `ansible-playbook` though. But you can use the `AWS_PROFILE` variable - e.g. `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, feel free to edit `ec2.ini` and list only the regions you are interested in. There are other config options in `ec2.ini` including cache control, and destination variables.

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 at: https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/openstack.py

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
```

Note: 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.

Note: 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.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
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, .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
```

See also:

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

Patterns

Topics

- *Patterns*

Patterns in Ansible are how we decide which hosts to manage. This can mean what hosts to communicate with, but in terms of *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 webservers -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 "webservers" 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:

```
webservers
webservers:dbservers
```

You can exclude groups as well, for instance, all machines must be in the group webservers but not in the group phoenix:

```
webservers:!phoenix
```

You can also specify the intersection of two groups. This would mean the hosts must be in the group webservers and the host must also be in the group staging:

```
webservers:&staging
```

You can do combinations:

```
webservers:dbservers:&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[0:1]      # == webservers[0],webservers[1]
                     # == cobweb,webbing
webservers[1:]       # == 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 '@'. Since Ansible 1.2:

```
ansible-playbook site.yml --limit @retry_hosts.txt
```

Easy enough. See *Introduction To Ad-Hoc Commands* and then *Playbooks* for how to apply this knowledge.

Note: With the exception of version 1.9, you can use ',' instead of ':' as a host list separator. The ',' is prefered specially when dealing with ranges and ipv6.

Note: As of 2.0 the ';' is deprecated as a host list separator.

See also:

Introduction To Ad-Hoc Commands Examples of basic commands

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

Introduction To Ad-Hoc Commands

Topics

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 *Playbooks* for more information about those)

If you haven't read *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-user otheruser [--ask-
↪become-pass]
```

Note: 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 *Patterns*.

The `-f 10` in the above specifies the usage of 10 simultaneous processes to use. You can also set this in *Configuration file* 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 *About Modules*.

Note: 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 *About 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 usually do not work like simple scripts. They make the remote system look like a state, and run the commands necessary to get it there. This is commonly referred to as 'idempotence', and is a core design goal of Ansible. However, we also recognize that running arbitrary commands is equally important, 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 *About 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 webservers -m git -a "repo=git://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 webservers:

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

Alternatively, restart a service on all webservers:

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

Ensure a service is stopped:

```
$ ansible webservers -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*.

See also:

Configuration file All about the Ansible config file

About Modules A list of available modules

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

Configuration file

Topics

Certain settings in Ansible are adjustable via a configuration file. The stock configuration should be sufficient for most users, but there may be reasons you would want to change them.

Changes can be made and used in a configuration file which will be processed in the following order:

```
* ANSIBLE_CONFIG (an environment variable)
* ansible.cfg (in the current directory)
* .ansible.cfg (in the home directory)
* /etc/ansible/ansible.cfg
```

Prior to 1.5 the order was:

```
* ansible.cfg (in the current directory)
* ANSIBLE_CONFIG (an environment variable)
* .ansible.cfg (in the home directory)
* /etc/ansible/ansible.cfg
```

Ansible will process the above list and use the first file found. Settings in files are not merged.

Note: Comments The configuration file is one variant of an INI format. Both the hash sign ("#") and semicolon (";") are allowed as comment markers when the comment starts the line. However, if the comment is inline with regular values, only the semicolon is allowed to introduce the comment. For instance:

```
# some basic default values...
inventory = /etc/ansible/hosts  ; This points to the file that lists your hosts
```

Getting the latest configuration

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

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

You may wish to consult the ansible.cfg in source control for all of the possible latest values.

Environmental configuration

Ansible also allows configuration of settings via environment variables. If these environment variables are set, they will override any setting loaded from the configuration file. These variables are for brevity not defined here, but look in constants.py in the source tree if you want to use these. They are mostly considered to be a legacy system as compared to the config file, but are equally valid.

Explanation of values by section

The configuration file is broken up into sections. Most options are in the "general" section but some sections of the file are specific to certain connection types.

General defaults

In the [defaults] section of ansible.cfg, the following settings are tunable:

action_plugins

Actions are pieces of code in ansible that enable things like module execution, templating, and so forth.

This is a developer-centric feature that allows low-level extensions around Ansible to be loaded from different locations:

```
action_plugins = ~/.ansible/plugins/action_plugins/:/usr/share/ansible_plugins/
↪action_plugins
```

Most users will not need to use this feature. See developing_plugins for more details.

allow_world_readable_tmpfiles

New in version 2.1.

This makes the temporary files created on the machine to be world readable and will issue a warning instead of failing the task.

It is useful when becoming an unprivileged user:

```
allow_world_readable_tmpfiles=True
```

ansible_managed

Ansible-managed is a string that can be inserted into files written by Ansible's config templating system, if you use a string like:

```
{{ ansible_managed }}
```

The default configuration shows who modified a file and when:

```
ansible_managed = Ansible managed: {file} modified on %Y-%m-%d %H:%M:%S by {uid}␣
↪on {host}
```

This is useful to tell users that a file has been placed by Ansible and manual changes are likely to be overwritten.

Note that if using this feature, and there is a date in the string, the template will be reported changed each time as the date is updated.

ask_pass

This controls whether an Ansible playbook should prompt for a password by default. The default behavior is no:

```
ask_pass=True
```

If using SSH keys for authentication, it's probably not needed to change this setting.

ask_sudo_pass

Similar to ask_pass, this controls whether an Ansible playbook should prompt for a sudo password by default when sudoing. The default behavior is also no:

```
ask_sudo_pass=True
```

Users on platforms where sudo passwords are enabled should consider changing this setting.

ask_vault_pass

This controls whether an Ansible playbook should prompt for the vault password by default. The default behavior is no:

```
ask_vault_pass=True
```

bin_ansible_callbacks

New in version 1.8.

Controls whether callback plugins are loaded when running /usr/bin/ansible. This may be used to log activity from the command line, send notifications, and so on. Callback plugins are always loaded for /usr/bin/ansible-playbook if present and cannot be disabled:

```
bin_ansible_callbacks=False
```

Prior to 1.8, callbacks were never loaded for /usr/bin/ansible.

callback_plugins

Callbacks are pieces of code in ansible that get called on specific events, permitting to trigger notifications.

This is a developer-centric feature that allows low-level extensions around Ansible to be loaded from different locations:

```
callback_plugins = ~/.ansible/plugins/callback:/usr/share/ansible/plugins/callback
```

Most users will not need to use this feature. See developing_plugins for more details

stdout_callback

New in version 2.0.

This setting allows you to override the default stdout callback for ansible-playbook:

```
stdout_callback = skippy
```

callback_whitelist

New in version 2.0.

Now ansible ships with all included callback plugins ready to use but they are disabled by default. This setting lets you enable a list of additional callbacks. This cannot change or override the default stdout callback, use *stdout_callback* for that:

```
callback_whitelist = timer,mail
```

command_warnings

New in version 1.8.

By default since Ansible 1.8, Ansible will issue a warning when the shell or command module is used and the command appears to be similar to an existing Ansible module. For example, this can include reminders to use the 'git' module instead of shell commands to execute 'git'. Using modules when possible over arbitrary shell commands can lead to more reliable and consistent playbook runs, and also easier to maintain playbooks:

```
command_warnings = False
```

These warnings can be silenced by adjusting the following setting or adding warn=yes or warn=no to the end of the command line parameter string, like so:

```
- name: usage of git that could be replaced with the git module
  shell: git update foo warn=yes
```

connection_plugins

Connections plugin permit to extend the channel used by ansible to transport commands and files.

This is a developer-centric feature that allows low-level extensions around Ansible to be loaded from different locations:

```
connection_plugins = ~/.ansible/plugins/connection_plugins/:/usr/share/ansible_
↪plugins/connection_plugins
```

Most users will not need to use this feature. See developing_plugins for more details

deprecation_warnings

New in version 1.3.

Allows disabling of deprecating warnings in ansible-playbook output:

```
deprecation_warnings = True
```

Deprecation warnings indicate usage of legacy features that are slated for removal in a future release of Ansible.

display_args_to_stdout

New in version 2.1.0.

By default, ansible-playbook will print a header for each task that is run to stdout. These headers will contain the `name:` field from the task if you specified one. If you didn't then ansible-playbook uses the task's action to help you tell which task is presently running. Sometimes you run many of the same action and so you want more information about the task to differentiate it from others of the same action. If you set this variable to `True` in the config then ansible-playbook will also include the task's arguments in the header.

This setting defaults to `False` because there is a chance that you have sensitive values in your parameters and do not want those to be printed to stdout:

```
display_args_to_stdout=False
```

If you set this to `True` you should be sure that you have secured your environment's stdout (no one can shoulder surf your screen and you aren't saving stdout to an insecure file) or made sure that all of your playbooks explicitly added the `no_log: True` parameter to tasks which have sensistive values See *How do I keep secret data in my playbook?* for more information.

display_skipped_hosts

If set to *False*, ansible will not display any status for a task that is skipped. The default behavior is to display skipped tasks:

```
display_skipped_hosts=True
```

Note that Ansible will always show the task header for any task, regardless of whether or not the task is skipped.

error_on_undefined_vars

On by default since Ansible 1.3, this causes ansible to fail steps that reference variable names that are likely typoed:

```
error_on_undefined_vars=True
```

If set to False, any '{{ template_expression }}' that contains undefined variables will be rendered in a template or ansible action line exactly as written.

executable

This indicates the command to use to spawn a shell under a sudo environment. Users may need to change this to /bin/bash in rare instances when sudo is constrained, but in most cases it may be left as is:

```
executable = /bin/bash
```

Starting in version 2.1 this can be overridden by the inventory var `ansible_shell_executable`.

filter_plugins

Filters are specific functions that can be used to extend the template system.

This is a developer-centric feature that allows low-level extensions around Ansible to be loaded from different locations:

```
filter_plugins = ~/.ansible/plugins/filter_plugins/:/usr/share/ansible_plugins/
↪filter_plugins
```

Most users will not need to use this feature. See developing_plugins for more details

force_color

This options forces color mode even when running without a TTY:

```
force_color = 1
```

force_handlers

New in version 1.9.1.

This option causes notified handlers to run on a host even if a failure occurs on that host:

```
force_handlers = True
```

The default is False, meaning that handlers will not run if a failure has occurred on a host. This can also be set per play or on the command line. See *Handlers and Failure* for more details.

forks

This is the default number of parallel processes to spawn when communicating with remote hosts. Since Ansible 1.3, the fork number is automatically limited to the number of possible hosts, so this is really a limit of how much network and CPU load you think you can handle. Many users may set this to 50, some set it to 500 or more. If you have a large number of hosts, higher values will make actions across all of those hosts complete faster. The default is very very conservative:

```
forks=5
```

gathering

New in 1.6, the 'gathering' setting controls the default policy of facts gathering (variables discovered about remote systems).

The value 'implicit' is the default, which means that the fact cache will be ignored and facts will be gathered per play unless 'gather_facts: False' is set. The value 'explicit' is the inverse, facts will not be gathered unless directly requested in the play. The value 'smart' means each new host that has no facts discovered will be scanned, but if the same host is addressed in multiple plays it will not be contacted again in the playbook run. This option can be useful for those wishing to save fact gathering time. Both 'smart' and 'explicit' will use the fact cache:

```
gathering = smart
```

New in version 2.1.

You can specify a subset of gathered facts using the following option:

```
gather_subset = all
```

all gather all subsets (the default)

network gather network facts

hardware gather hardware facts (longest facts to retrieve)

virtual gather facts about virtual machines hosted on the machine

ohai gather facts from ohai

facter gather facts from facter

You can combine them using a comma separated list (ex: network,virtual,facter)

You can also disable specific subsets by prepending with a *!* like this:

```
# Don't gather hardware facts, facts from chef's ohai or puppet's facter
gather_subset = !hardware,!ohai,!facter
```

A set of basic facts are always collected no matter which additional subsets are selected. If you want to collect the minimal amount of facts, use *!all*:

```
gather_subset = !all
```

hash_behaviour

Ansible by default will override variables in specific precedence orders, as described in *Variables*. When a variable of higher precedence wins, it will replace the other value.

Some users prefer that variables that are hashes (aka 'dictionaries' in Python terms) are merged. This setting is called 'merge'. This is not the default behavior and it does not affect variables whose values are scalars (integers, strings) or arrays. We generally recommend not using this setting unless you think you have an absolute need for it, and playbooks in the official examples repos do not use this setting:

```
hash_behaviour=replace
```

The valid values are either 'replace' (the default) or 'merge'.

New in version 2.0.

If you want to merge hashes without changing the global settings, use the *combine* filter described in *Jinja2 filters*.

hostfile

This is a deprecated setting since 1.9, please look at *inventory* for the new setting.

host_key_checking

As described in *Getting Started*, host key checking is on by default in Ansible 1.3 and later. If you understand the implications and wish to disable it, you may do so here by setting the value to False:

```
host_key_checking=True
```

internal_poll_interval

New in version 2.2.

This sets the interval (in seconds) of Ansible internal processes polling each other. Lower values improve performance with large playbooks at the expense of extra CPU load. Higher values are more suitable for Ansible usage in automation scenarios, when UI responsiveness is not required but CPU usage might be a concern. Default corresponds to the value hardcoded in Ansible 2.1:

```
internal_poll_interval=0.001
```

inventory

This is the default location of the inventory file, script, or directory that Ansible will use to determine what hosts it has available to talk to:

```
inventory = /etc/ansible/hosts
```

It used to be called hostfile in Ansible before 1.9

jinja2_extensions

This is a developer-specific feature that allows enabling additional Jinja2 extensions:

```
jinja2_extensions = jinja2.ext.do,jinja2.ext.i18n
```

If you do not know what these do, you probably don't need to change this setting :)

library

This is the default location Ansible looks to find modules:

```
library = /usr/share/ansible
```

Ansible knows how to look in multiple locations if you feed it a colon separated path, and it also will look for modules in the "./library" directory alongside a playbook.

local_tmp

New in version 2.1.

When Ansible gets ready to send a module to a remote machine it usually has to add a few things to the module: Some boilerplate code, the module's parameters, and a few constants from the config file. This combination of things gets stored in a temporary file until ansible exits and cleans up after itself. The default location is a subdirectory of the user's home directory. If you'd like to change that, you can do so by altering this setting:

```
local_tmp = $HOME/.ansible/tmp
```

Ansible will then choose a random directory name inside this location.

log_path

If present and configured in ansible.cfg, Ansible will log information about executions at the designated location. Be sure the user running Ansible has permissions on the logfile:

```
log_path=/var/log/ansible.log
```

This behavior is not on by default. Note that ansible will, without this setting, record module arguments called to the syslog of managed machines. Password arguments are excluded.

For Enterprise users seeking more detailed logging history, you may be interested in *Ansible Tower*.

lookup_plugins

This is a developer-centric feature that allows low-level extensions around Ansible to be loaded from different locations:

```
lookup_plugins = ~/.ansible/plugins/lookup_plugins/:/usr/share/ansible_plugins/
↪lookup_plugins
```

Most users will not need to use this feature. See developing_plugins for more details

module_set_locale

This boolean value controls whether or not Ansible will prepend locale-specific environment variables (as specified via the *module_lang* configuration option). If enabled, it results in the LANG, LC_MESSAGES, and LC_ALL being set when the module is executed on the given remote system. By default this is disabled.

Note: The module_set_locale option was added in Ansible-2.1 and defaulted to True. The default was changed to False in Ansible-2.2

---

module_lang

This is to set the default language to communicate between the module and the system. By default, the value is value *LANG* on the controller or, if unset, *en_US.UTF-8* (it used to be *C* in previous versions):

```
module_lang = en_US.UTF-8
```

module_name

This is the default module name (-m) value for /usr/bin/ansible. The default is the 'command' module. Remember the command module doesn't support shell variables, pipes, or quotes, so you might wish to change it to 'shell':

```
module_name = command
```

nocolor

By default ansible will try to colorize output to give a better indication of failure and status information. If you dislike this behavior you can turn it off by setting 'nocolor' to 1:

```
nocolor=0
```

nocows

By default ansible will take advantage of cowsay if installed to make /usr/bin/ansible-playbook runs more exciting. Why? We believe systems management should be a happy experience. If you do not like the cows, you can disable them by setting 'nocows' to 1:

```
nocows=0
```

pattern

This is the default group of hosts to talk to in a playbook if no "hosts:" stanza is supplied. The default is to talk to all hosts. You may wish to change this to protect yourself from surprises:

```
hosts=*
```

Note that /usr/bin/ansible always requires a host pattern and does not use this setting, only /usr/bin/ansible-playbook.

poll_interval

For asynchronous tasks in Ansible (covered in *Asynchronous Actions and Polling*), this is how often to check back on the status of those tasks when an explicit poll interval is not supplied. The default is a reasonably moderate 15 seconds which is a tradeoff between checking in frequently and providing a quick turnaround when something may have completed:

```
poll_interval=15
```

private_key_file

If you are using a pem file to authenticate with machines rather than SSH agent or passwords, you can set the default value here to avoid re-specifying `--private-key` with every invocation:

```
private_key_file=/path/to/file.pem
```

remote_port

This sets the default SSH port on all of your systems, for systems that didn't specify an alternative value in inventory. The default is the standard 22:

```
remote_port = 22
```

remote_tmp

Ansible works by transferring modules to your remote machines, running them, and then cleaning up after itself. In some cases, you may not wish to use the default location and would like to change the path. You can do so by altering this setting:

```
remote_tmp = $HOME/.ansible/tmp
```

The default is to use a subdirectory of the user's home directory. Ansible will then choose a random directory name inside this location.

remote_user

This is the default username ansible will connect as for /usr/bin/ansible-playbook. Note that /usr/bin/ansible will always default to the current user if this is not defined:

```
remote_user = root
```

retry_files_enabled

This controls whether a failed Ansible playbook should create a .retry file. The default setting is True:

```
retry_files_enabled = False
```

retry_files_save_path

The retry files save path is where Ansible will save .retry files when a playbook fails and retry_files_enabled is True (the default). The default location is adjacent to the play (~/ in versions older than 2.0) and can be changed to any writeable path:

```
retry_files_save_path = ~/.ansible/retry-files
```

The directory will be created if it does not already exist.

roles_path

New in version 1.4.

The roles path indicate additional directories beyond the 'roles/' subdirectory of a playbook project to search to find Ansible roles. For instance, if there was a source control repository of common roles and a different repository of playbooks, you might choose to establish a convention to checkout roles in /opt/mysite/roles like so:

```
roles_path = /opt/mysite/roles
```

Additional paths can be provided separated by colon characters, in the same way as other pathstrings:

```
roles_path = /opt/mysite/roles:/opt/othersite/roles
```

Roles will be first searched for in the playbook directory. Should a role not be found, it will indicate all the possible paths that were searched.

squash_actions

New in version 2.0.

Ansible can optimise actions that call modules that support list parameters when using with_ looping. Instead of calling the module once for each item, the module is called once with the full list.

The default value for this setting is only for certain package managers, but it can be used for any module:

```
squash_actions = apk,apt,dnf,homebrew,package,pacman,pkgng,yum,zypper
```

Currently, this is only supported for modules that have a name parameter, and only when the item is the only thing being passed to the parameter.

strategy_plugins

Strategy plugin allow users to change the way in which Ansible runs tasks on targeted hosts.

This is a developer-centric feature that allows low-level extensions around Ansible to be loaded from different locations:

```
strategy_plugins = ~/.ansible/plugins/strategy_plugins/:/usr/share/ansible_plugins/
↪strategy_plugins
```

Most users will not need to use this feature. See developing_plugins for more details

sudo_exe

If using an alternative sudo implementation on remote machines, the path to sudo can be replaced here provided the sudo implementation is matching CLI flags with the standard sudo:

```
sudo_exe=sudo
```

sudo_flags

Additional flags to pass to sudo when engaging sudo support. The default is '-H -S -n' which sets the HOME environment variable, prompts for passwords via STDIN, and avoids prompting the user for input of any kind. Note that '-n' will conflict with using password-less sudo auth, such as pam_ssh_agent_auth. In some situations you may wish to add or remove flags, but in general most users will not need to change this setting::

```
sudo_flags=-H -S -n
```

sudo_user

This is the default user to sudo to if `--sudo-user` is not specified or 'sudo_user' is not specified in an Ansible playbook. The default is the most logical: 'root':

```
sudo_user=root
```

system_warnings

New in version 1.6.

Allows disabling of warnings related to potential issues on the system running ansible itself (not on the managed hosts):

```
system_warnings = True
```

These may include warnings about 3rd party packages or other conditions that should be resolved if possible.

timeout

This is the default SSH timeout to use on connection attempts:

```
timeout = 10
```

transport

This is the default transport to use if "-c <transport_name>" is not specified to /usr/bin/ansible or /usr/bin/ansible-playbook. The default is 'smart', which will use 'ssh' (OpenSSH based) if the local operating system is new enough to support ControlPersist technology, and then will otherwise use 'paramiko'. Other transport options include 'local', 'chroot', 'jail', and so on.

Users should usually leave this setting as 'smart' and let their playbooks choose an alternate setting when needed with the 'connection:' play parameter:

```
transport = paramiko
```

vars_plugins

This is a developer-centric feature that allows low-level extensions around Ansible to be loaded from different locations:

```
vars_plugins = ~/.ansible/plugins/vars_plugins/:/usr/share/ansible_plugins/vars_
↪plugins
```

Most users will not need to use this feature. See developing_plugins for more details

vault_password_file

New in version 1.7.

Configures the path to the Vault password file as an alternative to specifying `--vault-password-file` on the command line:

```
vault_password_file = /path/to/vault_password_file
```

As of 1.7 this file can also be a script. 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.

Privilege Escalation Settings

Ansible can use existing privilege escalation systems to allow a user to execute tasks as another. As of 1.9 'become' supersedes the old sudo/su, while still being backwards compatible. Settings live under the [privilege_escalation] header.

become

The equivalent of adding sudo: or su: to a play or task, set to true/yes to activate privilege escalation. The default behavior is no:

```
become=True
```

become_method

Set the privilege escalation method. The default is `sudo`, other options are `su`, `pbrun`, `pfexec`, `doas`, `ksu`:

```
become_method=su
```

become_user

The equivalent to ansible_sudo_user or ansible_su_user, allows to set the user you become through privilege escalation. The default is 'root':

```
become_user=root
```

become_ask_pass

Ask for privilege escalation password, the default is False:

```
become_ask_pass=True
```

become_allow_same_user

Most of the time, using *sudo* to run a command as the same user who is running *sudo* itself is unnecessary overhead, so Ansible does not allow it. However, depending on the *sudo* configuration, it may be necessary to run a command as the same user through *sudo*, such as to switch SELinux contexts. For this reason, you can set `become_allow_same_user` to `True` and disable this optimization.

Paramiko Specific Settings

Paramiko is the default SSH connection implementation on Enterprise Linux 6 or earlier, and is not used by default on other platforms. Settings live under the [paramiko_connection] header.

record_host_keys

The default setting of yes will record newly discovered and approved (if host key checking is enabled) hosts in the user's hostfile. This setting may be inefficient for large numbers of hosts, and in those situations, using the ssh transport is definitely recommended instead. Setting it to False will improve performance and is recommended when host key checking is disabled:

```
record_host_keys=True
```

proxy_command

New in version 2.1.

Use an OpenSSH like ProxyCommand for proxying all Paramiko SSH connections through a bastion or jump host. Requires a minimum of Paramiko version 1.9.0. On Enterprise Linux 6 this is provided by `python-paramiko1.10` in the EPEL repository:

```
proxy_command = ssh -W "%h:%p" bastion
```

OpenSSH Specific Settings

Under the [ssh_connection] header, the following settings are tunable for SSH connections. OpenSSH is the default connection type for Ansible on OSes that are new enough to support ControlPersist. (This means basically all operating systems except Enterprise Linux 6 or earlier).

ssh_args

If set, this will pass a specific set of options to Ansible rather than Ansible's usual defaults:

```
ssh_args = -o ControlMaster=auto -o ControlPersist=60s
```

In particular, users may wish to raise the ControlPersist time to encourage performance. A value of 30 minutes may be appropriate. If *ssh_args* is set, the default `control_path` setting is not used.

control_path

This is the location to save ControlPath sockets. This defaults to:

```
control_path=%(directory)s/ansible-ssh-%%h-%%p-%%r
```

On some systems with very long hostnames or very long path names (caused by long user names or deeply nested home directories) this can exceed the character limit on file socket names (108 characters for most platforms). In that case, you may wish to shorten the string to something like the below:

```
control_path = %(directory)s/%%h-%%r
```

Ansible 1.4 and later will instruct users to run with "-vvvv" in situations where it hits this problem and if so it is easy to tell there is too long of a Control Path filename. This may be frequently encountered on EC2. This setting is ignored if `ssh_args` is set.

scp_if_ssh

Occasionally users may be managing a remote system that doesn't have SFTP enabled. If set to True, we can cause scp to be used to transfer remote files instead:

```
scp_if_ssh=False
```

There's really no reason to change this unless problems are encountered, and then there's also no real drawback to managing the switch. Most environments support SFTP by default and this doesn't usually need to be changed.

pipelining

Enabling pipelining reduces the number of SSH operations required to execute a module on the remote server, by executing many ansible modules without actual file transfer. This can result in a very significant performance improvement when enabled, however when using "sudo:" operations you must first disable 'requiretty' in /etc/sudoers on all managed hosts.

By default, this option is disabled to preserve compatibility with sudoers configurations that have requiretty (the default on many distros), but is highly recommended if you can enable it, eliminating the need for *Accelerated Mode*:

```
pipelining=False
```

Accelerated Mode Settings

Under the [accelerate] header, the following settings are tunable for *Accelerated Mode*. Acceleration is a useful performance feature to use if you cannot enable *pipelining* in your environment, but is probably not needed if you can.

accelerate_port

New in version 1.3.

This is the port to use for accelerated mode:

```
accelerate_port = 5099
```

accelerate_timeout

New in version 1.4.

This setting controls the timeout for receiving data from a client. If no data is received during this time, the socket connection will be closed. A keepalive packet is sent back to the controller every 15 seconds, so this timeout should not be set lower than 15 (by default, the timeout is 30 seconds):

```
accelerate_timeout = 30
```

accelerate_connect_timeout

New in version 1.4.

This setting controls the timeout for the socket connect call, and should be kept relatively low. The connection to the *accelerate_port* will be attempted 3 times before Ansible will fall back to ssh or paramiko (depending on your default connection setting) to try and start the accelerate daemon remotely. The default setting is 1.0 seconds:

```
accelerate_connect_timeout = 1.0
```

Note, this value can be set to less than one second, however it is probably not a good idea to do so unless you're on a very fast and reliable LAN. If you're connecting to systems over the internet, it may be necessary to increase this timeout.

accelerate_daemon_timeout

New in version 1.6.

This setting controls the timeout for the accelerated daemon, as measured in minutes. The default daemon timeout is 30 minutes:

```
accelerate_daemon_timeout = 30
```

Note, prior to 1.6, the timeout was hard-coded from the time of the daemon's launch. For version 1.6+, the timeout is now based on the last activity to the daemon and is configurable via this option.

accelerate_multi_key

New in version 1.6.

If enabled, this setting allows multiple private keys to be uploaded to the daemon. Any clients connecting to the daemon must also enable this option:

```
accelerate_multi_key = yes
```

New clients first connect to the target node over SSH to upload the key, which is done via a local socket file, so they must have the same access as the user that launched the daemon originally.

Selinux Specific Settings

These are settings that control SELinux interactions.

special_context_filesystems

New in version 1.9.

This is a list of file systems that require special treatment when dealing with security context. The normal behaviour is for operations to copy the existing context or use the user default, this changes it to use a file system dependent context. The default list is: nfs,vboxsf,fuse,ramfs:

```
special_context_filesystems = nfs,vboxsf,fuse,ramfs,myspecialfs
```

libvirt_lxc_noseclabel

New in version 2.1.

This setting causes libvirt to connect to lxc containers by passing –noseclabel to virsh. This is necessary when running on systems which do not have SELinux. The default behavior is no:

```
libvirt_lxc_noseclabel = True
```

Galaxy Settings

The following options can be set in the [galaxy] section of ansible.cfg:

server

Override the default Galaxy server value of https://galaxy.ansible.com. Useful if you have a hosted version of the Galaxy web app or want to point to the testing site https://galaxy-qa.ansible.com. It does not work against private, hosted repos, which Galaxy can use for fetching and installing roles.

ignore_certs

If set to *yes*, ansible-galaxy will not validate TLS certificates. Handy for testing against a server with a self-signed certificate .

BSD Support

Topics

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 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 simply execute the following for most versions of FreeBSD:

```
ansible -m raw -a "pkg install -y python27" mybsdhost1
```

Once this is done you can now use other Ansible modules apart from the `raw` module.

Note: This example used pkg as used on FreeBSD, 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
```

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!

See also:

Introduction To Ad-Hoc Commands Examples of basic commands

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

Windows Support

Topics

Windows: How Does It Work

As you may have already read, Ansible manages Linux/Unix machines using SSH by default.

Starting in version 1.7, Ansible also contains support for managing Windows machines. This uses native PowerShell remoting, rather than SSH.

Ansible will still be run from a Linux control machine, and uses the "winrm" Python module to talk to remote hosts.

No additional software needs to be installed on the remote machines for Ansible to manage them, it still maintains the agentless properties that make it popular on Linux/Unix.

Note that it is expected you have a basic understanding of Ansible prior to jumping into this section, so if you haven't written a Linux playbook first, it might be worthwhile to dig in there first.

Installing on the Control Machine

On a Linux control machine:

```
pip install "pywinrm>=0.1.1"
```

Note:: on distributions with multiple python versions, use pip2 or pip2.x, where x matches the python minor version Ansible is running under.

Active Directory Support

If you wish to connect to domain accounts published through Active Directory (as opposed to local accounts created on the remote host), you will need to install the "python-kerberos" module on the Ansible control host (and the MIT krb5 libraries it depends on). The Ansible control host also requires a properly configured computer account in Active Directory.

Installing python-kerberos dependencies

```
# Via Yum
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
```

Installing python-kerberos

Once you've installed the necessary dependencies, the python-kerberos wrapper can be installed via pip:

```
pip install kerberos requests_kerberos
```

Kerberos is installed and configured by default on OS X and many Linux distributions. If your control machine has not already done this for you, you will need to.

Configuring Kerberos

Edit your /etc/krb5.conf (which should be installed as a result of installing packages above) and add the following information for each domain you need to connect to:

In the section that starts with

```
[realms]
```

add the full domain name and the fully qualified domain names of your 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
 }
```

and in the [domain_realm] section add a line like the following for each domain you want to access:

```
[domain_realm]
    .my.domain.com = MY.DOMAIN.COM
```

You may wish to configure other settings here, such as the default domain.

Testing a kerberos connection

If you have installed krb5-workstation (yum) or krb5-user (apt-get) you can use the following command to test that you can be authorised by your domain controller.

```
kinit user@MY.DOMAIN.COM
```

Note that the domain part has to be fully qualified and must be in upper case.

To see what tickets if any you have acquired, use the command klist

```
klist
```

Troubleshooting kerberos connections

If you unable to connect using kerberos, check the following:

Ensure that forward and reverse DNS lookups are working properly on your domain.

To test this, ping the windows host you want to control by name then use the ip address returned with nslookup. You should get the same name back from DNS when you use nslookup on the ip address.

If you get different hostnames back than the name you originally pinged, speak to your active directory administrator and get them to check that DNS Scavenging is enabled and that DNS and DHCP are updating each other.

Ensure that the Ansible controller has a properly configured computer account in the domain.

Check your Ansible controller's clock is synchronised with your domain controller. Kerberos is time sensitive and a little clock drift can cause tickets not be granted.

Check you are using the real fully qualified domain name for the domain. Sometimes domains are commonly known to users by aliases. To check this run:

```
kinit -C user@MY.DOMAIN.COM
klist
```

If the domain name returned by klist is different from the domain name you requested, you are requesting using an alias, and you need to update your krb5.conf so you are using the fully qualified domain name, not its alias.

Inventory

Ansible's windows support relies on a few standard variables to indicate the username, password, and connection type (windows) of the remote hosts. These variables are most easily set up in inventory. This is used instead of SSH-keys or passwords as normally fed into Ansible:

```
[windows]
winserver1.example.com
winserver2.example.com
```

Note: Ansible 2.0 has deprecated the "ssh" from `ansible_ssh_user`, `ansible_ssh_host`, and `ansible_ssh_port` to become `ansible_user`, `ansible_host`, and `ansible_port`. If you are using a version of Ansible prior to 2.0, you should continue using the older style variables (`ansible_ssh_*`). These shorter variables are ignored, without warning, in older versions of Ansible.

In group_vars/windows.yml, define the following inventory variables:

```
# it is suggested that these be encrypted with ansible-vault:
# ansible-vault edit group_vars/windows.yml

ansible_user: Administrator
ansible_password: SecretPasswordGoesHere
ansible_port: 5986
ansible_connection: winrm
# The following is necessary for Python 2.7.9+ when using default WinRM self-
↪signed certificates:
ansible_winrm_server_cert_validation: ignore
```

Attention for the older style variables (`ansible_ssh_*`): ansible_ssh_password doesn't exist, should be ansible_ssh_pass.

Although Ansible is mostly an SSH-oriented system, Windows management will not happen over SSH (*yet <http://blogs.msdn.com/b/powershell/archive/2015/06/03/looking-forward-microsoft-support-for-secure-shell-ssh.aspx>*).

If you have installed the `kerberos` module and `ansible_user` contains `@` (e.g. `username@realm`), Ansible will first attempt Kerberos authentication. *This method uses the principal you are authenticated to Kerberos with on the control machine and not ''ansible_user''*. If that fails, either because you are not signed into Kerberos on the control machine or because the corresponding domain account on the remote host is not available, then Ansible will fall back to "plain" username/password authentication.

When using your playbook, don't forget to specify –ask-vault-pass to provide the password to unlock the file.

Test your configuration like so, by trying to contact your Windows nodes. Note this is not an ICMP ping, but tests the Ansible communication channel that leverages Windows remoting:

```
ansible windows [-i inventory] -m win_ping --ask-vault-pass
```

If you haven't done anything to prep your systems yet, this won't work yet. This is covered in a later section about how to enable PowerShell remoting - and if necessary - how to upgrade PowerShell to a version that is 3 or higher.

You'll run this command again later though, to make sure everything is working.

Since 2.0, the following custom inventory variables are also supported for additional configuration of WinRM connections:

```
* ``ansible_winrm_scheme``: Specify the connection scheme (``http`` or ``https``)␣
↪to use for the WinRM connection.  Ansible uses ``https`` by default unless the␣
↪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 the username contains ``@``, Ansible will use the part of the username after␣
↪``@`` by default.
* ``ansible_winrm_transport``: Specify one or more transports as a comma-separated␣
↪list.  By default, Ansible will use ``kerberos,plaintext`` if the ``kerberos`` ␣
↪module is installed and a realm is defined, otherwise ``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_*``: Any additional keyword arguments supported by ``winrm.
↪Protocol`` may be provided.
```

Windows System Prep

In order for Ansible to manage your windows machines, you will have to enable and configure PowerShell remoting.

To automate the setup of WinRM, you can run this PowerShell script on the remote machine.

The example script accepts a few arguments which Admins may choose to use to modify the default setup slightly, which might be appropriate in some cases.

Pass the -CertValidityDays option to customize the expiration date of the generated certificate.
powershell.exe -File ConfigureRemotingForAnsible.ps1 -CertValidityDays 100

Pass the -SkipNetworkProfileCheck switch to configure winrm to listen on PUBLIC zone interfaces. (Without this option, th
powershell.exe -File ConfigureRemotingForAnsible.ps1 -SkipNetworkProfileCheck

Pass the -ForceNewSSLCert switch to force a new SSL certificate to be attached to an already existing winrm listener. (Avoi
powershell.exe -File ConfigureRemotingForAnsible.ps1 -ForceNewSSLCert

---

Note: On Windows 7 and Server 2008 R2 machines, due to a bug in Windows Management Framework 3.0, it may be necessary to install this hotfix http://support.microsoft.com/kb/2842230 to avoid receiving out of memory and stack overflow exceptions. Newly-installed Server 2008 R2 systems which are not fully up to date with windows updates are known to have this issue.

Windows 8.1 and Server 2012 R2 are not affected by this issue as they come with Windows Management Framework 4.0.

Getting to PowerShell 3.0 or higher

PowerShell 3.0 or higher is needed for most provided Ansible modules for Windows, and is also required to run the above setup script. Note that PowerShell 3.0 is only supported on Windows 7 SP1, Windows Server 2008 SP1, and later releases of Windows.

Looking at an Ansible checkout, copy the examples/scripts/upgrade_to_ps3.ps1 script onto the remote host and run a PowerShell console as an administrator. You will now be running PowerShell 3 and can try connectivity again using the win_ping technique referenced above.

What modules are available

Most of the Ansible modules in core Ansible are written for a combination of Linux/Unix machines and arbitrary web services, though there are various Windows modules as listed in the "windows" subcategory of the Ansible module index.

Browse this index to see what is available.

In many cases, it may not be necessary to even write or use an Ansible module.

In particular, the "script" module can be used to run arbitrary PowerShell scripts, allowing Windows administrators familiar with PowerShell a very native way to do things, as in the following playbook:

```
- hosts: windows
  tasks:
    - script: foo.ps1 --argument --other-argument
```

Note:: There are a few other Ansible modules that don't start with "win" that also function with Windows, including "fetch", "slurp", "raw", and "setup" (which is how fact gathering works).

Developers: Supported modules and how it works

Developing Ansible modules are covered in a later section of the documentation, with a focus on Linux/Unix. What if you want to write Windows modules for Ansible though?

For Windows, Ansible modules are implemented in PowerShell. Skim those Linux/Unix module development chapters before proceeding. Windows modules in the core and extras repo live in a "windows/" subdir. Custom modules can go directly into the Ansible "library/" directories or those added in ansible.cfg. Documentation lives in a *.py* file with the same name. For example, if a module is named "win_ping", there will be embedded documentation in the "win_ping.py" file, and the actual PowerShell code will live in a "win_ping.ps1" file. Take a look at the sources and this will make more sense.

Modules (ps1 files) should start as follows:

```
#!powershell
# <license>

# WANT_JSON
# POWERSHELL_COMMON

# code goes here, reading in stdin as JSON and outputting JSON
```

The above magic is necessary to tell Ansible to mix in some common code and also know how to push modules out. The common code contains some nice wrappers around working with hash data structures and emitting JSON results, and possibly a few more useful things. Regular Ansible has this same concept for reusing Python code - this is just the windows equivalent.

What modules you see in windows/ are just a start. Additional modules may be submitted as pull requests to github.

Reminder: You Must Have a Linux Control Machine

Note running Ansible from a Windows control machine is NOT a goal of the project. Refrain from asking for this feature, as it limits what technologies, features, and code we can use in the main project in the future. A Linux control machine will be required to manage Windows hosts.

Cygwin is not supported, so please do not ask questions about Ansible running from Cygwin.

Windows Facts

Just as with Linux/Unix, facts can be gathered for windows hosts, which will return things such as the operating system version. To see what variables are available about a windows host, run the following:

```
ansible winhost.example.com -m setup
```

Note that this command invocation is exactly the same as the Linux/Unix equivalent.

Windows Playbook Examples

Look to the list of windows modules for most of what is possible, though also some modules like "raw" and "script" also work on Windows, as do "fetch" and "slurp".

Here is an example of pushing and running a PowerShell script:

```
- name: test script module
  hosts: windows
  tasks:
    - name: run test script
      script: files/test_script.ps1
```

Running individual commands uses the 'raw' module, as opposed to the shell or command module as is common on Linux/Unix operating systems:

```
- name: test raw module
  hosts: windows
  tasks:
    - name: run ipconfig
      raw: ipconfig
      register: ipconfig
    - debug: var=ipconfig
```

Running common DOS commands like 'del", 'move', or 'copy" is unlikely to work on a remote Windows Server using Powershell, but they can work by prefacing the commands with "CMD /C" and enclosing the command in double quotes as in this example:

```
- name: another raw module example
  hosts: windows
  tasks:
     - name: Move file on remote Windows Server from one location to another
       raw: CMD /C "MOVE /Y C:\teststuff\myfile.conf C:\builds\smtp.conf"
```

You may wind up with a more readable playbook by using the PowerShell equivalents of DOS commands. For example, to achieve the same effect as the example above, you could use:

```
- name: another raw module example demonstrating powershell one liner
  hosts: windows
  tasks:
     - name: Move file on remote Windows Server from one location to another
       raw: Move-Item C:\teststuff\myfile.conf C:\builds\smtp.conf
```

Bear in mind that using C(raw) will allways report "changed", and it is your responsiblity to ensure PowerShell will need to handle idempotency as appropriate (the move examples above are inherently not idempotent), so where possible use (or write) a module.

Here's an example of how to use the win_stat module to test for file existence. Note that the data returned by the win_stat module is slightly different than what is provided by the Linux equivalent:

```
- name: test stat module
  hosts: windows
```

```
tasks:
  - name: test stat module on file
    win_stat: path="C:/Windows/win.ini"
    register: stat_file

  - debug: var=stat_file

  - name: check stat_file result
    assert:
        that:
           - "stat_file.stat.exists"
           - "not stat_file.stat.isdir"
           - "stat_file.stat.size > 0"
           - "stat_file.stat.md5"
```

Again, recall that the Windows modules are all listed in the Windows category of modules, with the exception that the "raw", "script", "slurp" and "fetch" modules are also available. These modules do not start with a "win" prefix.

Windows Contributions

Windows support in Ansible is still relatively new, and contributions are quite welcome, whether this is in the form of new modules, tweaks to existing modules, documentation, or something else. Please stop by the ansible-devel mailing list if you would like to get involved and say hi.

See also:

developing_modules How to write modules

Playbooks Learning Ansible's configuration management language

List of Windows Modules Windows specific module list, all implemented in PowerShell

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

irc.freenode.net #ansible IRC chat channel

Networking Support

Topics

Working with Networking Devices

Starting with Ansible version 2.1, you can now use the familiar Ansible models of playbook authoring and module development to manage heterogenous networking devices. Ansible supports a growing number of network devices using both CLI over SSH and API (when available) transports.

Network Automation Installation

- Install the latest Ansible network release.
- Get the playbooks for testing Ansible core network modules.

Available Networking Modules

Most standard Ansible modules are designed to work with Linux/Unix or Windows machines and will not work with networking devices. Some modules (including "slurp", "raw", and "setup") are platform-agnostic and will work with networking devices.

To see what modules are available for networking devices, please browse the "networking" section of the Ansible module index.

Connecting to Networking Devices

All core networking modules implement a *provider* argument, which is a collection of arguments used to define the characteristics of how to connect to the device. This section will assist in understanding how the provider argument is used.

Each core network module supports an underlying operating system and transport. The operating system is a one-to-one match with the module, and the transport maintains a one-to-many relationship to the operating system as appropriate. Some network operating systems only have a single transport option.

Each core network module supports some basic arguments for configuring the transport:

- host - defines the hostname or IP address of the remote host
- port - defines the port to connect to
- username - defines the username to use to authenticate the connection
- password - defines the password to use to authenticate the connection
- transport - defines the type of connection transport to build
- authorize - enables privilege escalation for devices that require it
- auth_pass - defines the password, if needed, for privilege escalation

Individual modules can set defaults for these arguments to common values that match device default configuration settings. For instance, the default value for transport is universally 'cli'. Some modules support other values such as EOS (eapi) and NXOS (nxapi), while some only support 'cli'. All arguments are fully documented for each module.

By allowing individual tasks to set the transport arguments independently, modules that use different transport mechanisms and authentication credentials can be combined as necessary.

One downside to this approach is that every task needs to include the required arguments. This is where the provider argument comes into play. The provider argument accepts keyword arguments and passes them through to the task to assign connection and authentication parameters.

The following two config modules are essentially identical (using nxos_config) as an example but it applies to all core networking modules:

```
---
nxos_config:
   src: config.j2
   host: "{{ inventory_hostname }}"
   username: "{{ ansible_ssh_user }}"
   password: "{{ ansible_ssh_pass }}"
   transport: cli

---
```

```
vars:
   cli:
   host: "{{ inventory_hostname }}"
   username: "{{ ansible_ssh_user }}"
   password: "{{ ansible_ssh_pass }} "
   transport: cli


nxos_config:
   src: config.j2
   provider: "{{ cli }}"
```

Given the above two examples that are equivalent, the arguments can also be used to establish precedence and defaults. Consider the following example:

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

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

In this example, the values of admin for username and admin for password will override the values of operator in cli['username'] and secret in cli['password'])

This is true for all values in the provider including transport. So you could have a singular task that is now supported over CLI or NXAPI (assuming the configuration is value).

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

tasks:
  - nxos_config:
      src: config.j2
      provider: "{{ cli }}"
      transport: nxapi
```

If all values are provided via the provider argument, the rules for requirements are still honored for the module. For instance, take the following scenario:

```
---
vars:
  conn:
  password: cisco_pass
  transport: cli

tasks:
- nxos_config:
  src: config.j2
  provider: "{{ conn }}"
```

Running the above task will cause an error to be generated with a message that required parameters are missing.

```
"msg": "missing required arguments: username,host"
```

Overall, this provides a very granular level of control over how credentials are used with modules. It provides the playbook designer maximum control for changing context during a playbook run as needed.

Networking Environment Variables

The following environment variables are available to Ansible networking modules:

username ANSIBLE_NET_USERNAME password ANSIBLE_NET_PASSWORD ssh_keyfile ANSIBLE_NET_SSH_KEYFILE authorize ANSIBLE_NET_AUTHORIZE auth_pass ANSIBLE_NET_AUTH_PASS

Variables are evaulated in the following order, listed from lowest to highest priority:

- Default
- Environment
- Provider
- Task arguments

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 evalute the results from the commands that are executed remotely on the device. Once the task executes the command set, the waitfor argument can be used to evalute 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
      waitfor:
          - "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
      waitfor:
          - "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 waitfor 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.

CHAPTER

TWO

QUICKSTART VIDEO

We've recorded a short video that shows how to get started with Ansible that you may like to use alongside the documentation.

The quickstart video is about 13 minutes long and will show you some of the basics about your first steps with Ansible.

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

CHAPTER

THREE

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.

It is recommended to 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 *About 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 (and enable it at boot)
    service: name=httpd state=started enabled=yes
  handlers:
    - name: restart apache
      service: name=httpd state=restarted
```

When working with tasks that have really long parameters or modules that take many parameters, you can break tasks items over multiple lines to improve the structure. Below is another version of the above example but using YAML dictionaries to supply the modules with their `key=value` arguments.:

```
---
- 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:
  - 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 *Patterns* documentation. The `remote_user` is just the name of the user account:

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

Note: 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
```

Note: The `remote_user` parameter for tasks was added in 1.4.

Support for running things as another user is also available (see *Become (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
  remote_user: yourname
  tasks:
    - service: name=nginx state=started
      become: yes
      become_method: sudo
```

Note: The become syntax deprecates the old sudo/su specific syntax beginning in 1.9.

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

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

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

```
---
- hosts: webservers
  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.

Important: 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.

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 are 'idempotent', meaning if you run them again, they will make only the changes they must in order to bring the system to the desired state. This makes it very safe to rerun the same playbook multiple times. They won't change things unless they have to change things.

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 have 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: disable selinux
    command: /sbin/setenforce 0
```

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 using the `include:` directive. We'll show that a bit later.

Action Shorthand

New in version 0.8.

Ansible prefers listing modules like this in 0.8 and later:

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

You will notice in earlier versions, this was only available as:

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

The old form continues to work in newer versions without any plan of deprecation.

Handlers: Running Operations On Change

As we've mentioned, modules are written to 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).

Note:

- 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 though, in 1.2 and later, you can:

```
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

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.

Ansible playbook output is vastly upgraded if the cowsay package is installed. Try it!

To see what hosts would be affected by a playbook before you run it, you can do this:

```
ansible-playbook playbook.yml --list-hosts
```

See also:

YAML Syntax Learn about YAML syntax

Best Practices Various tips about managing playbooks in the real world

Ansible Documentation Hop back to the documentation index for a lot of special topics about playbooks

About Modules Learn about available modules

developing_modules Learn how to extend Ansible by writing your own modules

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

Playbook Roles and Include Statements

Topics

Introduction

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.

At a basic level, including task files allows you to break up bits of configuration policy into smaller files. Task includes pull in tasks from other files. Since handlers are tasks too, you can also include handler files from the 'handler' section.

See *Playbooks* if you need a review of these concepts.

Playbooks can also include plays from other playbook files. When that is done, the plays will be inserted into the playbook to form a longer list of plays.

When you start to think about it – tasks, handlers, variables, and so on – begin to form larger concepts. You start to think about modeling what something is, rather than how to make something look like something. It's no longer "apply this handful of THINGS to these hosts", you say "these hosts are dbservers" or "these hosts are webservers". In programming, we might call that "encapsulating" how things work. For instance, you can drive a car without knowing how the engine works.

Roles in Ansible build on the idea of include files and combine them to form clean, reusable abstractions – they allow you to focus more on the big picture and only dive down into the details when needed.

We'll start with understanding includes so roles make more sense, but our ultimate goal should be understanding roles – roles are great and you should use them every time you write playbooks.

See the ansible-examples repository on GitHub for lots of examples of all of this put together. You may wish to have this open in a separate tab as you dive in.

Task Include Files And Encouraging Reuse

Suppose you want to reuse lists of tasks between plays or playbooks. You can use include files to do this. Use of included task lists is a great way to define a role that system is going to fulfill. Remember, the goal of a play in a playbook is to map a group of systems into multiple roles. Let's see what this looks like...

A task include file simply contains a flat list of tasks, like so:

```
---
# possibly saved as tasks/foo.yml

- name: placeholder foo
  command: /bin/foo

- name: placeholder bar
  command: /bin/bar
```

Include directives look like this, and can be mixed in with regular tasks in a playbook:

```
tasks:

  - include: tasks/foo.yml
```

You can also pass variables into includes. We call this a 'parameterized include'.

For instance, to deploy to multiple wordpress instances, I could encapsulate all of my wordpress tasks in a single wordpress.yml file, and use it like so:

```
tasks:
  - include: wordpress.yml wp_user=timmy
  - include: wordpress.yml wp_user=alice
  - include: wordpress.yml wp_user=bob
```

Starting in 1.0, variables can also be passed to include files using an alternative syntax, which also supports structured variables:

```
tasks:

  - include: wordpress.yml
```

```
    vars:
        wp_user: timmy
        ssh_keys:
          - keys/one.txt
          - keys/two.txt
```

Using either syntax, variables passed in can then be used in the included files. We'll cover them in *Variables*. You can reference them like this:

```
{{ wp_user }}
```

(In addition to the explicitly passed-in parameters, all variables from the vars section are also available for use here as well.)

Playbooks can include other playbooks too, but that's mentioned in a later section.

Note: As of 1.0, task include statements can be used at arbitrary depth. They were previously limited to a single level, so task includes could not include other files containing task includes.

Includes 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:

```
---
# this might be in a file like handlers/handlers.yml
- name: restart apache
  service: name=apache state=restarted
```

And in your main playbook file, just include it like so, at the bottom of a play:

```
handlers:
  - include: handlers/handlers.yml
```

You can mix in includes along with your regular non-included tasks and handlers.

Includes can also be used to import one playbook file into another. This allows you to define a top-level playbook that is composed of other playbooks.

For example:

```
- name: this is a play at the top level of a file
  hosts: all
  remote_user: root

  tasks:

  - name: say hi
    tags: foo
    shell: echo "hi..."

- include: load_balancers.yml
- include: webservers.yml
- include: dbservers.yml
```

Note that you cannot do variable substitution when including one playbook inside another.

Note: You can not conditionally pass the location to an include file, like you can with 'vars_files'. If you find yourself needing to do this, consider how you can restructure your playbook to be more class/role oriented. This is to say you cannot use a 'fact' to decide what include file to use. All hosts contained within the play are going to get the same tasks. ('*when*' provides some ability for hosts to conditionally skip tasks).

Dynamic versus Static Includes

Ansible 2.0 changes how include tasks are processed. In previous versions of Ansible, includes acted as a pre-processor statement and were read during playbook parsing time. This created problems with things like inventory variables (like group and host vars, which are not available during the parsing time) were used in the included file name.

Ansible 2.0 instead makes includes "dynamic", meaning they are not evaluated until the include task is reached during the play execution. This change allows the reintroduction of loops on include statements, such as the following:

```
- include: foo.yml param={{item}}
  with_items:
  - 1
  - 2
  - 3
```

It is also possible to use variables from any source with a dynamic include:

```
- include: "{{inventory_hostname}}.yml"
```

Note: When an include statement loads different tasks for different hosts, the `linear` strategy keeps the hosts in lock-step by alternating which hosts are executing tasks while doing a `noop` for all other hosts. For example, if you had hostA, hostB and hostC with the above example, hostA would execute all of the tasks in hostA.yml while hostB and hostC waited. It is generally better to do the above with the `free` strategy, which does not force hosts to execute in lock-step.

Dynamic includes introduced some other limitations due to the fact that the included file is not read in until that task is reached during the execution of the play. When using dynamic includes, it is important to keep these limitations in mind:

- You cannot use `notify` to trigger a handler name which comes from a dynamic include.
- You cannot use `--start-at-task` to begin execution at a task inside a dynamic include.
- 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.

Note: In Ansible 1.9.x and earlier, an error would be raised if a tag name was used with `--tags` or `--skip-tags`. This error was disabled in Ansible 2.0 to prevent incorrect failures with tags which only existed inside of dynamic includes.

To work around these limitations, Ansible 2.1 introduces the `static` option for includes:

```
- include: foo.yml
  static: <yes|no|true|false>
```

By default in Ansible 2.1 and higher, includes are automatically treated as static rather than dynamic when the include meets the following conditions:

- The include does not use any loops
- The included file name does not use any variables
- The `static` option is not explicitly disabled (ie. `static: no`)
- The ansible.cfg options to force static includes (see below) are disabled

Two options are available in the ansible.cfg configuration for static includes:

- `task_includes_static` - forces all includes in tasks sections to be static.

- `handler_includes_static` - forces all includes in handlers sections to be static.

These options allow users to force playbooks to behave exactly as they did in 1.9.x and before.

Roles

New in version 1.2.

Now that you have learned about tasks and handlers, what is the best way to organize your playbooks? The short answer is to use roles! 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.

Roles are just automation around 'include' directives as described above, and really don't contain much additional magic beyond some improvements to search path handling for referenced files. However, that can be a big thing!

Example project structure:

```
site.yml
webservers.yml
fooservers.yml
roles/
   common/
     files/
     templates/
     tasks/
     handlers/
     vars/
     defaults/
     meta/
   webservers/
     files/
     templates/
     tasks/
     handlers/
     vars/
     defaults/
     meta/
```

In a playbook, it would look like this:

```
---
- 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

In Ansible 1.4 and later you can configure a 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 *Configuration file* for details about how to set this up in ansible.cfg.

Note: Role dependencies are discussed below.

If any files are not present, they are just ignored. So it's ok to not have a 'vars/' subdirectory for the role, for instance.

Note, you are still allowed to list tasks, vars_files, and handlers "loose" in playbooks without using roles, but roles are a good organizational feature and are highly recommended. If there are loose things in the playbook, the roles are evaluated first.

Also, should you wish to parameterize roles, by adding variables, you can do so, like this:

```
---

- hosts: webservers
  roles:
    - common
    - { role: foo_app_instance, dir: '/opt/a',  app_port: 5000 }
    - { role: foo_app_instance, dir: '/opt/b',  app_port: 5001 }
```

While it's probably not something you should do often, you can also conditionally apply roles like so:

```
---

- hosts: webservers
  roles:
    - { role: some_role, when: "ansible_os_family == 'RedHat'" }
```

This works by applying the conditional to every task in the role. Conditionals are covered later on in the documentation.

Finally, you may wish to assign tags to the roles you specify. You can do so inline:

```
---

- hosts: webservers
  roles:
    - { role: foo, tags: ["bar", "baz"] }
```

Note that this *tags all of the tasks in that role with the tags specified*, overriding any tags that are specified inside the role. 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.

If the play still has a 'tasks' section, those tasks are executed after roles are applied.

If you want to define certain tasks to happen before AND after roles are applied, you can do this:

```
---

- hosts: webservers

  pre_tasks:
    - shell: echo 'hello'

  roles:
    - { role: some_role }

  tasks:
    - shell: echo 'still busy'

  post_tasks:
    - shell: echo 'goodbye'
```

Note: 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 and post_tasks and pass those along as well, especially if the pre and post tasks are used for monitoring outage window control or load balancing.

Role Default Variables

New in version 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

New in version 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. 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, some_parameter: 3 }
  - { role: apache, apache_port: 80 }
  - { role: postgres, dbname: blarg, other_parameter: 12 }
```

Role dependencies can also be specified as a full path, just like top level roles:

```
---
dependencies:
   - { role: '/path/to/common/roles/foo', x: 1 }
```

Role dependencies can also be installed from source control repos or tar files (via *galaxy*) using comma separated format of path, an optional version (tag, commit, branch etc) and optional friendly role name (an attempt is made to derive a role name from the repo name or archive filename). Both through the command line or via a requirements.yml passed to ansible-galaxy.

Roles dependencies are always executed before the role that includes them, and are recursive. By default, roles can also only be added as a dependency once - if another role also lists it as a dependency it will not be run again. This behavior can be overridden by adding *allow_duplicates: yes* to the *meta/main.yml* file. For example, a role named 'car' could add a role named 'wheel' to its dependencies as follows:

```
---
dependencies:
- { role: wheel, n: 1 }
- { role: wheel, n: 2 }
- { role: wheel, n: 3 }
- { role: wheel, n: 4 }
```

And the *meta/main.yml* for wheel contained the following:

```
---
allow_duplicates: yes
dependencies:
- { role: tire }
- { role: brake }
```

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: 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
    - 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'

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.

You can sign up with social auth, and the download client 'ansible-galaxy' is included in Ansible 1.4.2 and later.

Read the "About" page on the Galaxy site for more information.

See also:

Ansible Galaxy How to share roles on galaxy, role management

YAML Syntax Learn about YAML syntax

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

About 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

Variables

Topics

While automation exists to make it easier to make things repeatable, all of your systems are likely not exactly alike.

On some systems you may want to set some behavior or configuration that is slightly different from others.

Also, some of the observed behavior or state of remote systems might need to influence how you configure those systems. (Such as you might need to find out the IP address of a system and even use it as a configuration value on another system).

You might have some templates for configuration files that are mostly the same, but slightly different based on those variables.

Variables in Ansible are how we deal with differences between systems.

To understand variables you'll also want to dig into *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.

It's highly recommended that you consult the ansible-examples github repository to see a lot of examples of variables put to use.

For best practices advice, 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 *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 *Playbook Roles and Include Statements*, 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 instance, 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.

Note: 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.

Jinja2 Filters

Note: 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 *Jinja2 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* page.

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 ginormous amount of variable data, which may look like this, as taken from Ansible 1.4 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": {},
        "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": "NECVMWar"
    }
},
"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": [
        {
            "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",
"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 *About 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)

New in version 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?

For instance, what if you want users to be able to control some aspect about how their systems are managed? "Facts.d" is one such mechanism.

Note: 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.

For instance assume a `/etc/ansible/facts.d/preferences.fact`:

```
[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.

Note: 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 impi 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

New in 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

New in version 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?

Imagine, for instance, a very large infrastructure with thousands of hosts. Fact caching could be configured to run nightly, but 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 a 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 playbook_loops section on using register with a loop.

Note: 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 talked about 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.

New in version 2.2.

`ansible_play_hosts` is the full list of all hosts still active in the current play. .. versionadded:: 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`).

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
  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
```

Note: 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 send variables over the Ansible command line. This is particularly useful when writing a generic release playbook where you may want to pass in the version of the application to deploy:

```
ansible-playbook release.yml --extra-vars "version=1.23.45 other_variable=foo"
```

This is useful, for, among other things, setting the hosts group or the user for the playbook.

Example:

```
---

- hosts: '{{ hosts }}'
  remote_user: '{{ user }}'

  tasks:
     - ...

ansible-playbook release.yml --extra-vars "hosts=vipers user=starbuck"
```

As of Ansible 1.2, you can also pass in extra vars as quoted JSON, like so:

```
--extra-vars '{"pacman":"mrs","ghosts":["inky","pinky","clyde","sue"]}'
```

The `key=value` form is obviously simpler, but it's there if you need it!

Note: 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).

As of Ansible 1.3, extra vars can be loaded from a JSON file with the `@` syntax:

```
--extra-vars "@some_file.json"
```

Also as of Ansible 1.3, extra vars can be formatted as YAML, either on the command line or in a file as above.

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.

Note: Ansible 2.0 has deprecated the "ssh" from `ansible_ssh_user`, `ansible_ssh_host`, and `ansible_ssh_port` to become `ansible_user`, `ansible_host`, and `ansible_port`. If you are using a version of Ansible prior to 2.0, you should continue using the older style variables (`ansible_ssh_*`). These shorter variables are ignored, without warning, in older versions of Ansible.

In 1.x, the precedence is as follows (with the last listed variables winning prioritization):

- "role defaults", which lose in priority to everything and are the most easily overridden
- variables defined in inventory
- facts discovered about a system
- "most everything else" (command line switches, vars in play, included vars, role vars, etc.)
- connection variables (`ansible_user`, etc.)
- extra vars (`-e` in the command line) always win

Note: In versions prior to 1.5.4, facts discovered about a system were in the "most everything else" category above.

In 2.x, we have made the order of precedence more specific (with the last listed variables winning prioritization):

- role defaults [1]
- inventory vars [2]
- inventory group_vars
- inventory host_vars
- playbook group_vars
- playbook host_vars
- host facts
- play vars
- play vars_prompt
- play vars_files
- registered vars
- set_facts
- role and include vars
- block vars (only for tasks in block)
- task vars (only for the task)
- 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.

[1] Tasks in each role will see their own role's defaults. Tasks defined outside of a role will see the last role's defaults.
[2] Variables defined in inventory file or provided by dynamic inventory.

Note: 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 2nd one wins.

Note: the previous describes the default config *hash_behavior=replace*, switch to 'merge' to only partially overwrite.

Another important thing to consider (for all versions) is that connection specific variables override config, command line and play specific options and directives. For example:

```
ansible_ssh_user will override `-u <user>` and `remote_user: <user>`
```

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, so if you really want to override the remote user globally even over inventory you can use extra vars:

```
ansible... -e "ansible_ssh_user=<user>"
```

Variable Scopes

Ansible has 3 main scopes:

- Global: this is set by config, environment variables and the command line
- Play: each play and contained structures, vars entries, include_vars, role defaults and vars.
- Host: variables directly associated to a host, like inventory, facts or registered task outputs

Variable Examples

That seems a little theoretical. 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 super 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 *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.

Ok, so 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. It's just a default. That's why it says "defaults" :) See *Playbook Roles and Include Statements* 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
```

So the above is a great 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, 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, like so:

```
roles:
   - { role: app_user, name: Ian    }
   - { role: app_user, name: Terry  }
   - { role: app_user, name: Graham }
   - { role: app_user, name: John   }
```

That's a bit arbitrary, but you can see how the same role was invoked multiple times. In that example it's quite likely there was no default for 'name' supplied at all. Ansible can yell at you when variables aren't defined – it's the default behavior in fact.

So that's a bit about roles.

There are a few bonus 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, foo: 12 }
   - { role: something_else }
```

Note: 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*.

See also:

Playbooks An introduction to playbooks

Conditionals Conditional statements in playbooks

Jinja2 filters Jinja2 filters and their uses

Loops Looping in playbooks

Playbook Roles and Include Statements 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

Jinja2 filters

Topics

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.

Take into account that filters always execute on the Ansible controller, **not** on the task target, as they manipulate local data.

In addition to those, Ansible supplies many more.

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.

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)}}
  with_items:
    - 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.

Note: 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.

New in version 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 }}
```

Set Theory Filters

All these functions return a unique set from sets or lists.

New in version 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) }}
```

Random Number Filter

New in version 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 from 0 to supplied end:

```
{{ 59 |random}} * * * * root /script/from/cron
```

Get a random number from 0 to 100 but in steps of 10:

```
{{ 100 |random(step=10) }}  => 70
```

Get a random number from 1 to 100 but in steps of 10:

```
{{ 100 |random(1, 10) }}    => 31
{{ 100 |random(start=1, step=10) }}    => 51
```

Shuffle Filter

New in version 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']
```

note that when used with a non 'listable' item it is a noop, otherwise it always returns a list

Math

New in version 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().

IP address filter

New in version 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.

Hashing filters

New in version 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') }}
```

Hash types available depend on the master system running ansible, 'hash' depends on hashlib password_hash depends on crypt.

Combining hashes/dictionaries

New in version 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

New in version 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

New in version 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') }}
```

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
#
```

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 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 }}
```

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
```

New in version 1.6.

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>') }}
```

Note: 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 (\\\\) instead of 2 (\\).

New in version 2.0.

To escape special characters within a regex, use the "regex_escape" filter:

```
# convert '^f.*o(.*)$' to '\^f\.\*o\(\.\*\)\$'
{{ '^f.*o(.*)$' | regex_escape() }}
```

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 amount of seconds between two dates, default date format is %Y-%d-%m %H:%M:%S but you can pass your own one {{ (("2016-08-04 20:00:12"|to_datetime) - ("2015-10-06"|to_datetime('%Y-%d-%m'))).seconds }}

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.

See also:

Playbooks An introduction to playbooks

Conditionals Conditional statements in playbooks

Variables All about variables

Loops Looping in playbooks

Playbook Roles and Include Statements 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

Jinja2 tests

Topics

Tests in Jinja2 are a way of evaluating template expressions and returning True or False. Jinja2 ships with many of these. See builtin tests in the official Jinja2 template documentation. Tests are very similar to filters and are used mostly the same way, but they can also be used in list processing filters, like C(map()) and C(select()) to choose items in the list.

Like filters, 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.

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:
    - shell: "msg='matched pattern 1'"
      when: url | match("http://example.com/users/.*/resources/.*")

    - debug: "msg='matched pattern 2'"
      when: url | search("/users/.*/resources/.*")

    - debug: "msg='matched pattern 3'"
      when: url | search("/users/")
```

'match' requires a complete match in the string, while 'search' only requires matching a subset of the string.

Version Comparison

New in version 1.6.

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_compare` filter.

The `version_compare` filter can also be used to evaluate the `ansible_distribution_version`:

```
{{ ansible_distribution_version | version_compare('12.04', '>=') }}
```

If `ansible_distribution_version` is greater than or equal to 12, this filter returns True, otherwise False.

The `version_compare` filter 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 | version_compare('1.0', operator='lt', strict=True) }}
```

Group theory tests

To see if a list includes or is included by another list, you can use 'issubset' and 'issuperset':

```
vars:
    a: [1,2,3,4,5]
    b: [2,3]
tasks:
    - debug: msg="A includes B"
      when: a|issuperset(b)

    - debug: msg="B is included in A"
      when: b|issubset(a)
```

Testing paths

The following tests can provide information about a path on the controller:

```
- debug: msg="path is a directory"
  when: mypath|isdir

- 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|exists

- debug: msg="path is {{ (mypath|is_abs)|ternary('absolute','relative')}}"

- debug: msg="path is the same file as path2"
  when: mypath|samefile(path2)

- debug: msg="path is a mount"
  when: mypath|ismount
```

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|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|changed

  - debug: msg="it succeeded in Ansible >= 2.1"
    when: result|succeeded

  - debug: msg="it succeeded"
    when: result|success

  - debug: msg="it was skipped"
    when: result|skipped
```

Note: 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.

See also:

Playbooks An introduction to playbooks

Conditionals Conditional statements in playbooks

Variables All about variables

Loops Looping in playbooks

Playbook Roles and Include Statements 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

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. Further, additional groups can be created to manage hosts based on whether the hosts match other criteria. There are many options to control execution flow in Ansible.

Let's dig into what they are.

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|failed

  # In older versions of ansible use |success, now both are valid but succeeded␣
↪uses the correct tense.
  - command: /bin/something_else
    when: result|succeeded

  - command: /bin/still/something_else
    when: result|skipped
```

Note: the filters have been updated in 2.1 so both *success* and *succeeded* work (*fail/failed*, etc).

Note that was a little bit of foreshadowing on the 'register' statement. We'll get to it a bit later in this chapter.

As a reminder, to see what facts are available on a particular system, you can do:

```
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
```

Note: 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).

Loops and Conditionals

Combining *when* with *with_items* (see *Loops*), be aware that the *when* statement is processed separately for each item. This is by design:

```
tasks:
    - command: echo {{ item }}
      with_items: [ 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 }}
  with_items: "{{ mylist|default([]) }}"
  when: item > 5
```

If using *with_dict* which does not take a list:

```
- command: echo {{ item.key }}
  with_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 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:

```
- include: tasks/sometasks.yml
  when: "'reticulating splines' in output"
```

Note: 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 *About Modules* docs for a more streamlined way to accomplish the same thing.

Conditional Imports

Note: This is an advanced topic that is infrequently used. You can probably skip this section.

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 running
    service: name={{ apache }} state=running
```

Note: 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/CentOS.yml
apache: httpd
somethingelse: 42
```

How does this work? If the operating system was 'CentOS', the first file Ansible would try to import would be 'vars/CentOS.yml', followed by '/vars/os_defaults.yml' if that file did not exist. If no files in the list were found, an error would be raised. On Debian, it would instead first look towards 'vars/Debian.yml' instead of 'vars/CentOS.yml', before falling back on 'vars/os_defaults.yml'. Pretty simple.

To use this conditional import feature, you'll need facter or ohai installed prior to running the playbook, but you can of course push this out with Ansible if you like:

```
# for facter
ansible -m yum -a "pkg=facter state=present"
ansible -m yum -a "pkg=ruby-json state=present"

# for ohai
ansible -m yum -a "pkg=ohai state=present"
```

Ansible's approach to configuration – separating variables from tasks, keeps your playbooks from turning into arbitrary code with ugly nested ifs, conditionals, and so on - and results in more streamlined & auditable configuration rules – especially because there are a minimum of decision points to track.

Selecting Files And Templates Based On Variables

Note: 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
  with_first_found:
    - files:
       - {{ ansible_distribution }}.conf
       - default.conf
      paths:
       - 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 "with_items" 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 with_items 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
        with_items: "{{ home_dirs.stdout_lines }}"
        # same as with_items: "{{ 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 == ""
```

See also:

Playbooks An introduction to playbooks

Playbook Roles and Include Statements 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

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
  with_items:
```

```
    - testuser1
    - testuser2
```

If you have defined a YAML list in a variables file, or the 'vars' section, you can also do:

```
with_items: "{{ 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
```

The yum and apt modules use with_items to execute fewer package manager transactions.

Note that the types of items you iterate over with 'with_items' 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 }}
  with_items:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

Also be aware that when combining *when* with *with_items* (or any other loop statement), the *when* statement is processed separately for each item. See *The When Statement* for an example.

Loops are actually a combination of things *with_* + *lookup()*, so any lookup plugin can be used as a source for a loop, 'items' is lookup.

Nested Loops

Loops can be nested as well:

```
- name: give users access to multiple databases
  mysql_user: name={{ item[0] }} priv={{ item[1] }}.*:ALL append_privs=yes␣
↪password=foo
  with_nested:
    - [ 'alice', 'bob' ]
    - [ 'clientdb', 'employeedb', 'providerdb' ]
```

As with the case of 'with_items' above, you can use previously defined variables.:

```
- name: here, 'users' contains the above list of employees
  mysql_user: name={{ item[0] }} priv={{ item[1] }}.*:ALL append_privs=yes␣
↪password=foo
  with_nested:
    - "{{ users }}"
    - [ 'clientdb', 'employeedb', 'providerdb' ]
```

Looping over Hashes

New in version 1.5.

Suppose you have the following variable:

```
---
users:
  alice:
```

```
    name: Alice Appleworth
    telephone: 123-456-7890
  bob:
    name: Bob Bananarama
    telephone: 987-654-3210
```

And you want to print every user's name and phone number. You can loop through the elements of a hash using `with_dict` like this:

```
tasks:
  - name: Print phone records
    debug: msg="User {{ item.key }} is {{ item.value.name }} ({{ item.value.
↪telephone }})"
    with_dict: "{{ users }}"
```

Looping over Files

`with_file` iterates over the content of a list of files, *item* will be set to the content of each file in sequence. It can be used like this:

```
---
- hosts: all

  tasks:

    # emit a debug message containing the content of each file.
    - debug:
        msg: "{{ item }}"
      with_file:
        - first_example_file
        - second_example_file
```

Assuming that `first_example_file` contained the text "hello" and `second_example_file` contained the text "world", this would result in:

```
TASK [debug msg={{ item }}] ***************************************************
ok: [localhost] => (item=hello) => {
    "item": "hello",
    "msg": "hello"
}
ok: [localhost] => (item=world) => {
    "item": "world",
    "msg": "world"
}
```

Looping over Fileglobs

`with_fileglob` matches all files in a single directory, non-recursively, that match a pattern. It calls Python's glob library, and can be used like this:

```
---
- hosts: all

  tasks:

    # first ensure our target directory exists
    - file: dest=/etc/fooapp state=directory

    # copy each file over that matches the given pattern
```

```
    - copy: src={{ item }} dest=/etc/fooapp/ owner=root mode=600
      with_fileglob:
        - /playbooks/files/fooapp/*
```

Note: When using a relative path with `with_fileglob` in a role, Ansible resolves the path relative to the *roles/<rolename>/files* directory.

Looping over Parallel Sets of Data

Note: This is an uncommon thing to want to do, but we're documenting it for completeness. You probably won't be reaching for this one often.

Suppose you have the following variable data was loaded in via somewhere:

```
---
alpha: [ 'a', 'b', 'c', 'd' ]
numbers:  [ 1, 2, 3, 4 ]
```

And you want the set of '(a, 1)' and '(b, 2)' and so on. Use 'with_together' to get this:

```
tasks:
    - debug: msg="{{ item.0 }} and {{ item.1 }}"
      with_together:
        - "{{ alpha }}"
        - "{{ numbers }}"
```

Looping over Subelements

Suppose you want to do something like loop over a list of users, creating them, and allowing them to login by a certain set of SSH keys.

How might that be accomplished? Let's assume you had the following defined and loaded in via "vars_files" or maybe a "group_vars/all" file:

```
---
users:
  - name: alice
    authorized:
      - /tmp/alice/onekey.pub
      - /tmp/alice/twokey.pub
    mysql:
        password: mysql-password
        hosts:
          - "%"
          - "127.0.0.1"
          - "::1"
          - "localhost"
        privs:
          - "*.*:SELECT"
          - "DB1.*:ALL"
  - name: bob
    authorized:
      - /tmp/bob/id_rsa.pub
    mysql:
        password: other-mysql-password
```

```
        hosts:
          - "db1"
        privs:
          - "*.*:SELECT"
          - "DB2.*:ALL"
```

It might happen like so:

```
- user: name={{ item.name }} state=present generate_ssh_key=yes
  with_items: "{{ users }}"

- authorized_key: "user={{ item.0.name }} key='{{ lookup('file', item.1) }}'"
  with_subelements:
     - "{{ users }}"
     - authorized
```

Given the mysql hosts and privs subkey lists, you can also iterate over a list in a nested subkey:

```
- name: Setup MySQL users
  mysql_user: name={{ item.0.name }} password={{ item.0.mysql.password }} host={{␣
↪item.1 }} priv={{ item.0.mysql.privs | join('/') }}
  with_subelements:
    - "{{ users }}"
    - mysql.hosts
```

Subelements walks a list of hashes (aka dictionaries) and then traverses a list with a given (nested sub-)key inside of those records.

Optionally, you can add a third element to the subelements list, that holds a dictionary of flags. Currently you can add the 'skip_missing' flag. If set to True, the lookup plugin will skip the lists items that do not contain the given subkey. Without this flag, or if that flag is set to False, the plugin will yield an error and complain about the missing subkey.

The authorized_key pattern is exactly where it comes up most.

Looping over Integer Sequences

`with_sequence` generates a sequence of items in ascending numerical order. You can specify a start, end, and an optional step value.

Arguments should be specified in key=value pairs. If supplied, the 'format' is a printf style string.

Numerical values can be specified in decimal, hexadecimal (0x3f8) or octal (0600). Negative numbers are not supported. This works as follows:

```
---
- hosts: all

  tasks:

    # create groups
    - group: name=evens state=present
    - group: name=odds state=present

    # create some test users
    - user: name={{ item }} state=present groups=evens
      with_sequence: start=0 end=32 format=testuser%02x

    # create a series of directories with even numbers for some reason
    - file: dest=/var/stuff/{{ item }} state=directory
      with_sequence: start=4 end=16 stride=2
```

```
    # a simpler way to use the sequence plugin
    # create 4 groups
    - group: name=group{{ item }} state=present
      with_sequence: count=4
```

Random Choices

The 'random_choice' feature can be used to pick something at random. While it's not a load balancer (there are modules for those), it can somewhat be used as a poor man's loadbalancer in a MacGyver like situation:

```
- debug: msg={{ item }}
  with_random_choice:
     - "go through the door"
     - "drink from the goblet"
     - "press the red button"
     - "do nothing"
```

One of the provided strings will be selected at random.

At a more basic level, they can be used to add chaos and excitement to otherwise predictable automation environments.

Do-Until Loops

New in version 1.4.

Sometimes you would want to retry a task until a certain condition is met. Here's an example:

```
- action: 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.

Finding First Matched Files

Note: This is an uncommon thing to want to do, but we're documenting it for completeness. You probably won't be reaching for this one often.

This isn't exactly a loop, but it's close. What if you want to use a reference to a file based on the first file found that matches a given criteria, and some of the filenames are determined by variable names? Yes, you can do that as follows:

```
- name: INTERFACES | Create Ansible header for /etc/network/interfaces
  template: src={{ item }} dest=/etc/foo.conf
  with_first_found:
    - "{{ ansible_virtualization_type }}_foo.conf"
    - "default_foo.conf"
```

This tool also has a long form version that allows for configurable search paths. Here's an example:

```
- name: some configuration template
  template: src={{ item }} dest=/etc/file.cfg mode=0444 owner=root group=root
  with_first_found:
    - files:
       - "{{ inventory_hostname }}/etc/file.cfg"
      paths:
       - ../../../templates.overwrites
       - ../../../templates
    - files:
        - etc/file.cfg
      paths:
        - templates
```

Iterating Over The Results of a Program Execution

Note: This is an uncommon thing to want to do, but we're documenting it for completeness. You probably won't be reaching for this one often.

Sometimes you might want to execute a program, and based on the output of that program, loop over the results of that line by line. Ansible provides a neat way to do that, though you should remember, this is always executed on the control machine, not the remote machine:

```
- name: Example of looping over a command result
  shell: /usr/bin/frobnicate {{ item }}
  with_lines: /usr/bin/frobnications_per_host --param {{ inventory_hostname }}
```

Ok, that was a bit arbitrary. In fact, if you're doing something that is inventory related you might just want to write a dynamic inventory source instead (see *Dynamic Inventory*), but this can be occasionally useful in quick-and-dirty implementations.

Should you ever need to execute a command remotely, you would not use the above method. Instead do this:

```
- name: Example of looping over a REMOTE command result
  shell: /usr/bin/something
  register: command_result

- name: Do something with each result
  shell: /usr/bin/something_else --param {{ item }}
  with_items: "{{ command_result.stdout_lines }}"
```

Looping Over A List With An Index

Note: This is an uncommon thing to want to do, but we're documenting it for completeness. You probably won't be reaching for this one often.

New in version 1.3.

If you want to loop over an array and also get the numeric index of where you are in the array as you go, you can also do that. It's uncommonly used:

```
- name: indexed loop demo
  debug: msg="at array position {{ item.0 }} there is a value {{ item.1 }}"
  with_indexed_items: "{{ some_list }}"
```

Using ini file with a loop

New in version 2.0.

The ini plugin can use regexp to retrieve a set of keys. As a consequence, we can loop over this set. Here is the ini file we'll use:

```
[section1]
value1=section1/value1
value2=section1/value2

[section2]
value1=section2/value1
value2=section2/value2
```

Here is an example of using `with_ini`:

```
- debug: msg="{{ item }}"
  with_ini: value[1-2] section=section1 file=lookup.ini re=true
```

And here is the returned value:

```
{
      "changed": false,
      "msg": "All items completed",
      "results": [
          {
              "invocation": {
                  "module_args": "msg=\"section1/value1\"",
                  "module_name": "debug"
              },
              "item": "section1/value1",
              "msg": "section1/value1",
              "verbose_always": true
          },
          {
              "invocation": {
                  "module_args": "msg=\"section1/value2\"",
                  "module_name": "debug"
              },
              "item": "section1/value2",
              "msg": "section1/value2",
              "verbose_always": true
          }
      ]
  }
```

Flattening A List

Note: This is an uncommon thing to want to do, but we're documenting it for completeness. You probably won't be reaching for this one often.

In rare instances you might have several lists of lists, and you just want to iterate over every item in all of those lists. Assume a really crazy hypothetical datastructure:

```
---
# file: roles/foo/vars/main.yml
packages_base:
  - [ 'foo-package', 'bar-package' ]
```

```
packages_apps:
  - [ ['one-package', 'two-package' ]]
  - [ ['red-package'], ['blue-package']]
```

As you can see the formatting of packages in these lists is all over the place. How can we install all of the packages in both lists?:

```
- name: flattened loop demo
  yum: name={{ item }} state=installed
  with_flattened:
     - "{{ packages_base }}"
     - "{{ packages_apps }}"
```

That's how!

Using register with a loop

When using `register` with a loop the data structure placed in the variable during a loop, will contain a `results` attribute, that is a list of all responses from the module.

Here is an example of using `register` with `with_items`:

```
- shell: echo "{{ item }}"
  with_items:
    - one
    - two
  register: echo
```

This differs from the data structure returned when using `register` without a loop:

```
{
    "changed": true,
    "msg": "All items completed",
    "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
  with_items: "{{ echo.results }}"
```

Looping over the inventory

If you wish to loop over the inventory, or just a subset of it, there is multiple ways. One can use a regular `with_items` with the `play_hosts` or `groups` variables, like this:

```
# show all the hosts in the inventory
- debug: msg={{ item }}
  with_items: "{{ groups['all'] }}"

# show all the hosts in the current play
- debug: msg={{ item }}
  with_items: play_hosts
```

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 }}
  with_inventory_hostnames: all

# show all the hosts matching the pattern, ie all but the group www
- debug: msg={{ item }}
  with_inventory_hostnames: all:!www
```

More information on the patterns can be found on *Patterns*

Loop Control

New in version 2.1.

In 2.0 you are again able to use *with_* 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: inner.yml
  with_items:
    - 1
    - 2
    - 3
  loop_control:
    loop_var: outer_item

# inner.yml
- debug: msg="outer item={{ outer_item }} inner item={{ item }}"
```

```
  with_items:
    - a
    - b
    - c
```

Note: 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.

New in version 2.2.

When using complex data structures for looping the display might get a bit too "busy", this is where the C(label) directive comes to help:

```
- name: create servers
  digital_ocean: name={{item.name}} state=present ....
  with_items:
    - name: server1
      disks: 3gb
      ram: 15Gb
      netowrk:
        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.

New in version 2.2.

Another option to loop control is C(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 ....
  with_items:
    - server1
    - server2
  loop_control:
    pause: 3
```

Loops and Includes in 2.0

Because *loop_control* is not available in Ansible 2.0, when using an include with a loop you should use *set_fact* to save the "outer" loops value for *item*:

```
# main.yml
- include: inner.yml
  with_items:
    - 1
    - 2
    - 3

# inner.yml
- set_fact:
    outer_item: "{{ item }}"

- debug:
```

```
    msg: "outer item={{ outer_item }} inner item={{ item }}"
  with_items:
    - a
    - b
    - c
```

Writing Your Own Iterators

While you ordinarily shouldn't have to, should you wish to write your own ways to loop over arbitrary datastructures, you can read developing_plugins for some starter information. Each of the above features are implemented as plugins in ansible, so there are many implementations to reference.

See also:

Playbooks An introduction to playbooks

Playbook Roles and Include Statements 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

In 2.0 we added a block feature to allow for logical grouping of tasks and even 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.

Listing 3.1: Block example

```
  tasks:
    - 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 the 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.

Error Handling

Blocks also introduce the ability to handle errors in a way similar to exceptions in most programming languages.

Listing 3.2: Block error handling example

```
tasks:
 - block:
     - debug: msg='i execute normally'
     - 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"
```

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.

Another example is how to run handlers after an error occurred :

Listing 3.3: Block run handlers in error handling

```
tasks:
 - 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'
```

See also:

Playbooks An introduction to playbooks

Playbook Roles and Include Statements Playbook organization by roles

User Mailing List Have a question? Stop by the google group!

irc.freenode.net #ansible IRC chat channel

Strategies

In 2.0 we added a new way to control play execution, `strategy`, by default plays will still run as they used to, with what we call the `linear` strategy. 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 via a new type of plugin, this means that in the future new execution types 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.

See also:

Playbooks An introduction to playbooks

Playbook Roles and Include Statements 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

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 thing you will definitely want to do though, is use the "roles" organization feature, which is documented as part of the main playbooks page. See *Playbook Roles and Include Statements*. You absolutely should be using roles. Roles are great. Use roles. Roles! Did we say that enough? Roles are great.

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                 # here we assign variables to particular groups
   group2                 # ""
host_vars/
   hostname1              # if systems need specific variables, put them here
   hostname2              # ""

library/                  # if any custom 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
        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

    webtier/              # same kind of structure as "common" was above, done for␣
↪the webtier role
    monitoring/           # ""
    fooapp/               # ""
```

Use Dynamic Inventory With Clouds

If you are using a cloud provider, you should not be managing your inventory in a static file. See *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-dbservers]
db-atl-1.example.com
db-atl-2.example.com

[boston-dbservers]
db-bos-1.example.com

# webservers in all geos
[webservers:children]
atlanta-webservers
boston-webservers

# dbservers in all geos
[dbservers:children]
atlanta-dbservers
boston-dbservers

# everything in the atlanta geo
[atlanta:children]
atlanta-webservers
atlanta-dbservers

# everything in the boston geo
[boston:children]
boston-webservers
boston-dbservers
```

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 webservers have some configuration that doesn't make sense for the database servers:

```
---
# file: group_vars/webservers
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 include a playbook that defines our entire infrastructure. Note this is SUPER short, because it's just including some other playbooks. Remember, playbooks are nothing more than lists of plays:

```
---
# file: site.yml
- include: webservers.yml
- include: dbservers.yml
```

In a file like webservers.yml (also at the top level), we simply map the configuration of the webservers group to the roles performed by the webservers group. Also notice this is incredibly short. For example:

```
---
# 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 *Playbook Roles and Include Statements* 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
```

What about just reconfiguring NTP on everything? Easy.:

```
ansible-playbook -i production site.yml --tags ntp
```

What about just reconfiguring my webservers?:

```
ansible-playbook -i production webservers.yml
```

What about just my webservers in Boston?:

```
ansible-playbook -i production webservers.yml --limit boston
```

What about just the first 10, and then the next 10?:

```
ansible-playbook -i production webservers.yml --limit boston[1-10]
ansible-playbook -i production webservers.yml --limit boston[11-20]
```

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 (at least in 1.1 and higher):

```
# 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_exampledotcom.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 *Inventory* and *Patterns*. Having groups named after things like *webservers* and *dbservers* 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 *Playbook Roles and Include Statements*.

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...

- 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 }}.yml"
    - 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 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.

See also:

YAML Syntax Learn about YAML syntax

Playbooks Review the basic playbook features

About Modules Learn about available modules

developing_modules Learn how to extend Ansible by writing your own modules

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

CHAPTER

FOUR

PLAYBOOKS: SPECIAL TOPICS

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.

Become (Privilege Escalation)

Ansible can use existing privilege escalation systems to allow a user to execute tasks as another.

Topics

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, which you probably already use or have configured, like *sudo*, *su*, *pfexec*, *doas*, *pbrun*, *dzdo*, *ksu* and others.

Note: Before 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, create resources with the 2nd user's permissions. As of 1.9 *become* supersedes the old sudo/su, while still being backwards compatible. This new system also makes it easier to add other privilege escalation tools like *pbrun* (Powerbroker), *pfexec*, *dzdo* (Centrify), and others.

Note: Become vars & directives are independent, i.e. setting *become_user* does not set *become*.

Directives

These can be set from play to task level, but are overriden by connection variables as they can be host specific.

become set to 'true'/'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*

become_flags (at play or task level) permit to use specific flags for the tasks or role. One common use is to change 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: true
```

To run a command as the `apache` user:

```
- name: Run a command as the apache user
  command: somecommand
  become: true
  become_user: apache
```

To do something as the `nobody` user when the shell is nologin:

```
- name: Run a command as nobody
  command: somecommand
  become: true
  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 allows to set privilege escalation method

ansible_become_user allows to set the user you become through privilege escalation, does not imply *ansible_become: True*

ansible_become_pass allows you to set the privilege escalation password

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=true
```

New command line options

--ask-become-pass, -K ask for privilege escalation password, does not imply become will be used

--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]

--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`.

Note: 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 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.

Changed in version 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.

See also:

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

irc.freenode.net #ansible IRC chat channel

Accelerated Mode

New in version 1.3.

Note: Accelerated mode is deprecated. Consider using SSH with ControlPersist and pipelining enabled instead. This feature will be removed in a future release. Deprecation warnings can be disabled by setting `deprecation_warnings=False` in `ansible.cfg`.

You Might Not Need This!

Are you running Ansible 1.5 or later? If so, you may not need accelerated mode due to a new feature called "SSH pipelining" and should read the *pipelining* section of the documentation.

For users on 1.5 and later, accelerated mode only makes sense if you (A) are managing from an Enterprise Linux 6 or earlier host and still are on paramiko, or (B) can't enable TTYs with sudo as described in the pipelining docs.

If you can use pipelining, Ansible will reduce the amount of files transferred over the wire, making everything much more efficient, and performance will be on par with accelerated mode in nearly all cases, possibly excluding very large file transfer. Because less moving parts are involved, pipelining is better than accelerated mode for nearly all use cases.

Accelerated moded remains around in support of EL6 control machines and other constrained environments.

Accelerated Mode Details

While OpenSSH using the ControlPersist feature is quite fast and scalable, there is a certain small amount of overhead involved in using SSH connections. While many people will not encounter a need, if you are running on a platform that doesn't have ControlPersist support (such as an EL6 control machine), you'll probably be even more interested in tuning options.

Accelerated mode is there to help connections work faster, but still uses SSH for initial secure key exchange. There is no additional public key infrastructure to manage, and this does not require things like NTP or even DNS.

Accelerated mode can be anywhere from 2-6x faster than SSH with ControlPersist enabled, and 10x faster than paramiko.

Accelerated mode works by launching a temporary daemon over SSH. Once the daemon is running, Ansible will connect directly to it via a socket connection. Ansible secures this communication by using a temporary AES key that is exchanged during the SSH connection (this key is different for every host, and is also regenerated periodically).

By default, Ansible will use port 5099 for the accelerated connection, though this is configurable. Once running, the daemon will accept connections for 30 minutes, after which time it will terminate itself and need to be restarted over SSH.

Accelerated mode offers several improvements over the (deprecated) original fireball mode from which it was based:

- No bootstrapping is required, only a single line needs to be added to each play you wish to run in accelerated mode.
- Support for sudo commands (see below for more details and caveats) is available.
- There are fewer requirements. ZeroMQ is no longer required, nor are there any special packages beyond python-keyczar
- python 2.5 or higher is required.

In order to use accelerated mode, simply add *accelerate: true* to your play:

```
---

- hosts: all
  accelerate: true

  tasks:

  - name: some task
    command: echo {{ item }}
    with_items:
    - foo
    - bar
    - baz
```

If you wish to change the port Ansible will use for the accelerated connection, just add the *accelerate_port* option:

```
---

- hosts: all
  accelerate: true
  # default port is 5099
  accelerate_port: 10000
```

The *accelerate_port* option can also be specified in the environment variable ACCELERATE_PORT, or in your *ansible.cfg* configuration:

```
[accelerate]
accelerate_port = 5099
```

As noted above, accelerated mode also supports running tasks via sudo, however there are two important caveats:

- You must remove requiretty from your sudoers options.
- Prompting for the sudo password is not yet supported, so the NOPASSWD option is required for sudo'ed commands.

As of Ansible version *1.6*, you can also allow the use of multiple keys for connections from multiple Ansible management nodes. To do so, add the following option to your *ansible.cfg* configuration:

```
accelerate_multi_key = yes
```

When enabled, the daemon will open a UNIX socket file (by default *$ANSIBLE_REMOTE_TEMP/.ansible-accelerate/.local.socket*). New connections over SSH can use this socket file to upload new keys to the daemon.

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.

The easiest way to do this is to kick them off all at once and then poll until they are done.

You will also want to use asynchronous mode on very long running operations that might be subject to timeout.

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
```

Note: 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 "fire and forget" 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
```

Note: You shouldn't "fire and forget" 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.

Note: 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 variation of the "fire and forget" where you "fire and forget, check on it later" you can perform a task similar to the following:

```
---
# Requires ansible 1.8+
- name: 'YUM - fire and forget task'
  yum: name=docker-io state=installed
  async: 1000
  poll: 0
  register: yum_sleeper

- name: 'YUM - check on fire and forget task'
  async_status: jid={{ yum_sleeper.ansible_job_id }}
  register: job_result
  until: job_result.finished
  retries: 30
```

Note: 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

See also:

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")

New in version 1.1.

Topics

- *Check Mode ("Dry Run")*

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

New in version 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`.

Note: Prior to version 2.2 only the 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 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

New in version 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`

New in version 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, if any templated files on the remote system are changed, and the ansible-playbook CLI will report back the textual changes made to the file (or, if used with `--check`, the changes that would have been made). Since the diff feature produces a large amount of output, it is best used when checking a single host at a time, like so:

```
ansible-playbook foo.yml --check --diff --limit foo.example.com
```

Playbook Debugger

Topics

In 2.1 we added a `debug` strategy. This strategy enables you to invoke a debugger when a task is failed, and check several info, such as the value of a variable. Also, it is possible to update module arguments in the debugger, and run the failed task again with new arguments to consider how you can fix an issue.

To use `debug` strategy, change `strategy` attribute like this:

```
- hosts: test
  strategy: debug
  tasks:
  ...
```

For example, run the playbook below:

```
- hosts: test
  strategy: debug
  gather_facts: no
  vars:
    var1: value1
  tasks:
    - name: wrong variable
      ping: data={{ wrong_var }}
```

The debugger is invoked since *wrong_var* variable is undefined. Let's change the module's args, 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
(debug) p result
{'msg': u"ERROR! 'wrong_var' is undefined", 'failed': True}
(debug) p task.args
{u'data': u'{{ wrong_var }}'}
(debug) task.args['data'] = '{{ var1 }}'
(debug) p task.args
{u'data': '{{ var1 }}'}
(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 *task/vars/host/result*

Print values used to execute a module:

```
(debug) p task
TASK: install package
(debug) p task.args
{u'name': u'{{ pkg_name }}'}
(debug) p vars
{u'ansible_all_ipv4_addresses': [u'192.0.2.10'],
 u'ansible_architecture': u'x86_64',
 ...
}
(debug) p vars['pkg_name']
u'bash'
(debug) p host
192.0.2.10
(debug) p 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:
    - name: install package
      apt: name={{ pkg_name }}
```

Debugger is invoked due to wrong package name, so let's fix the module's args:

```
(debug) p task.args
{u'name': u'{{ pkg_name }}'}
(debug) task.args['name'] = 'bash'
(debug) p task.args
{u'name': 'bash'}
(debug) redo
```

Then the task runs again with new args.

vars[*key*] = *value*

Update vars.

Let's use the same playbook above, but fix vars instead of args:

```
(debug) p vars['pkg_name']
u'not_exist'
(debug) vars['pkg_name'] = 'bash'
(debug) p vars['pkg_name']
'bash'
(debug) redo
```

Then the task runs again with new vars.

r(edo)

Run the task again.

c(ontinue)

Just continue.

q(uit)

Quit from the debugger. The playbook execution is aborted.

See also:

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

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 *About 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 *Playbook Roles and Include Statements*, as the 'pre_task' and 'post_task' concepts are the places where you would typically call these modules.

Rolling Update Batch Size

New in version 0.7.

By default, Ansible will try to manage all of the machines referenced in a play in parallel. For a rolling updates 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 in Ansible 1.8 and later, 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 batche 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%"
```

Note: No matter how small the percentage, the number of hosts per pass will always be 1 or greater.

Maximum Failure Percentage

New in version 1.3.

By default, Ansible will continue executing actions as long as there are hosts in the group that have not yet failed. 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, as of version 1.3 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.

Note: 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

New in version 0.7.

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. Using this with the 'serial' keyword to control the number of hosts executing at one time is also a good idea:

```
---

- hosts: webservers
  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.

The *ansible_host* variable (*ansible_ssh_host* in 1.x) reflects the host a task is delegated to.

Delegated facts

New in version 2.0.

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). In 2.0, 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
      with_items: "{{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_addresses'][0]* even though dbservers were not part of the play, or left out by using *–limit*.

Run Once

New in version 1.7.

In some cases there may be a need to only run a task one time and only on one host. This can be achieved by configuring "run_once" on a task:

```
---
# ...

  tasks:

    # ...

    - command: /opt/application/upgrade_db.py
      run_once: true

    # ...
```

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
```

When "run_once" is not used with "delegate_to" it will execute on the first host, as defined by inventory, in the group(s) of hosts targeted by the play - e.g. webservers[0] if the play targeted "hosts: webservers".

This approach is similar to applying a conditional to a task such as:

```
- command: /opt/application/upgrade_db.py
  when: inventory_hostname == webservers[0]
```

Note: 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.

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
```

Interrupt execution on any error

With option ''any_errors_fatal" 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 - number of hosts is unpredictable (because of dynamic inventory), speed is crucial (simultaneous execution is required). But all tasks must be 100% successful to continue playbook execution.

For example there is a service located in many datacenters, there a some load balancers to pass traffic from users to service. There is a deploy playbook to upgrade service deb-packages. Playbook stages:

- disable traffic on load balancers (must be turned off simultaneously)
- gracefully stop service
- upgrade software (this step includes tests and starting service)
- enable traffic on load balancers (should be turned off simultaneously)

Service can't be stopped with "alive" load balancers, they must be disabled, all of them. So second stage can't be played if any server failed on "stage 1".

For datacenter "A" 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 software upgrade on frontends only if all load balancers are successfully disabled.

See also:

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)

New in version 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:
  http_proxy: http://proxy.bos.example.com:8080
  https_proxy: http://proxy.bos.example.com:8080
```

Note: `environment:` is not currently supported for Windows targets

See also:

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

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 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

New in version 0.6.

Generally playbooks will stop executing any more steps on a host that has a failure. 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, it will still raise an error that users will need to address. Neither will this prevent failures on connection nor execution issues, the task must be able to run and return a value of 'failed'.

Handlers and Failure

New in version 1.9.1.

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

New in version 1.4.

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 in 1.4 and later provides a way to specify this behavior as follows:

```
- name: this command prints FAILED when it fails
  command: /usr/bin/example-command -x -y -z
  register: command_result
  failed_when: "'FAILED' in command_result.stderr"
```

In previous version of Ansible, this can be 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

New in version 1.3.

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.

See also:

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

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

As of Ansible 2.0, there is 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 subsitition 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'
```

See also:

Variables All about variables

User Mailing List Have a question? Stop by the google group!

irc.freenode.net #ansible IRC chat channel

Using Lookups

Lookup plugins allow access of data in Ansible from outside sources. These plugins are evaluated on the Ansible control machine, and can include reading the filesystem but also contacting external datastores and services. These values are then made available using the standard templating system in Ansible, and are typically used to load variables or templates with information from those systems.

Note: This is considered an advanced feature, and many users will probably not rely on these features.

Note: Lookups occur on the local computer, not on the remote computer.

Note: Lookups are executed with a cwd relative to the role or play, as opposed to local tasks which are executed with the cwd of the executed script.

Note: Since 1.9 you can pass wantlist=True to lookups to use in jinja2 template "for" loops.

Topics

Intro to Lookups: Getting File Contents

The file lookup is the most basic lookup type.

Contents can be read off the filesystem as follows:

```
---
- hosts: all
  vars:
     contents: "{{ lookup('file', '/etc/foo.txt') }}"

  tasks:

     - debug: msg="the value of foo.txt is {{ contents }}"
```

The Password Lookup

Note: A great alternative to the password lookup plugin, if you don't need to generate random passwords on a per-host basis, would be to use *Vault*. Read the documentation there and consider using it first, it will be more desirable for most applications.

`password` generates a random plaintext password and stores it in a file at a given filepath.

(Docs about crypted save modes are pending)

If the file exists previously, it will retrieve its contents, behaving just like with_file. Usage of variables like "{{ inventory_hostname }}" in the filepath can be used to set up random passwords per host (which simplifies password management in 'host_vars' variables).

Generated passwords contain a random mix of upper and lowercase ASCII letters, the numbers 0-9 and punctuation (". , : - _"). The default length of a generated password is 20 characters. This length can be changed by passing an extra parameter:

```
---
- hosts: all

  tasks:

    # create a mysql user with a random password:
    - mysql_user: name={{ client }}
                  password="{{ lookup('password', 'credentials/' + client + '/' +␣
↪tier + '/' + role + '/mysqlpassword length=15') }}"
                  priv={{ client }}_{{ tier }}_{{ role }}.*:ALL

    (...)
```

Note: If the file already exists, no data will be written to it. If the file has contents, those contents will be read in as the password. Empty files cause the password to return as an empty string.

Caution: Since this runs on the ansible host as the user running the playbook, and "become" does not apply, the target file must be readable by the playbook user, or, if it does not exist, the playbook user must have sufficient privileges to create it. (So, for example, attempts to write into areas such as /etc will fail unless the entire playbook is being run as root).

Starting in version 1.4, password accepts a "chars" parameter to allow defining a custom character set in the generated passwords. It accepts comma separated list of names that are either string module attributes (ascii_letters,digits, etc) or are used literally:

```
---
- hosts: all

  tasks:
```

```
    # create a mysql user with a random password using only ascii letters:
    - mysql_user: name={{ client }}
                  password="{{ lookup('password', '/tmp/passwordfile chars=ascii_
↪letters') }}"
                  priv={{ client }}_{{ tier }}_{{ role }}.*:ALL

    # create a mysql user with a random password using only digits:
    - mysql_user: name={{ client }}
                  password="{{ lookup('password', '/tmp/passwordfile chars=digits
↪') }}"
                  priv={{ client }}_{{ tier }}_{{ role }}.*:ALL

    # create a mysql user with a random password using many different char sets:
    - mysql_user: name={{ client }}
                  password="{{ lookup('password', '/tmp/passwordfile chars=ascii_
↪letters,digits,hexdigits,punctuation') }}"
                  priv={{ client }}_{{ tier }}_{{ role }}.*:ALL

    (...)
```

To enter comma use two commas ‘,,’ somewhere - preferably at the end. Quotes and double quotes are not supported.

The CSV File Lookup

New in version 1.5.

The `csvfile` lookup reads the contents of a file in CSV (comma-separated value) format. The lookup looks for the row where the first column matches `keyname`, and returns the value in the second column, unless a different column is specified.

The example below shows the contents of a CSV file named elements.csv with information about the periodic table of elements:

```
Symbol,Atomic Number,Atomic Mass
H,1,1.008
He,2,4.0026
Li,3,6.94
Be,4,9.012
B,5,10.81
```

We can use the `csvfile` plugin to look up the atomic number or atomic of Lithium by its symbol:

```
- debug: msg="The atomic number of Lithium is {{ lookup('csvfile', 'Li␣
↪file=elements.csv delimiter=,') }}"
- debug: msg="The atomic mass of Lithium is {{ lookup('csvfile', 'Li file=elements.
↪csv delimiter=, col=2') }}"
```

The `csvfile` lookup supports several arguments. The format for passing arguments is:

```
lookup('csvfile', 'key arg1=val1 arg2=val2 ...')
```

The first value in the argument is the `key`, which must be an entry that appears exactly once in column 0 (the first column, 0-indexed) of the table. All other arguments are optional.

Field	Default	Description
file	ansible.csv	Name of the file to load
col	1	The column to output, indexed by 0
delimiter	TAB	Delimiter used by CSV file. As a special case, tab can be specified as either TAB or t.
default	empty string	Default return value if the key is not in the csv file
encoding	utf-8	Encoding (character set) of the used CSV file (added in version 2.1)

Note: The default delimiter is TAB, *not* comma.

The INI File Lookup

New in version 2.0.

The `ini` lookup reads the contents of a file in INI format (key1=value1). This plugin retrieve the value on the right side after the equal sign ('=') of a given section ([section]). You can also read a property file which - in this case - does not contain section.

Here's a simple example of an INI file with user/password configuration:

```
[production]
# My production information
user=robert
pass=somerandompassword

[integration]
# My integration information
user=gertrude
pass=anotherpassword
```

We can use the `ini` plugin to lookup user configuration:

```
- debug: msg="User in integration is {{ lookup('ini', 'user section=integration␣
↪file=users.ini') }}"
- debug: msg="User in production  is {{ lookup('ini', 'user section=production ␣
↪file=users.ini') }}"
```

Another example for this plugin is for looking for a value on java properties. Here's a simple properties we'll take as an example:

```
user.name=robert
user.pass=somerandompassword
```

You can retrieve the `user.name` field with the following lookup:

```
- debug: msg="user.name is {{ lookup('ini', 'user.name type=properties file=user.
↪properties') }}"
```

The `ini` lookup supports several arguments like the csv plugin. The format for passing arguments is:

```
lookup('ini', 'key [type=<properties|ini>] [section=section] [file=file.ini]␣
↪[re=true] [default=<defaultvalue>]')
```

The first value in the argument is the `key`, which must be an entry that appears exactly once on keys. All other arguments are optional.

Field	Default	Description
type	ini	Type of the file. Can be ini or properties (for java properties).
file	ansible.ini	Name of the file to load
section	global	Default section where to lookup for key.
re	False	The key is a regexp.
default	empty string	return value if the key is not in the ini file

Note: In java properties files, there's no need to specify a section.

The Credstash Lookup

New in version 2.0.

Credstash is a small utility for managing secrets using AWS's KMS and DynamoDB: https://github.com/LuminalOSS/credstash

First, you need to store your secrets with credstash:

```
$ credstash put my-github-password secure123

my-github-password has been stored
```

Example usage:

```
---
- name: "Test credstash lookup plugin -- get my github password"
  debug: msg="Credstash lookup! {{ lookup('credstash', 'my-github-password') }}"
```

You can specify regions or tables to fetch secrets from:

```
---
- name: "Test credstash lookup plugin -- get my other password from us-west-1"
  debug: msg="Credstash lookup! {{ lookup('credstash', 'my-other-password', region=
↪'us-west-1') }}"


- name: "Test credstash lookup plugin -- get the company's github password"
  debug: msg="Credstash lookup! {{ lookup('credstash', 'company-github-password',␣
↪table='company-passwords') }}"
```

If you use the context feature when putting your secret, you can get it by passing a dictionary to the context option like this:

```
---
- name: test
  hosts: localhost
  vars:
    context:
      app: my_app
      environment: production
  tasks:

  - name: "Test credstash lookup plugin -- get the password with a context passed␣
↪as a variable"
    debug: msg="{{ lookup('credstash', 'some-password', context=context) }}"

  - name: "Test credstash lookup plugin -- get the password with a context defined␣
↪here"
    debug: msg="{{ lookup('credstash', 'some-password', context=dict(app='my_app',␣
↪environment='production')) }}"
```

If you're not using 2.0 yet, you can do something similar with the credstash tool and the pipe lookup (see below):

```
debug: msg="Poor man's credstash lookup! {{ lookup('pipe', 'credstash -r us-west-1␣
↪get my-other-password') }}"
```

The DNS Lookup (dig)

New in version 1.9.0.

Warning: This lookup depends on the dnspython library.

The `dig` lookup runs queries against DNS servers to retrieve DNS records for a specific name (*FQDN* - fully qualified domain name). It is possible to lookup any DNS record in this manner.

There is a couple of different syntaxes that can be used to specify what record should be retrieved, and for which name. It is also possible to explicitly specify the DNS server(s) to use for lookups.

In its simplest form, the `dig` lookup plugin can be used to retrieve an IPv4 address (DNS `A` record) associated with *FQDN*:

Note: If you need to obtain the `AAAA` record (IPv6 address), you must specify the record type explicitly. Syntax for specifying the record type is described below.

Note: The trailing dot in most of the examples listed is purely optional, but is specified for completeness/correctness sake.

```
- debug: msg="The IPv4 address for example.com. is {{ lookup('dig', 'example.com.
↪')}}"
```

In addition to (default) `A` record, it is also possible to specify a different record type that should be queried. This can be done by either passing-in additional parameter of format `qtype=TYPE` to the `dig` lookup, or by appending `/TYPE` to the *FQDN* being queried. For example:

```
- debug: msg="The TXT record for example.org. is {{ lookup('dig', 'example.org.',
↪'qtype=TXT') }}"
- debug: msg="The TXT record for example.org. is {{ lookup('dig', 'example.org./TXT
↪') }}"
```

If multiple values are associated with the requested record, the results will be returned as a comma-separated list. In such cases you may want to pass option `wantlist=True` to the plugin, which will result in the record values being returned as a list over which you can iterate later on:

```
- debug: msg="One of the MX records for gmail.com. is {{ item }}"
  with_items: "{{ lookup('dig', 'gmail.com./MX', wantlist=True) }}"
```

In case of reverse DNS lookups (`PTR` records), you can also use a convenience syntax of format `IP_ADDRESS/PTR`. The following three lines would produce the same output:

```
- debug: msg="Reverse DNS for 192.0.2.5 is {{ lookup('dig', '192.0.2.5/PTR') }}"
- debug: msg="Reverse DNS for 192.0.2.5 is {{ lookup('dig', '5.2.0.192.in-addr.
↪arpa./PTR') }}"
- debug: msg="Reverse DNS for 192.0.2.5 is {{ lookup('dig', '5.2.0.192.in-addr.
↪arpa.', 'qtype=PTR') }}"
```

By default, the lookup will rely on system-wide configured DNS servers for performing the query. It is also possible to explicitly specify DNS servers to query using the

`@DNS_SERVER_1,DNS_SERVER_2,...,DNS_SERVER_N` notation. This needs to be passed-in as an additional parameter to the lookup. For example:

```
- debug: msg="Querying 198.51.100.23 for IPv4 address for example.com. produces {{␣
↪lookup('dig', 'example.com', '@198.51.100.23') }}"
```

In some cases the DNS records may hold a more complex data structure, or it may be useful to obtain the results in a form of a dictionary for future processing. The `dig` lookup supports parsing of a number of such records, with the result being returned as a dictionary. This way it is possible to easily access such nested data. This return format can be requested by passing-in the `flat=0` option to the lookup. For example:

```
- debug: msg="XMPP service for gmail.com. is available at {{ item.target }} on␣
↪port {{ item.port }}"
  with_items: "{{ lookup('dig', '_xmpp-server._tcp.gmail.com./SRV', 'flat=0',␣
↪wantlist=True) }}"
```

Take note that due to the way Ansible lookups work, you must pass the `wantlist=True` argument to the lookup, otherwise Ansible will report errors.

Currently the dictionary results are supported for the following records:

Note: *ALL* is not a record per-se, merely the listed fields are available for any record results you retrieve in the form of a dictionary.

Record	Fields
ALL	owner, ttl, type
A	address
AAAA	address
CNAME	target
DNAME	target
DLV	algorithm, digest_type, key_tag, digest
DNSKEY	flags, algorithm, protocol, key
DS	algorithm, digest_type, key_tag, digest
HINFO	cpu, os
LOC	latitude, longitude, altitude, size, horizontal_precision, vertical_precision
MX	preference, exchange
NAPTR	order, preference, flags, service, regexp, replacement
NS	target
NSEC3PARAM	algorithm, flags, iterations, salt
PTR	target
RP	mbox, txt
SOA	mname, rname, serial, refresh, retry, expire, minimum
SPF	strings
SRV	priority, weight, port, target
SSHFP	algorithm, fp_type, fingerprint
TLSA	usage, selector, mtype, cert
TXT	strings

More Lookups

Various *lookup plugins* allow additional ways to iterate over data. In *Loops* you will learn how to use them to walk over collections of numerous types. However, they can also be used to pull in data from remote sources, such as shell commands or even key value stores. This section will cover lookup plugins in this capacity.

Here are some examples:

```
---
- hosts: all
```

```
  tasks:

    - debug: msg="{{ lookup('env','HOME') }} is an environment variable"

    - debug: msg="{{ item }} is a line from the result of this command"
      with_lines:
        - cat /etc/motd

    - debug: msg="{{ lookup('pipe','date') }} is the raw result of running this␣
↪command"

    # redis_kv lookup requires the Python redis package
    - debug: msg="{{ lookup('redis_kv', 'redis://localhost:6379,somekey') }} is␣
↪value in Redis for somekey"

    # dnstxt lookup requires the Python dnspython package
    - debug: msg="{{ lookup('dnstxt', 'example.com') }} is a DNS TXT record for␣
↪example.com"

    - debug: msg="{{ lookup('template', './some_template.j2') }} is a value from␣
↪evaluation of this template"

    # loading a json file from a template as a string
    - debug: msg="{{ lookup('template', './some_json.json.j2', convert_
↪data=False) }} is a value from evaluation of this template"


    - debug: msg="{{ lookup('etcd', 'foo') }} is a value from a locally running␣
↪etcd"

    # shelvefile lookup retrieves a string value corresponding to a key inside a␣
↪Python shelve file
    - debug: msg="{{ lookup('shelvefile', 'file=path_to_some_shelve_file.db␣
↪key=key_to_retrieve') }}

    # The following lookups were added in 1.9
    - debug: msg="{{item}}"
      with_url:
           - 'https://github.com/gremlin.keys'

    # outputs the cartesian product of the supplied lists
    - debug: msg="{{item}}"
      with_cartesian:
           - list1
           - list2
           - list3
```

As an alternative you can also assign lookup plugins to variables or use them elsewhere. This 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 }}"
```

See also:

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

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?"
```

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 crypt 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.

See also:

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.

Example:

```
tasks:

    - yum: name={{ item }} state=installed
      with_items:
         - httpd
         - 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 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 roles:

```
roles:
  - { role: webserver, port: 5000, tags: [ 'web', 'foo' ] }
```

And include statements:

```
- include: foo.yml
  tags: [web,foo]
```

All of these apply the specified tags to EACH task inside the play, included file, or role, so that these tasks can be selectively run when the playbook is invoked with the corresponding tags.

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
```

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.

See also:

Playbooks An introduction to playbooks

Playbook Roles and Include Statements Playbook organization by roles

User Mailing List Have a question? Stop by the google group!

irc.freenode.net #ansible IRC chat channel

Vault

Topics

New in Ansible 1.5, "Vault" is a feature of ansible that allows keeping sensitive data such as passwords or keys in encrypted files, rather than as plaintext in your 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

The vault feature 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. However, that might be a little too much and could annoy your coworkers :)

The vault feature can also encrypt arbitrary files, even binary files. If a vault-encrypted file is given as the *src* argument to the *copy* module, the file will be placed at the destination on the target host decrypted (assuming a valid vault password is supplied when running the play).

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

Available since Ansible 1.8

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
```

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.

Note: 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).

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
```

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.

CHAPTER

FIVE

ABOUT MODULES

Introduction

Modules (also referred to as "task plugins" or "library plugins") are the ones that do the actual work in ansible, they are what gets executed in each playbook task. But you can also run a single one using the 'ansible' command.

Let's review how we execute three different modules from the command line:

```
ansible webservers -m service -a "name=httpd state=started"
ansible webservers -m ping
ansible webservers -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 yaml 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 strive to be *idempotent*, meaning they will seek to avoid changes to the system unless a change needs to be made. 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
```

A list of all installed modules is also available:

```
ansible-doc -l
```

See also:

Introduction To Ad-Hoc Commands Examples of using modules in /usr/bin/ansible

Playbooks Examples of using modules with /usr/bin/ansible-playbook

developing_modules How to write your own modules

developing_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

Core Modules

These are modules that the core ansible team maintains and will always ship with ansible itself. They will also receive slightly higher priority for all requests than those in the "extras" repos.

The source of these modules is hosted on GitHub in the ansible-modules-core repo.

If you believe you have found a bug in a core module and are already running the latest stable or development version of Ansible, first look in the issue tracker at github.com/ansible/ansible-modules-core to see if a bug has already been filed. If not, we would be grateful if you would 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. Development oriented topics should instead use the similar ansible-devel google group.

Documentation updates for these modules can also be edited directly in the module itself and by submitting a pull request to the module source code, just look for the "DOCUMENTATION" block in the source tree.

Extras Modules

These modules are currently shipped with Ansible, but might be shipped separately in the future. They are also mostly maintained by the community. Non-core modules are still fully usable, but may receive slightly lower response rates for issues and pull requests.

Popular "extras" modules may be promoted to core modules over time.

This source for these modules is hosted on GitHub in the ansible-modules-extras repo.

If you believe you have found a bug in an extras module and are already running the latest stable or development version of Ansible, first look in the issue tracker at github.com/ansible/ansible-modules-extras to see if a bug has already been filed. If not, we would be grateful if you would file one.

Should you have a question rather than a bug report, inquries are welcome on the ansible-project google group or on Ansible's "#ansible" channel, located on irc.freenode.net. Development oriented topics should instead use the similar ansible-devel google group.

Documentation updates for this module can also be edited directly in the module and by submitting a pull request to the module source code, just look for the "DOCUMENTATION" block in the source tree.

For help in developing on modules, should you be so inclined, please read *Community Information & Contributing*, developing_test_pr and developing_modules.

Common Return Values

Topics

- *Common Return Values*

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 https://docs.ansible.com).

This document covers return values common to all modules.

Note: Some of these keys might be set by Ansible itself once it processes the module's return information.

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 c(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 c(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.

See also:

About 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

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.

See also:

Introduction To Ad-Hoc Commands Examples of using modules in /usr/bin/ansible

Playbooks Examples of using modules with /usr/bin/ansible-playbook

developing_modules How to write your own modules

developing_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

CHAPTER

SIX

DETAILED GUIDES

This section is new and evolving. The idea here is to explore particular use cases in greater depth and provide a more "top down" explanation of some basic features.

Amazon Web Services Guide

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:
    - ...
```

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--"
```

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
  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
    with_items: ec2.instances
```

With the host group now created, a second play at the bottom of the 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
```

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 *Dynamic Inventory*.

This will also dynamically select nodes that were even created outside of Ansible, and allow Ansible to manage them.

See *Dynamic Inventory* for how to use this, then flip back over to this chapter.

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.

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.

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.

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.

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.

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.

See also:

About Modules All the documentation for Ansible modules

Playbooks An introduction to playbooks

Delegation, Rolling Updates, and Local Actions Delegation, useful for working with loud balancers, clouds, and locally executed steps.

User Mailing List Have a question? Stop by the google group!

irc.freenode.net #ansible IRC chat channel

Getting Started with Azure

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.

Requirements

Using the Azure Resource Manager modules requires having Azure Python SDK installed on the host running Ansible. You will need to have == v2.0.0RC5 installed. The simplest way to install the SDK is via pip:

```
$ pip install "azure==2.0.0rc5"
```

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
- AZURE_SUBSCRIPTION_ID

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=xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxx
client_id=xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxx
secret=xxxxxxxxxxxxxxx
tenant=xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxx
```

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

- subscription_id

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:
      - 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
```

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"
  },
  "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
```

Rackspace Cloud Guide

Introduction

Note: 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:
```

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

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
```

Note: 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.

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

Note: 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.

Note: Ansible 2.0 has deprecated the "ssh" from `ansible_ssh_user`, `ansible_ssh_host`, and `ansible_ssh_port` to become `ansible_user`, `ansible_host`, and `ansible_port`. If you are using a version of Ansible prior to 2.0, you should continue using the older style variables (`ansible_ssh_*`). These shorter variables are ignored, without warning, in older versions of Ansible.

```
- 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
  with_items: 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.

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.

Note: 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`.

Note: 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_accessipv4": "198.51.100.1",
                "rax_accessipv6": "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",
                            "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"
            }
        }
    }
}
```

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_accessipv4 }}"
```

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_accessipv4": "198.51.100.1",
        "rax_accessipv6": "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",
                    "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"
    },
```

```
    "changed": false
}
```

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: performance1-1
      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 }}"
      ansible_host: "{{ item.rax_accessipv4 }}"
      ansible_ssh_pass: "{{ item.rax_adminpass }}"
      ansible_user: root
      groups: web
    with_items: 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
      with_items: 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: performance1-1
        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
      with_items: 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 rackconnnect 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
      register: rax_facts
      until: rax_facts.ansible_facts['rax_metadata']['rax_service_level_automation
↪']|default('') == 'Complete'
      retries: 30
      delay: 10

- 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'
      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'
      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
```

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

Google Cloud Platform Guide

Introduction

Note: This section of the documentation is under construction. We are in the process of adding more examples about all of the GCE modules and how they work together. Upgrades via github pull requests are welcomed!

Ansible contains modules for managing Google Compute Engine resources, including creating instances, controlling network access, working with persistent disks, and managing load balancers. Additionally, there is an inventory plugin that can automatically suck down all of your GCE instances into Ansible dynamic inventory, and create groups by tag and other properties.

The GCE modules all require the apache-libcloud module which you can install from pip:

```
$ pip install apache-libcloud
```

Note: If you're using Ansible on Mac OS X, libcloud also needs to access a CA cert chain. You'll need to download one (you can get one for here.)

Credentials

To work with the GCE modules, you'll first need to get some credentials in the JSON format:

1. Create a Service Account
2. Download JSON credentials

There are three different ways to provide credentials to Ansible so that it can talk with Google Cloud for provisioning and configuration actions:

Note: If you would like to use JSON credentials you must have libcloud >= 0.17.0

- by providing to the modules directly

- by populating a `secrets.py` file
- by setting environment variables

Calling Modules By Passing Credentials

For the GCE modules you can specify the credentials as arguments:

- `service_account_email`: email associated with the project
- `credentials_file`: path to the JSON credentials file
- `project_id`: id of the project

For example, to create a new instance using the cloud module, you can use the following configuration:

```
- name: Create instance(s)
  hosts: localhost
  connection: local
  gather_facts: no

  vars:
    service_account_email: unique-id@developer.gserviceaccount.com
    credentials_file: /path/to/project.json
    project_id: project-id
    machine_type: n1-standard-1
    image: debian-7

  tasks:

   - name: Launch instances
     gce:
         instance_names: dev
         machine_type: "{{ machine_type }}"
         image: "{{ image }}"
         service_account_email: "{{ service_account_email }}"
         credentials_file: "{{ credentials_file }}"
         project_id: "{{ project_id }}"
```

When running Ansible inside a GCE VM you can use the service account credentials from the local metadata server by setting both `service_account_email` and `credentials_file` to a blank string.

Configuring Modules with secrets.py

Create a file `secrets.py` looking like following, and put it in some folder which is in your `$PYTHONPATH`:

```
GCE_PARAMS = ('i...@project.googleusercontent.com', '/path/to/project.json')
GCE_KEYWORD_PARAMS = {'project': 'project_id'}
```

Ensure to enter the email address from the created services account and not the one from your main account.

Now the modules can be used as above, but the account information can be omitted.

If you are running Ansible from inside a GCE VM with an authorized service account you can set the email address and credentials path as follows so that get automatically picked up:

```
GCE_PARAMS = ('', '')
GCE_KEYWORD_PARAMS = {'project': 'project_id'}
```

Configuring Modules with Environment Variables

Set the following environment variables before running Ansible in order to configure your credentials:

```
GCE_EMAIL
GCE_PROJECT
GCE_CREDENTIALS_FILE_PATH
```

GCE Dynamic Inventory

The best way to interact with your hosts is to use the gce inventory plugin, which dynamically queries GCE and tells Ansible what nodes can be managed.

Note that when using the inventory script `gce.py`, you also need to populate the `gce.ini` file that you can find in the contrib/inventory directory of the ansible checkout.

To use the GCE dynamic inventory script, copy `gce.py` from `contrib/inventory` into your inventory directory and make it executable. You can specify credentials for `gce.py` using the `GCE_INI_PATH` environment variable – the default is to look for gce.ini in the same directory as the inventory script.

Let's see if inventory is working:

```
$ ./gce.py --list
```

You should see output describing the hosts you have, if any, running in Google Compute Engine.

Now let's see if we can use the inventory script to talk to Google.

```
$ GCE_INI_PATH=~/.gce.ini ansible all -i gce.py -m setup
hostname | success >> {
  "ansible_facts": {
    "ansible_all_ipv4_addresses": [
      "x.x.x.x"
    ],
```

As with all dynamic inventory scripts in Ansible, you can configure the inventory path in ansible.cfg. The recommended way to use the inventory is to create an `inventory` directory, and place both the `gce.py` script and a file containing `localhost` in it. This can allow for cloud inventory to be used alongside local inventory (such as a physical datacenter) or machines running in different providers.

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 once again use our inventory script to see if it can talk to Google Cloud:

```
$ ansible all -i inventory/ -m setup
hostname | success >> {
  "ansible_facts": {
    "ansible_all_ipv4_addresses": [
        "x.x.x.x"
    ],
```

The output should be similar to the previous command. If you're wanting less output and just want to check for SSH connectivity, use "-m" ping instead.

Use Cases

For the following use case, let's use this small shell script as a wrapper.

```
#!/usr/bin/env bash
PLAYBOOK="$1"

if [[ -z $PLAYBOOK ]]; then
  echo "You need to pass a playbook as argument to this script."
  exit 1
```

```
fi

export SSL_CERT_FILE=$(pwd)/cacert.cer
export ANSIBLE_HOST_KEY_CHECKING=False

if [[ ! -f "$SSL_CERT_FILE" ]]; then
  curl -O http://curl.haxx.se/ca/cacert.pem
fi

ansible-playbook -v -i inventory/ "$PLAYBOOK"
```

Create an instance

The GCE module provides the ability to provision instances within Google Compute Engine. The provisioning task is typically performed from your Ansible control server against Google Cloud's API.

A playbook would looks like this:

```
- name: Create instance(s)
  hosts: localhost
  gather_facts: no
  connection: local

  vars:
    machine_type: n1-standard-1 # default
    image: debian-7
    service_account_email: unique-id@developer.gserviceaccount.com
    credentials_file: /path/to/project.json
    project_id: project-id

  tasks:
    - name: Launch instances
      gce:
          instance_names: dev
          machine_type: "{{ machine_type }}"
          image: "{{ image }}"
          service_account_email: "{{ service_account_email }}"
          credentials_file: "{{ credentials_file }}"
          project_id: "{{ project_id }}"
          tags: webserver
      register: gce

    - name: Wait for SSH to come up
      wait_for: host={{ item.public_ip }} port=22 delay=10 timeout=60
      with_items: gce.instance_data

    - name: Add host to groupname
      add_host: hostname={{ item.public_ip }} groupname=new_instances
      with_items: gce.instance_data

- 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.

Configuring instances in a group

All of the created instances in GCE are grouped by tag. Since this is a cloud, it's probably best to ignore hostnames and just focus on group management.

Normally we'd also use roles here, but the following example is a simple one. Here we will also use the "gce_net" module to open up access to port 80 on these nodes.

The variables in the 'vars' section could also be kept in a 'vars_files' file or something encrypted with Ansible-vault, if you so choose. This is just a basic example of what is possible:

```
- name: Setup web servers
  hosts: tag_webserver
  gather_facts: no

  vars:
    machine_type: n1-standard-1 # default
    image: debian-7
    service_account_email: unique-id@developer.gserviceaccount.com
    credentials_file: /path/to/project.json
    project_id: project-id

  roles:

    - name: Install lighttpd
      apt: pkg=lighttpd state=installed
      sudo: True

    - name: Allow HTTP
      local_action: gce_net
      args:
        fwname: "all-http"
        name: "default"
        allowed: "tcp:80"
        state: "present"
        service_account_email: "{{ service_account_email }}"
        credentials_file: "{{ credentials_file }}"
        project_id: "{{ project_id }}"
```

By pointing your browser to the IP of the server, you should see a page welcoming you.

Upgrades to this documentation are welcome, hit the github link at the top right of this page if you would like to make additions!

CloudStack Cloud Guide

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 and are designed to use idempotence and have been created, tested and are maintained by the community.

Note: Some of the modules will require domain admin or root admin privileges.

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.python.org/pypi/cs.

You'll need this Python module installed on the execution host, usually your workstation.

```
$ pip install cs
```

Note: cs also includes a command line interface for ad-hoc interaction with the CloudStack API e.g. `$ cs listVirtualMachines state=Running`.

Limitations and Known Issues

VPC support is not yet fully implemented and tested. The community is working on the VPC integration.

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
```

Note: The section `[cloudstack]` is the default section. `CLOUDSTACK_REGION` environment variable can be used to define the default section.

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

[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
```

Hint: 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 }}"
    with_items:
      - exoscale
      - exmaple_cloud_one
      - exmaple_cloud_two
```

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.

The 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 specifing 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.yaml
---
- 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') }}"
      with_items: 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 `connetion=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 exisiting 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.

Note: The public IP addresses must have been acquired in advance, also see `cs_ip_address`

Note: 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:
```

```
      name: "{{ item }}"
    with_items:
      - default
      - web

  - name: add inbound SSH to security group default
    cs_securitygroup_rule:
      security_group: default
      start_port: "{{ item }}"
      end_port: "{{ item }}"
    with_items:
      - 22

  - name: add inbound TCP rules to security group web
    cs_securitygroup_rule:
      security_group: web
      start_port: "{{ item }}"
      end_port: "{{ item }}"
    with_items:
      - 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: assing 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 accassing 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.

Using Vagrant and Ansible

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 for more information.

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.verbose = "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.

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
```

Advanced Usages

The "Tips and Tricks" chapter of the Ansible Provisioner documentation provides detailed information about more advanced Ansible features like:

- how to parallely execute a playbook in a multi-machine environment
- how to integrate a local `ansible.cfg` configuration file

See also:

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

Playbooks An introduction to playbooks

Continuous Delivery and Rolling Upgrades

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.

Note: 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.

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

  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: lbservers

  roles:
  - haproxy

# Configure and deploy the Nagios monitoring node(s).
- hosts: monitoring

  roles:
  - base-apache
  - nagios
```

Note: If you're not familiar with terms like playbooks and plays, you should review *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.

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 *Playbook Roles and Include Statements* section.

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 `lbservers` 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.

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_upgrade.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 webservers, 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 }}"
  with_items: groups.monitoring

- name: disable the server in haproxy
  shell: echo "disable server myapplb/{{ inventory_hostname }}" | socat stdio /var/
↪lib/haproxy/stats
  delegate_to: "{{ item }}"
  with_items: groups.lbservers
```

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 `with_items` 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 myapplb/{{ inventory_hostname }}" | socat stdio /var/
↪lib/haproxy/stats
  delegate_to: "{{ item }}"
  with_items: groups.lbservers
```

```
- name: re-enable nagios alerts
  nagios: action=enable_alerts host={{ inventory_hostname }} services=webserver
  delegate_to: "{{ item }}"
  with_items: groups.monitoring
```

Again, if you were using a Netscaler or F5 or Elastic Load Balancer, you would just substitute in the appropriate modules instead.

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 *About 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!

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.

See also:

lamp_haproxy example The lamp_haproxy example discussed here.

Playbooks An introduction to playbooks

Playbook Roles and Include Statements An introduction to playbook roles

Variables An introduction to Ansible variables

Ansible.com: Continuous Delivery An introduction to Continuous Delivery with Ansible

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.

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'
```

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.

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 fromm 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:
   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
```

Pending topics may include: Docker, Jenkins, Google Compute Engine, Linode/DigitalOcean, Continuous Deployment, and more.

CHAPTER

SEVEN

DEVELOPER INFORMATION

Ansible 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.

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.

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.

Developing Modules

Topics

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 in one of several ways which we'll go into as we work through this tutorial.

See modules for a list of various ones developed in core.

Modules can be written in any language and are found in the path specified by `ANSIBLE_LIBRARY` or the `--module-path` command line option.

By default, everything that ships with Ansible is pulled from its source tree, but additional paths can be added.

The directory i:file:*./library*, alongside your top level *playbooks*, is also automatically added as a search directory.

Should you develop an interesting Ansible module, consider sending a pull request to the modules-extras project. There's also a core repo for more established and widely used modules. "Extras" modules may be promoted to core periodically, but there's no fundamental difference in the end - both ship with Ansible, all in one package, regardless of how you acquire Ansible.

Tutorial

Let's build a very-basic module to get and set the system time. For starters, let's build a module that just outputs the current time.

We are going to use Python here but any language is possible. Only File I/O and outputting to standard out are required. So, bash, C++, clojure, Python, Ruby, whatever you want is fine.

Now Python Ansible modules contain some extremely powerful shortcuts (that all the core modules use) but first we are going to build a module the very hard way. The reason we do this is because modules written in any language OTHER than Python are going to have to do exactly this. We'll show the easy way later.

So, here's an example. You would never really need to build a module to set the system time, the 'command' module could already be used to do this.

Reading the modules that come with Ansible (linked above) is a great way to learn how to write modules. Keep in mind, though, that some modules in Ansible's source tree are internalisms, so look at service or yum, and don't stare too close into things like `async_wrapper` or you'll turn to stone. Nobody ever executes `async_wrapper` directly.

Ok, let's get going with an example. We'll use Python. For starters, save this as a file named `timetest.py`:

```
#!/usr/bin/python

import datetime
import json

date = str(datetime.datetime.now())
print json.dumps({
    "time" : date
})
```

Testing Modules

There's a useful test script in the source checkout for Ansible:

```
git clone git://github.com/ansible/ansible.git --recursive
source ansible/hacking/env-setup
```

For instructions on setting up Ansible from source, please see intro_installation.

Let's run the script you just wrote with that:

```
ansible/hacking/test-module -m ./timetest.py
```

You should see output that looks something like this:

```
{'time': '2012-03-14 22:13:48.539183'}
```

If you did not, you might have a typo in your module, so recheck it and try again.

Reading Input

Let's modify the module to allow setting the current time. We'll do this by seeing if a key value pair in the form *time=<string>* is passed in to the module.

Ansible internally saves arguments to an arguments file. So we must read the file and parse it. The arguments file is just a string, so any form of arguments are legal. Here we'll do some basic parsing to treat the input as key=value.

The example usage we are trying to achieve to set the time is:

```
time time="March 14 22:10"
```

If no time parameter is set, we'll just leave the time as is and return the current time.

Note: This is obviously an unrealistic idea for a module. You'd most likely just use the command module. However, it makes for a decent tutorial.

Let's look at the code. Read the comments as we'll explain as we go. Note that this is highly verbose because it's intended as an educational example. You can write modules a lot shorter than this:

```
#!/usr/bin/python

# import some python modules that we'll use.  These are all
# available in Python's core

import datetime
import sys
```

```
import json
import os
import shlex

# read the argument string from the arguments file
args_file = sys.argv[1]
args_data = file(args_file).read()

# For this module, we're going to do key=value style arguments.
# Modules can choose to receive json instead by adding the string:
#   WANT_JSON
# Somewhere in the file.
# Modules can also take free-form arguments instead of key-value or json
# but this is not recommended.

arguments = shlex.split(args_data)
for arg in arguments:

    # ignore any arguments without an equals in it
    if "=" in arg:

        (key, value) = arg.split("=")

        # if setting the time, the key 'time'
        # will contain the value we want to set the time to

        if key == "time":

            # now we'll affect the change.  Many modules
            # will strive to be 'idempotent', meaning they
            # will only make changes when the desired state
            # expressed to the module does not match
            # the current state.  Look at 'service'
            # or 'yum' in the main git tree for an example
            # of how that might look.

            rc = os.system("date -s \"%s\"" % value)

            # always handle all possible errors
            #
            # when returning a failure, include 'failed'
            # in the return data, and explain the failure
            # in 'msg'.  Both of these conventions are
            # required however additional keys and values
            # can be added.

            if rc != 0:
                print json.dumps({
                    "failed" : True,
                    "msg"    : "failed setting the time"
                })
                sys.exit(1)

            # when things do not fail, we do not
            # have any restrictions on what kinds of
            # data are returned, but it's always a
            # good idea to include whether or not
            # a change was made, as that will allow
            # notifiers to be used in playbooks.

            date = str(datetime.datetime.now())
            print json.dumps({
                "time" : date,
```

```
                "changed" : True
            })
            sys.exit(0)

# if no parameters are sent, the module may or
# may not error out, this one will just
# return the time

date = str(datetime.datetime.now())
print json.dumps({
    "time" : date
})
```

Let's test that module:

```
ansible/hacking/test-module -m ./timetest.py -a "time=\"March 14 12:23\""
```

This should return something like:

```
{"changed": true, "time": "2012-03-14 12:23:00.000307"}
```

Binary Modules Input

Support for binary modules was added in Ansible 2.2. When Ansible detects a binary module, it will proceed to supply the argument input as a file on `argv[1]` that is formatted as JSON. The JSON contents of that file would resemble something similar to the following payload for a module accepting the same arguments as the `ping` module:

```
{
    "data": "pong",
    "_ansible_verbosity": 4,
    "_ansible_diff": false,
    "_ansible_debug": false,
    "_ansible_check_mode": false,
    "_ansible_no_log": false
}
```

Module Provided 'Facts'

The setup module that ships with Ansible provides many variables about a system that can be used in playbooks and templates. However, it's possible to also add your own facts without modifying the system module. To do this, just have the module return a *ansible_facts* key, like so, along with other return data:

```
{
    "changed" : True,
    "rc" : 5,
    "ansible_facts" : {
        "leptons" : 5000,
        "colors" : {
            "red"   : "FF0000",
            "white" : "FFFFFF"
        }
    }
}
```

These 'facts' will be available to all statements called after that module (but not before) in the playbook. A good idea might be to make a module called 'site_facts' and always call it at the top of each playbook, though we're always open to improving the selection of core facts in Ansible as well.

Common Module Boilerplate

As mentioned, if you are writing a module in Python, there are some very powerful shortcuts you can use. Modules are still transferred as one file, but an arguments file is no longer needed, so these are not only shorter in terms of code, they are actually FASTER in terms of execution time.

Rather than mention these here, the best way to learn is to read some of the source of the modules that come with Ansible.

The 'group' and 'user' modules are reasonably non-trivial and showcase what this looks like.

Key parts include always importing the boilerplate code from `ansible.module_utils.basic` like this:

```
from ansible.module_utils.basic import AnsibleModule
if __name__ == '__main__':
    main()
```

Note: Prior to Ansible-2.1.0, importing only what you used from `ansible.module_utils.basic` did not work. You needed to use a wildcard import like this:

```
from ansible.module_utils.basic import *
```

And instantiating the module class like:

```
def main():
    module = AnsibleModule(
        argument_spec = dict(
            state      = dict(default='present', choices=['present', 'absent']),
            name       = dict(required=True),
            enabled    = dict(required=True, type='bool'),
            something = dict(aliases=['whatever'])
        )
    )
```

The `AnsibleModule` provides lots of common code for handling returns, parses your arguments for you, and allows you to check inputs.

Successful returns are made like this:

```
module.exit_json(changed=True, something_else=12345)
```

And failures are just as simple (where *msg* is a required parameter to explain the error):

```
module.fail_json(msg="Something fatal happened")
```

There are also other useful functions in the module class, such as `module.sha1(path)()`. See `lib/ansible/module_utils/basic.py` in the source checkout for implementation details.

Again, modules developed this way are best tested with the `hacking/test-module` script in the git source checkout. Because of the magic involved, this is really the only way the scripts can function outside of Ansible.

If submitting a module to Ansible's core code, which we encourage, use of `AnsibleModule` is required.

Check Mode

New in version 1.1.

Modules may optionally support check mode. If the user runs Ansible in check mode, the module should try to predict whether changes will occur.

For your module to support check mode, you must pass `supports_check_mode=True` when instantiating the AnsibleModule object. The AnsibleModule.check_mode attribute will evaluate to True when check mode is enabled. For example:

```
module = AnsibleModule(
    argument_spec = dict(...),
    supports_check_mode=True
)

if module.check_mode:
    # Check if any changes would be made but don't actually make those changes
    module.exit_json(changed=check_if_system_state_would_be_changed())
```

Remember that, as module developer, you are responsible for ensuring that no system state is altered when the user enables check mode.

If your module does not support check mode, when the user runs Ansible in check mode, your module will simply be skipped.

Common Pitfalls

You should also never do this in a module:

```
print "some status message"
```

Because the output is supposed to be valid JSON.

Modules must not output anything on standard error, because the system will merge standard out with standard error and prevent the JSON from parsing. Capturing standard error and returning it as a variable in the JSON on standard out is fine, and is, in fact, how the command module is implemented.

If a module returns stderr or otherwise fails to produce valid JSON, the actual output will still be shown in Ansible, but the command will not succeed.

Always use the hacking/test-module script when developing modules and it will warn you about these kind of things.

Conventions/Recommendations

As a reminder from the example code above, here are some basic conventions and guidelines:

- If the module is addressing an object, the parameter for that object should be called 'name' whenever possible, or accept 'name' as an alias.
- If you have a company module that returns facts specific to your installations, a good name for this module is *site_facts*.
- Modules accepting boolean status should generally accept 'yes', 'no', 'true', 'false', or anything else a user may likely throw at them. The AnsibleModule common code supports this with "type='bool"'.
- Include a minimum of dependencies if possible. If there are dependencies, document them at the top of the module file, and have the module raise JSON error messages when the import fails.
- Modules must be self-contained in one file to be auto-transferred by ansible.
- If packaging modules in an RPM, they only need to be installed on the control machine and should be dropped into /usr/share/ansible. This is entirely optional and up to you.
- Modules must output valid JSON only. The toplevel return type must be a hash (dictionary) although they can be nested. Lists or simple scalar values are not supported, though they can be trivially contained inside a dictionary.

- In the event of failure, a key of 'failed' should be included, along with a string explanation in 'msg'. Modules that raise tracebacks (stacktraces) are generally considered 'poor' modules, though Ansible can deal with these returns and will automatically convert anything unparseable into a failed result. If you are using the AnsibleModule common Python code, the 'failed' element will be included for you automatically when you call 'fail_json'.
- Return codes from modules are actually not significant, but continue on with 0=success and non-zero=failure for reasons of future proofing.
- As results from many hosts will be aggregated at once, modules should return only relevant output. Returning the entire contents of a log file is generally bad form.

Documenting Your Module

All modules included in the CORE distribution must have a `DOCUMENTATION` string. This string MUST be a valid YAML document which conforms to the schema defined below. You may find it easier to start writing your `DOCUMENTATION` string in an editor with YAML syntax highlighting before you include it in your Python file.

Example

See an example documentation string in the checkout under examples/DOCUMENTATION.yml.

Include it in your module file like this:

```
#!/usr/bin/python
# Copyright header....

DOCUMENTATION = '''
---
module: modulename
short_description: This is a sentence describing the module
# ... snip ...
'''
```

If an argument takes both C(True)/C(False) and C(Yes)/C(No), the documentation should use C(True) and C(False).

The `description`, and `notes` fields support formatting with some special macros.

These formatting functions are `U()`, `M()`, `I()`, and `C()` for URL, module, italic, and constant-width respectively. It is suggested to use `C()` for file and option names, and `I()` when referencing parameters; module names should be specified as `M(module)`.

Examples (which typically contain colons, quotes, etc.) are difficult to format with YAML, so these must be written in plain text in an `EXAMPLES` string within the module like this:

```
EXAMPLES = '''
- action: modulename opt1=arg1 opt2=arg2
'''
```

The EXAMPLES section, just like the documentation section, is required in all module pull requests for new modules.

The RETURN section documents what the module returns. For each value returned, provide a `description`, in what circumstances the value is `returned`, the `type` of the value and a `sample`. For example, from the `copy` module:

```
RETURN = '''
dest:
    description: destination file/path
    returned: success
    type: string
```

```
    sample: "/path/to/file.txt"
src:
    description: source file used for the copy on the target machine
    returned: changed
    type: string
    sample: "/home/httpd/.ansible/tmp/ansible-tmp-1423796390.97-147729857856000/
↪source"
md5sum:
    description: md5 checksum of the file after running copy
    returned: when supported
    type: string
    sample: "2a5aeecc61dc98c4d780b14b330e3282"
...
'''
```

Building & Testing

Put your completed module file into the 'library' directory and then run the command: `make webdocs`. The new 'modules.html' file will be built and appear in the 'docsite/' directory.

Tip: If you're having a problem with the syntax of your YAML you can validate it on the YAML Lint website.

Tip: You can set the environment variable ANSIBLE_KEEP_REMOTE_FILES=1 on the controlling host to prevent ansible from deleting the remote files so you can debug your module.

Debugging AnsibleModule-based modules

Tip: If you're using the `hacking/test-module` script then most of this is taken care of for you. If you need to do some debugging of the module on the remote machine that the module will actually run on or when the module is used in a playbook then you may need to use this information instead of relying on test-module.

Starting with Ansible-2.1.0, AnsibleModule-based modules are put together as a zip file consisting of the module file and the various python module boilerplate inside of a wrapper script instead of as a single file with all of the code concatenated together. Without some help, this can be harder to debug as the file needs to be extracted from the wrapper in order to see what's actually going on in the module. Luckily the wrapper script provides some helper methods to do just that.

If you are using Ansible with the `ANSIBLE_KEEP_REMOTE_FILES` environment variables to keep the remote module file, here's a sample of how your debugging session will start:

```
$ ANSIBLE_KEEP_REMOTE_FILES=1 ansible localhost -m ping -a 'data=debugging_session
↪' -vvv
<127.0.0.1> ESTABLISH LOCAL CONNECTION FOR USER: badger
<127.0.0.1> EXEC /bin/sh -c '( umask 77 && mkdir -p "` echo $HOME/.ansible/tmp/
↪ansible-tmp-1461434734.35-235318071810595 `" && echo "` echo $HOME/.ansible/tmp/
↪ansible-tmp-1461434734.35-235318071810595 `" )'
<127.0.0.1> PUT /var/tmp/tmpjdbJ1w TO /home/badger/.ansible/tmp/ansible-tmp-
↪1461434734.35-235318071810595/ping
<127.0.0.1> EXEC /bin/sh -c 'LANG=en_US.UTF-8 LC_ALL=en_US.UTF-8 LC_MESSAGES=en_US.
↪UTF-8 /usr/bin/python /home/badger/.ansible/tmp/ansible-tmp-1461434734.35-
↪235318071810595/ping'
localhost | SUCCESS => {
    "changed": false,
    "invocation": {
```

```
        "module_args": {
            "data": "debugging_session"
        },
        "module_name": "ping"
    },
    "ping": "debugging_session"
}
```

Setting `ANSIBLE_KEEP_REMOTE_FILE` to `1` tells Ansible to keep the remote module files instead of deleting them after the module finishes executing. Giving Ansible the `-vvv` optin makes Ansible more verbose. That way it prints the file name of the temporary module file for you to see.

If you want to examine the wrapper file you can. It will show a small python script with a large, base64 encoded string. The string contains the module that is going to be executed. Run the wrapper's explode command to turn the string into some python files that you can work with:

```
$ python /home/badger/.ansible/tmp/ansible-tmp-1461434734.35-235318071810595/ping␣
↪explode
Module expanded into:
/home/badger/.ansible/tmp/ansible-tmp-1461434734.35-235318071810595/debug_dir
```

When you look into the debug_dir you'll see a directory structure like this:

```
- ansible_module_ping.py
- args
- ansible
    - __init__.py
    - module_utils
        - basic.py
        - __init__.py
```

- `ansible_module_ping.py` is the code for the module itself. The name is based on the name of the module with a prefix so that we don't clash with any other python module names. You can modify this code to see what effect it would have on your module.
- The `args` file contains a JSON string. The string is a dictionary containing the module arguments and other variables that Ansible passes into the module to change it's behaviour. If you want to modify the parameters that are passed to the module, this is the file to do it in.
- The `ansible` directory contains code from `ansible.module_utils` that is used by the module. Ansible includes files for any *:module:'ansible.module_utils* imports in the module but not no files from any other module. So if your module uses `ansible.module_utils.url` Ansible will include it for you, but if your module includes `requests` then you'll have to make sure that the python requests library is installed on the system before running the module. You can modify files in this directory if you suspect that the module is having a problem in some of this boilerplate code rather than in the module code you have written.

Once you edit the code or arguments in the exploded tree you need some way to run it. There's a separate wrapper subcommand for this:

```
$ python /home/badger/.ansible/tmp/ansible-tmp-1461434734.35-235318071810595/ping␣
↪execute
{"invocation": {"module_args": {"data": "debugging_session"}}, "changed": false,
↪"ping": "debugging_session"}
```

This subcommand takes care of setting the PYTHONPATH to use the exploded `debug_dir/ansible/module_utils` directory and invoking the script using the arguments in the `args` file. You can continue to run it like this until you understand the problem. Then you can copy it back into your real module file and test that the real module works via **`ansible`** or **`ansible-playbook`**.

Note: The wrapper provides one more subcommand, `excommunicate`. This subcommand is very similar to `execute` in that it invokes the exploded module on the arguments in the `args`. The way it does this is different,

however. `excommunicate` imports the `main()` function from the module and then calls that. This makes excommunicate execute the module in the wrapper's process. This may be useful for running the module under some graphical debuggers but it is very different from the way the module is executed by Ansible itself. Some modules may not work with `excommunicate` or may behave differently than when used with Ansible normally. Those are not bugs in the module; they're limitations of `excommunicate`. Use at your own risk.

Module Paths

If you are having trouble getting your module "found" by ansible, be sure it is in the `ANSIBLE_LIBRARY` environment variable.

If you have a fork of one of the ansible module projects, do something like this:

```
ANSIBLE_LIBRARY=~/ansible-modules-core:~/ansible-modules-extras
```

And this will make the items in your fork be loaded ahead of what ships with Ansible. Just be sure to make sure you're not reporting bugs on versions from your fork!

To be safe, if you're working on a variant on something in Ansible's normal distribution, it's not a bad idea to give it a new name while you are working on it, to be sure you know you're pulling your version.

Getting Your Module Into Ansible

High-quality modules with minimal dependencies can be included in Ansible, but modules (just due to the programming preferences of the developers) will need to be implemented in Python and use the AnsibleModule common code, and should generally use consistent arguments with the rest of the program. Stop by the mailing list to inquire about requirements if you like, and submit a github pull request to the extras project. Included modules will ship with ansible, and also have a chance to be promoted to 'core' status, which gives them slightly higher development priority (though they'll work in exactly the same way).

Module checklist

The following checklist items are important guidelines for people who want to contribute to the development of modules to Ansible on GitHub. Please read the guidelines before you submit your PR/proposal.

- The shebang should always be `#!/usr/bin/python`, this allows ansible_python_interpreter to work
- Modules must be written to support Python 2.4. If this is not possible, required minimum python version and rationale should be explained in the requirements section in DOCUMENTATION.
- Modules must be written to use proper Python-3 syntax. At some point in the future we'll come up with rules for running on Python-3 but we're not there yet. See developing_modules_python3 for help on how to do this.
- **Documentation: Make sure it exists**
  - Module documentation should briefly and accurately define what each module and option does, and how it works with others in the underlying system. Documentation should be written for broad audience–readable both by experts and non-experts. This documentation is not meant to teach a total novice, but it also should not be reserved for the Illuminati (hard balance).
  - If an argument takes both C(True)/C(False) and C(Yes)/C(No), the documentation should use C(True) and C(False).
  - Descriptions should always start with a capital letter and end with a full stop. Consistency always helps.
  - The *required* setting is only required when true, otherwise it is assumed to be false.
  - If *required* is false/missing, *default* may be specified (assumed 'null' if missing). Ensure that the default parameter in docs matches default parameter in code.

  - Documenting *default* is not needed for *required: true*.
  - Remove unnecessary doc like *aliases: []* or *choices: []*.
  - Do not use Boolean values in a choice list . For example, in the list *choices: ['no', 'verify', 'always]*, 'no' will be interpreted as a Boolean value (you can check basic.py for BOOLEANS_* constants to see the full list of Boolean keywords). If your option actually is a boolean, just use *type=bool*; there is no need to populate 'choices'.
  - For new modules or options in a module add version_added. The version should match the value of the current development version and is a string (not a float), so be sure to enclose it in quotes.
  - Verify that arguments in doc and module spec dict are identical.
  - For password / secret arguments no_log=True should be set.
  - Requirements should be documented, using the *requirements=[]* field.
  - Author should be set, with their name and their github id, at the least.
  - Ensure that you make use of U() for urls, C() for files and options, I() for params, M() for modules.
  - If an optional parameter is sometimes required this need to be reflected in the documentation, e.g. "Required when C(state=present)."
  - Verify that a GPL 3 License header is included.
  - Does module use check_mode? Could it be modified to use it? Document it. Documentation is everyone's friend.
  - Examples–include them whenever possible and make sure they are reproducible.
  - Document the return structure of the module. Refer to *Common Return Values* and module_documenting for additional information.

- **Predictable user interface: This is a particularly important section as it is also an area where we need significant impr**
  - Name consistency across modules (we've gotten better at this, but we still have many deviations).
  - Declarative operation (not CRUD)–this makes it easy for a user not to care what the existing state is, just about the final state. `started/stopped`, `present/absent`–don't overload options too much. It is preferable to add a new, simple option than to add choices/states that don't fit with existing ones.
  - Keep options small, having them take large data structures might save us a few tasks, but adds a complex requirement that we cannot easily validate before passing on to the module.
  - Allow an "expert mode". This may sound like the absolute opposite of the previous one, but it is always best to let expert users deal with complex data. This requires different modules in some cases, so that you end up having one (1) expert module and several 'piecemeal' ones (ec2_vpc_net?). The reason for this is not, as many users express, because it allows a single task and keeps plays small (which just moves the data complexity into vars files, leaving you with a slightly different structure in another YAML file). It does, however, allow for a more 'atomic' operation against the underlying APIs and services.

- **Informative responses: Please note, that for >= 2.0, it is required that return data to be documented.**
  - Always return useful data, even when there is no change.
  - Be consistent about returns (some modules are too random), unless it is detrimental to the state/action.
  - Make returns reusable–most of the time you don't want to read it, but you do want to process it and re-purpose it.
  - Return diff if in diff mode. This is not required for all modules, as it won't make sense for certain ones, but please attempt to include this when applicable).

- **Code: This applies to all code in general, but often seems to be missing from modules, so please keep the following in r**
  - Validate upfront–fail fast and return useful and clear error messages.
  - Defensive programming–modules should be designed simply enough that this should be easy. Modules should always handle errors gracefully and avoid direct stacktraces. Ansible deals with this better in 2.0 and returns them in the results.
  - Fail predictably–if we must fail, do it in a way that is the most expected. Either mimic the underlying tool or the general way the system works.
  - Modules should not do the job of other modules, that is what roles are for. Less magic is more.
  - Don't reinvent the wheel. Part of the problem is that code sharing is not that easy nor documented, we also need to expand our base functions to provide common patterns (retry, throttling, etc).
  - Support check mode. This is not required for all modules, as it won't make sense for certain ones, but please attempt to include this when applicable). For more information, refer to *Check Mode As A Drift Test* and *Check Mode ("Dry Run")*.
- **Exceptions: The module must handle them. (exceptions are bugs)**
  - Give out useful messages on what you were doing and you can add the exception message to that.
  - Avoid catchall exceptions, they are not very useful unless the underlying API gives very good error messages pertaining the attempted action.
- **Module-dependent guidelines: Additional module guidelines may exist for certain families of modules.**
  - **Be sure to check out the modules themselves for additional information.**
    - Amazon: https://github.com/ansible/ansible-modules-extras/blob/devel/cloud/amazon/GUIDELINES.md
  - Modules should make use of the "extends_documentation_fragment" to ensure documentation available. For example, the AWS module should include:

```
extends_documentation_fragment:
    - aws
    - ec2
```

- The module must not use sys.exit() –> use fail_json() from the module object.
- Import custom packages in try/except and handled with fail_json() in main() e.g.:

```
try:
    import foo
    HAS_LIB=True
except:
    HAS_LIB=False
```

- The return structure should be consistent, even if NA/None are used for keys normally returned under other options.
- Are module actions idempotent? If not document in the descriptions or the notes.
- Import module snippets *from ansible.module_utils.basic import* * at the bottom, conserves line numbers for debugging.
- The module must have a *main* function that wraps the normal execution.
- Call your `main()` from a conditional so that it would be possible to import them into unittests in the future example:

```
if __name__ == '__main__':
    main()
```

- Try to normalize parameters with other modules, you can have aliases for when user is more familiar with underlying API name for the option
- Being pep8 compliant is nice, but not a requirement. Specifically, the 80 column limit now hinders readability more that it improves it
- Avoid '*action/command*', they are imperative and not declarative, there are other ways to express the same thing
- Do not add *list* or *info* state options to an existing module - create a new *_facts* module.
- If you are asking 'how can I have a module execute other modules' ... you want to write a role
- Return values must be able to be serialized as json via the python stdlib json library. basic python types (strings, int, dicts, lists, etc) are serializable. A common pitfall is to try returning an object via exit_json(). Instead, convert the fields you need from the object into the fields of a dictionary and return the dictionary.
- When fetching URLs, please use either fetch_url or open_url from ansible.module_utils.urls rather than urllib2; urllib2 does not natively verify TLS certificates and so is insecure for https.

Windows modules checklist

- Favour native powershell and .net ways of doing things over calls to COM libraries or calls to native executables which may or may not be present in all versions of windows
- modules are in powershell (.ps1 files) but the docs reside in same name python file (.py)
- look at ansible/lib/ansible/module_utils/powershell.ps1 for common code, avoid duplication
- Ansible uses strictmode version 2.0 so be sure to test with that enabled
- start with:

```
#!powershell
```

then:

```
<GPL header>
```

then:

```
# WANT_JSON
# POWERSHELL_COMMON
```

then, to parse all arguments into a variable modules generally use:

```
$params = Parse-Args $args
```

- **Arguments:**
  - Try and use state present and state absent like other modules
  - You need to check that all your mandatory args are present. You can do this using the builtin Get-AnsibleParam function.
  - Required arguments:

```
$package =  Get-AnsibleParam -obj $params -name name -failifempty $true
```

  - Required arguments with name validation:

```
$state = Get-AnsibleParam -obj $params -name "State" -ValidateSet
↪"Present","Absent" -resultobj $resultobj -failifempty $true
```

  - Optional arguments with name validation:

```
$state = Get-AnsibleParam -obj $params -name "State" -default "Present
↪" -ValidateSet "Present","Absent"
```

  - the If "FailIfEmpty" is true, the resultobj parameter is used to specify the object returned to fail-json. You can also override the default message using $emptyattributefailmessage (for missing required attributes) and $ValidateSetErrorMessage (for attribute validation errors)
  - Look at existing modules for more examples of argument checking.

- **Results**
  - The result object should allways contain an attribute called changed set to either $true or $false
  - Create your result object like this:

```
$result = New-Object psobject @{
changed = $false
other_result_attribute = $some_value
};

If all is well, exit with a
Exit-Json $result
```

  - Ensure anything you return, including errors can be converted to json.
  - Be aware that because exception messages could contain almost anything.
  - ConvertTo-Json will fail if it encounters a trailing in a string.
  - If all is not well use Fail-Json to exit.
- Have you tested for powershell 3.0 and 4.0 compliance?

Deprecating and making module aliases

Starting in 1.8, you can deprecate modules by renaming them with a preceding _, i.e. old_cloud.py to _old_cloud.py. This keeps the module available, but hides it from the primary docs and listing.

You can also rename modules and keep an alias to the old name by using a symlink that starts with _. This example allows the stat module to be called with fileinfo, making the following examples equivalent:

```
EXAMPLES = '''
ln -s stat.py _fileinfo.py
ansible -m stat -a "path=/tmp" localhost
ansible -m fileinfo -a "path=/tmp" localhost
'''
```

See also:

modules Learn about available modules

Developing Plugins Learn about developing plugins

Python API Learn about the Python API for playbook and task execution

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

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.

- a10.py - Utilities used by the a10_server module to manage A10 Networks devices.
- api.py - Adds shared support for generic API modules.
- asa.py - Module support utilities for managing Cisco ASA network devices.
- azure_rm_common.py - Definitions and utilities for Microsoft Azure Resource Manager template deployments.
- basic.py - General definitions and helper utilites for Ansible modules.
- cloudstack.py - Utilities for CloudStack modules.
- database.py - Miscellaneous helper functions for PostGRES and MySQL
- docker_common.py - Definitions and helper utilites for modules working with Docker.
- ec2.py - Definitions and utilites for modules working with Amazon EC2
- eos.py - Helper functions for modules working with EOS networking devices.
- f5.py - Helper functions for modules working with F5 networking devices.
- facts.py - Helper functions for modules that return facts.
- gce.py - Definitions and helper functions for modules that work with Google Compute Engine resources.
- ios.py - Definitions and helper functions for modules that manage Cisco IOS networking devices
- iosxr.py - Definitions and helper functions for modules that manage Cisco IOS-XR networking devices
- ismount.py - Contains single helper function that fixes os.path.ismount
- junos.py - Definitions and helper functions for modules that manage Junos networking devices
- known_hosts.py - Utilites for working with known_hosts file
- mysql.py - Allows modules to connect to a MySQL instance
- netcfg.py - Configuration utility functions for use by networking modules
- netcmd.py - Defines commands and comparison operators for use in networking modules
- network.py - Functions for running commands on networking devices
- nxos.py - Contains definitions and helper functions specific to Cisco NXOS networking devices
- 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
- 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.py - Module utils for working with the Six python 2 and 3 compatibility library
- splitter.py - String splitting and manipulation utilites 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
- vyos.py - Definitions and functions for working with VyOS networking

Developing Plugins

Topics

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.

The following types of plugins are available:

- *Callback* plugins enable you to hook into Ansible events for display or logging purposes.
- *Connection* plugins define how to communicate with inventory hosts.
- *Lookup* plugins are used to pull data from an external source.
- *Vars* plugins inject additional variable data into Ansible runs that did not come from an inventory, playbook, or the command line.

This section describes the various types of plugins and how to implement them.

Callback Plugins

Callback plugins enable adding new behaviors to Ansible when responding to events.

Example Callback Plugins

Ansible comes with a number of callback plugins that you can look at for examples. These can be found in lib/ansible/plugins/callback.

The log_plays callback is an example of how to intercept playbook events to a log file, and the mail callback sends email when playbooks complete.

The osx_say callback provided is particularly entertaining – it will respond with computer synthesized speech on OS X in relation to playbook events, and is guaranteed to entertain and/or annoy coworkers.

Configuring Callback Plugins

To activate a callback, drop it in a callback directory as 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*.

Callbacks need to be whitelisted in your *ansible.cfg* file in order to function. For example:

```
#callback_whitelist = timer, mail, myplugin
```

Developing Callback Plugins

Callback plugins are created by creating a new class with the Base(Callbacks) class as the parent:

```
from ansible.plugins.callback import CallbackBase
from ansible import constants as C

class CallbackModule(CallbackBase):
```

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 example shows how Ansible's timer plugin is implemented:

```
# Make coding more python3-ish
from __future__ import (absolute_import, division, print_function)
__metaclass__ = type

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'
    CALLBACK_NEEDS_WHITELIST = True

    def __init__(self):

        super(CallbackModule, self).__init__()

        self.start_time = datetime.now()

    def days_hours_minutes_seconds(self, runtime):
        minutes = (runtime.seconds // 60) % 60
        r_seconds = runtime.seconds - (minutes * 60)
        return runtime.days, runtime.seconds // 3600, minutes, r_seconds

    def playbook_on_stats(self, stats):
```

```
        self.v2_playbook_on_stats(stats)

    def v2_playbook_on_stats(self, stats):
        end_time = datetime.now()
        runtime = end_time - self.start_time
        self._display.display("Playbook run took %s days, %s hours, %s minutes, %s␣
↪seconds" % (self.days_hours_minutes_seconds(runtime)))
```

Note that the CALLBACK_VERSION and CALLBACK_NAME definitons are required. If your callback plugin needs to write to stdout, you should define CALLBACK_TYPE = stdout in the subclass, and then the stdout plugin needs to be configured in *ansible.cfg* to override the default. For example:

```
#stdout_callback = mycallbackplugin
```

Connection Type Plugins

By default, ansible ships with a 'paramiko' SSH, native ssh (just called 'ssh'), 'local' connection type, and there are also some minor players like 'chroot' and 'jail'. 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 intro_getting_started section. Should you want to extend Ansible to support other transports (SNMP? Message bus? Carrier Pigeon?) it's as simple as copying the format of one of the existing modules and dropping it into the connection plugins directory. The value of 'smart' for a connection allows selection of paramiko or openssh based on system capabilities, and chooses 'ssh' if OpenSSH supports ControlPersist, in Ansible 1.2.1 and later. Previous versions did not support 'smart'.

More documentation on writing connection plugins is pending, though you can jump into lib/ansible/plugins/connection and figure things out pretty easily.

Lookup Plugins

Language constructs like "with_fileglob" and "with_items" are implemented via lookup plugins. Just like other plugin types, you can write your own.

More documentation on writing lookup plugins is pending, though you can jump into lib/ansible/plugins/lookup and figure things out pretty easily.

Vars Plugins

Playbook constructs like 'host_vars' and 'group_vars' work via 'vars' plugins. They inject additional variable data into ansible runs that did not come from an inventory, playbook, or command line. Note that variables can also be returned from inventory, so in most cases, you won't need to write or understand vars_plugins.

More documentation on writing vars plugins is pending, though you can jump into lib/ansible/inventory/vars_plugins and figure things out pretty easily.

If you find yourself wanting to write a vars_plugin, it's more likely you should write an inventory script instead.

Filter Plugins

If you want more Jinja2 filters available in a Jinja2 template (filters like to_yaml and to_json are provided by default), they can be extended by writing a filter plugin. Most of the time, when someone comes up with an idea for a new filter they would like to make available in a playbook, we'll just include them in 'core.py' instead.

Jump into lib/ansible/plugins/filter for details.

Distributing Plugins

Plugins are loaded from both Python's site_packages (those that ship with ansible) and a configured plugins directory, which defaults to /usr/share/ansible/plugins, in a subfolder for each plugin type:

```
* action
* lookup
* callback
* connection
* filter
* strategy
* cache
* test
* shell
```

To change this path, edit the ansible configuration file.

In addition, plugins can be shipped in a subdirectory relative to a top-level playbook, in folders named the same as indicated above.

They can also be shipped as part of a role, in a subdirectory named as indicated above. The plugin will be availiable as soon as the role is called.

See also:

modules List of built-in modules

Python API Learn about the Python API for task execution

Developing Dynamic Inventory Sources 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

Developing Dynamic Inventory Sources

Topics

As described in intro_dynamic_inventory, ansible can pull inventory information from dynamic sources, including cloud sources.

How do we write a new one?

Simple! We just create a script or program that can print JSON in the right format when fed the proper arguments. You can do this in any language.

Script Conventions

When the external node script is called with the single argument `--list`, the script must output a JSON encoded hash/dictionary of all the groups to be managed to stdout. Each group's value should be either a hash/dictionary containing a list of each host/IP, potential child groups, and potential group variables, or simply a list of host/IP addresses, like so:

```
{
    "databases"   : {
        "hosts"   : [ "host1.example.com", "host2.example.com" ],
        "vars"    : {
            "a"   : true
        }
    },
    "webservers"  : [ "host2.example.com", "host3.example.com" ],
    "atlanta"     : {
        "hosts"   : [ "host1.example.com", "host4.example.com", "host5.example.com
↪" ],
        "vars"    : {
            "b"   : false
        },
        "children": [ "marietta", "5points" ]
    },
    "marietta"    : [ "host6.example.com" ],
    "5points"     : [ "host7.example.com" ]
}
```

New in version 1.0.

Before version 1.0, each group could only have a list of hostnames/IP addresses, like the webservers, marietta, and 5points groups above.

When called with the arguments `--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. Printing variables is optional, if the script does not wish to do this, printing an empty hash/dictionary is the way to go:

```
{
    "favcolor"   : "red",
    "ntpserver"  : "wolf.example.com",
    "monitoring" : "pack.example.com"
}
```

Tuning the External Inventory Script

New in version 1.3.

The stock inventory script system detailed above works for all versions of Ansible, but calling `--host` for every host can be rather expensive, especially if it involves expensive API calls to a remote subsystem. In Ansible 1.3 or later, if the inventory script returns a top level element called "_meta", it is possible to return all of the host variables in one inventory script call. 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, and also makes client side caching easier to implement for the inventory script.

The data to be added to the top level JSON dictionary looks like this:

```
{

    # results of inventory script as above go here
    # ...

    "_meta" : {
       "hostvars" : {
          "moocow.example.com"     : { "asdf" : 1234 },
          "llama.example.com"      : { "asdf" : 5678 },
       }
    }

}
```

See also:

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

Python API

Topics

Please note that while we make this API available it is not intended for direct consumption, it is here for the support of the Ansible command line tools. We try not to make breaking changes but we reserve the right to do so at any time if it makes sense for the Ansible toolset.

The following documentation is provided for those that still want to use the API directly, but be mindful this is not something the Ansible team supports.

There are several interesting 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 various plugins, and you can plug in inventory data from external data sources. This document covers the execution and Playbook API at a basic level.

If you are looking to use Ansible programmatically from something other than Python, trigger events asynchronously, or have access control and logging demands, take a look at tower as it has a very nice REST API that provides all of these things at a higher level.

Ansible is written in its own API so you have a considerable amount of power across the board. This chapter discusses the Python API. The Python API is very powerful, and is how the all the ansible CLI tools are implemented. In version 2.0 the core ansible got rewritten and the API was mostly rewritten.

Note: Ansible relies on forking processes, as such the API is not thread safe.

Python API 2.0

In 2.0 things get a bit more complicated to start, but you end up with much more discrete and readable classes:

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

import json
from collections import namedtuple
from ansible.parsing.dataloader import DataLoader
from ansible.vars import VariableManager
from ansible.inventory import Inventory
```

```
from ansible.playbook.play import Play
from ansible.executor.task_queue_manager import TaskQueueManager
from ansible.plugins.callback import CallbackBase

class ResultCallback(CallbackBase):
    """A sample callback plugin used for performing an action as results come in

    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)

Options = namedtuple('Options', ['connection', 'module_path', 'forks', 'become',␣
↪'become_method', 'become_user', 'check'])
# initialize needed objects
variable_manager = VariableManager()
loader = DataLoader()
options = Options(connection='local', module_path='/path/to/mymodules', forks=100,␣
↪become=None, become_method=None, become_user=None, check=False)
passwords = dict(vault_pass='secret')

# Instantiate our ResultCallback for handling results as they come in
results_callback = ResultCallback()

# create inventory and pass to var manager
inventory = Inventory(loader=loader, variable_manager=variable_manager, host_list=␣
↪'localhost')
variable_manager.set_inventory(inventory)

# create play with tasks
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}}␣
↪')))
         ]
    )
play = Play().load(play_source, variable_manager=variable_manager, loader=loader)

# actually run it
tqm = None
try:
    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
          )
    result = tqm.run(play)
```

```
finally:
    if tqm is not None:
        tqm.cleanup()
```

Python API pre 2.0

It's pretty simple:

```
import ansible.runner

runner = ansible.runner.Runner(
   module_name='ping',
   module_args='',
   pattern='web*',
   forks=10
)
datastructure = runner.run()
```

The run method returns results per host, grouped by whether they could be contacted or not. Return types are module specific, as expressed in the modules documentation.:

```
{
    "dark" : {
       "web1.example.com" : "failure message"
    },
    "contacted" : {
       "web2.example.com" : 1
    }
}
```

A module can return any type of JSON data it wants, so Ansible can be used as a framework to rapidly build powerful applications and scripts.

Detailed API Example

The following script prints out the uptime information for all hosts:

```
#!/usr/bin/python

import ansible.runner
import sys

# construct the ansible runner and execute on all hosts
results = ansible.runner.Runner(
    pattern='*', forks=10,
    module_name='command', module_args='/usr/bin/uptime',
).run()

if results is None:
   print "No hosts found"
   sys.exit(1)

print "UP ***********"
for (hostname, result) in results['contacted'].items():
    if not 'failed' in result:
        print "%s >>> %s" % (hostname, result['stdout'])

print "FAILED *******"
for (hostname, result) in results['contacted'].items():
```

```
    if 'failed' in result:
        print "%s >>> %s" % (hostname, result['msg'])

print "DOWN *********"
for (hostname, result) in results['dark'].items():
    print "%s >>> %s" % (hostname, result)
```

Advanced programmers may also wish to read the source to ansible itself, for it uses the API (with all available options) to implement the `ansible` command line tools (`lib/ansible/cli/`).

See also:

Developing Dynamic Inventory Sources 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

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.

Modules

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.

Note: 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"}"""
```

Note: 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.

See also:

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.

See also:

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.

Note: 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 frameworkthe 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.

Note: Ansible wraps the zipfile in the Python script for two reasons:

- for compatibility with Python-2.4 and Python-2.6 which have less featureful versions 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.

Warning: 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.

Note: 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`.

Note: `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.

New in version 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)
```

New in version 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.

New in version 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 *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.

See also:

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

Helping Testing PRs

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, so clearing bugs out of the way is one of the best things you can do.

Even if you're not a developer, helping test pull requests for bug fixes and features is still immensely valuable.

This goes for testing new features as well as testing bugfixes.

In many cases, code should add tests that prove it works but 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 that 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 too.

Thankfully helping test ansible is pretty straightforward, assuming you are already used to how ansible works.

Get Started with A Source Checkout

You can do this by checking out ansible, making a test branch off the main one, merging a GitHub issue, testing, and then commenting on that particular issue on GitHub. Here's how:

Note: 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.

First, you will need to configure your testing environment with the necessary tools required to run our test suites. You will need at least:

```
git
python-nosetests (sometimes named python-nose)
python-passlib
python-mock
```

If you want to run the full integration test suite you'll also need the following packages installed:

```
svn
hg
python-pip
gem
```

Second, if you haven't already, clone the Ansible source code from GitHub:

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

Note: If you have previously forked the repository on GitHub, you could also clone it from there.

Note: If updating your repo for testing something module related, use "git rebase origin/devel" and then "git submodule update" to fetch the latest development versions of modules. Skipping the "git submodule update" step will result in versions that will be stale.

Activating The Source Checkout

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 enviromnent variables (along with a few other things), which will be temporarily set as long as your shell session is open.

If you'd like your testing environment to always use the latest source, you could call the command from startup scripts (for example, *.bash_profile*).

Finding A Pull Request and Checking It Out On A Branch

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
```

Note: 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.

Note: 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.

Note: Some users do not create feature branches, which can cause problems when they have multiple, un-related 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.

Finding a Pull Request for Ansible Modules

Ansible modules are in separate repositories, which are managed as Git submodules. Here's a step by step process for checking out a PR for an Ansible extras module, for instance:

1. git clone https://github.com/ansible/ansible.git
2. cd ansible
3. git submodule init
4. git submodule update –recursive [fetches the submodules]
5. cd lib/ansible/modules/extras
6. git fetch origin pull/1234/head:pr/1234 [fetches the specific PR]
7. git checkout pr/1234 [do your testing here]
8. cd /path/to/ansible/clone
9. git submodule update –recursive

For Those About To Test, We Salute You

At this point, you should be ready to begin testing!

If the PR is a bug-fix pull request, the first things to do are to run the suite of unit and integration tests, to ensure the pull request does not break current functionality:

```
# Unit Tests
make tests

# Integration Tests
cd test/integration
make
```

Note: Ansible does provide integration tests for cloud-based modules as well, however we do not recommend using them for some users due to the associated costs from the cloud providers. As such, typically it's better to run specific parts of the integration battery and skip these tests.

Integration tests aren't the end all beat all - in many cases what is fixed might not *HAVE* a test, so determining if it works means checking the functionality of the system and making sure it does what it said it would do.

Pull requests for bug-fixes should reference the bug issue number they are fixing.

We encourage users to provide playbook examples for bugs that show how to reproduce the error, and these playbooks should be used to verify the bugfix does resolve the issue if available. You may wish to also do your own review to poke the corners of the change.

Since some reproducers can be quite involved, you might wish to create a testing directory with the issue # as a sub- directory to keep things organized:

```
mkdir -p testing/XXXX # where XXXX is again the issue # for the original issue or␣
↪PR
cd testing/XXXX
<create files or git clone example playbook repo>
```

While it should go without saying, be sure to read any playbooks before you run them. VMs help with running untrusted content greatly, though a playbook could still do something to your computing resources that you'd rather not like.

Once the files are in place, you can run the provided playbook (if there is one) to test the functionality:

```
ansible-playbook -vvv playbook_name.yml
```

If there's no playbook, you may have to copy and paste playbook snippets or run an ad-hoc command that was pasted in.

Our issue template also included sections for "Expected Output" and "Actual Output", which should be used to gauge the output from the provided examples.

If the pull request resolves the issue, please leave a comment on the pull request, showing the following information:

- "Works for me!"
- The output from *ansible –version*.

In some cases, you may wish to share playbook output from the test run as well.

Example!:

```
Works for me!  Tested on `Ansible 1.7.1`.  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 my toaster started making loud noises!

Output from the toaster looked like this:

   ```
   BLARG
   StrackTrace
   RRRARRGGG
   ```
```

When you are done testing a feature branch, you can remove it with the following command:

```
git branch -D someuser-feature_branch_name
```

We understand some users may be inexperienced with git, or other aspects of the above procedure, so feel free to stop by ansible-devel list for questions and we'd be happy to help answer them.

Releases

Topics

Release Schedule

Ansible is on a 'flexible' 4 month release schedule, sometimes this can be extended if there is a major change that requires a longer cycle (i.e. 2.0 core rewrite). Currently modules get released at the same time as the main Ansible repo, even though they are separated into ansible-modules-core and ansible-modules-extras.

The major features and bugs fixed in a release should be reflected in the CHANGELOG.md, minor ones will be in the commit history (FIXME: add git exmaple to list). When a fix/feature gets added to the *devel* branch it will be part of the next release, some bugfixes can be backported to previous releases and might be part of a minor point release if it is deemed necessary.

Sometimes an RC can be extended by a few days if a bugfix makes a change that can have far reaching consequences, so users have enough time to find any new issues that may stem from this.

Release methods

Ansible normally goes through a 'release candidate', issuing an RC1 for a release, if no major bugs are discovered in it after 5 business days we'll get a final release. Otherwise fixes will be applied and an RC2 will be provided for testing and if no bugs after 2 days, the final release will be made, iterating this last step and incrementing the candidate number as we find major bugs.

Release feature freeze

During the release candidate process, the focus will be on bugfixes that affect the RC, new features will be delayed while we try to produce a final version. Some bugfixes that are minor or don't affect the RC will also be postponed until after the release is finalized.

See also:

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

CHAPTER

EIGHT

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!

CHAPTER

NINE

COMMUNITY INFORMATION & CONTRIBUTING

Ansible is an open source project designed to bring together administrators and developers of all kinds to collaborate on building IT automation solutions that work well for them.

Should you wish to get more involved – whether in terms of just asking a question, helping other users, introducing new people to Ansible, or helping with the software or documentation, we welcome your contributions to the project.

Topics

Ansible Users

I've Got A Question

We're happy to help!

Ansible questions are best asked on the Ansible Google Group Mailing List.

This is a very large high-traffic list for answering questions and sharing tips and tricks. Anyone can join, and email delivery is optional if you just want to read the group online. To cut down on spam, your first post is moderated, though posts are approved quickly.

Please be sure to share any relevant commands you ran, output, and detail, indicate the version of Ansible you are using when asking a question.

Where needed, link to gists or GitHub repos to show examples, rather than sending attachments to the list.

We recommend using Google search to see if a topic has been answered recently, but comments found in older threads may no longer apply, depending on the topic.

Before you post, be sure you are running the latest stable version of Ansible. You can check this by comparing the output of `ansible --version` with the version indicated on PyPi.

Alternatively, you can also join our IRC channel - #ansible on irc.freenode.net. It's a very high traffic channel as well, if you don't get an answer you like, please stop by our mailing list, which is more likely to get attention of core developers since it's asynchronous.

I'd Like To Keep Up With Release Announcements

Release announcements are posted to ansible-project, though if you don't want to keep up with the very active list, you can join the Ansible Announce Mailing List.

This is a low-traffic read-only list, where we'll share release announcements and occasionally links to major Ansible Events around the world.

I'd Like To Help Share and Promote Ansible

You can help share Ansible with others by telling friends and colleagues, writing a blog post, or presenting at user groups (like DevOps groups or the local LUG).

You are also welcome to share slides on speakerdeck, sign up for a free account and tag it "Ansible". On Twitter, you can also share things with #ansible and may wish to follow us.

I'd Like To Help Ansible Move Faster

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, so clearing bugs out of the way is one of the best things you can do.

If you're not a developer, helping test pull requests for bug fixes and features is still immensely valuable. You can do this by checking out ansible, making a test branch off the main one, merging a GitHub issue, testing, and then commenting on that particular issue on GitHub.

I'd Like To Report A Bug

Ansible practices responsible disclosure - if this is a security related bug, email security@ansible.com instead of filing a ticket or posting to the Google Group and you will receive a prompt response.

Bugs related to the core language should be reported to github.com/ansible/ansible 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.

MODULE related bugs however should go to ansible-modules-core or ansible-modules-extras based on the classification of the module. This is listed on the bottom of the docs page for any module.

When filing a bug, please use the issue template to provide all relevant information, regardless of what repo you are filing a ticket against.

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.

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. 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.

It may take some time to get to your report, see our information about priority flags below.

I'd Like To Help With Documentation

Ansible documentation is a community project too!

If you would like to help with the documentation, whether correcting a typo or improving a section, or maybe even documenting a new feature, submit a GitHub pull request to the code that lives in the `docsite/rst` subdirectory of the project for most pages, and there is an "Edit on GitHub" link up on those.

Module documentation is generated from a DOCUMENTATION structure embedded in the source code of each module, which is in either the ansible-modules-core or ansible-modules-extra repos on GitHub, depending on the module. Information about this is always listed on the bottom of the web documentation for each module.

Aside from modules, the main docs are in restructured text format.

If you aren't comfortable with restructured text, you can also open a ticket on GitHub about any errors you spot or sections you would like to see added. For more information on creating pull requests, please refer to the github help guide.

For Current and Prospective Developers

I'd Like To Learn How To Develop on Ansible

If you're new to Ansible and would like to figure out how to work on things, stop by the ansible-devel mailing list and say hi, and we can hook you up.

A great way to get started would be reading over some of the development documentation on the module site, and then finding a bug to fix or small feature to add.

Modules are some of the easiest places to get started.

Contributing Code (Features or Bugfixes)

The Ansible project keeps its source on GitHub at github.com/ansible/ansible for the core application, and two sub repos github.com/ansible/ansible-modules-core and ansible/ansible-modules-extras for module related items. If you need to know if a module is in 'core' or 'extras', consult the web documentation page for that module.

The project takes contributions through github pull requests.

It is usually a good idea to join the ansible-devel list to discuss any large features prior to submission, and this especially helps in avoiding duplicate work or efforts where we decide, upon seeing a pull request for the first time, that revisions are needed. (This is not usually needed for module development, but can be nice for large changes).

Note that we do keep Ansible to a particular aesthetic, so if you are unclear about whether a feature is a good fit or not, having the discussion on the development list is often a lot easier than having to modify a pull request later.

When submitting patches, be sure to run the unit tests first `make tests` and always use, these are the same basic tests that will automatically run on Shippable when creating the PR. There are more in depth tests in the `tests/integration` directory, classified as destructive and non_destructive, run these if they pertain to your modification. They are set up with tags so you can run subsets, some of the tests require cloud credentials and will only run if they are provided. When adding new features or fixing bugs it would be nice to add new tests to avoid regressions. For more information about testing see test/README.md.

In order to keep the history clean and better audit incoming code, we will require resubmission of pull requests that contain merge commits. Use `git pull --rebase` (rather than `git pull`) and `git rebase` (rather than `git merge`). Also be sure to use topic branches to keep your additions on different branches, such that they won't pick up stray commits later.

If you make a mistake you do not need to close your PR, create a clean branch locally and then push to GitHub with `--force` to overwrite the existing branch (permissible in this case as no one else should be using that branch as reference). Code comments won't be lost, they just won't be attached to the existing branch.

We'll then review your contributions and engage with you about questions and so on.

Because we have a very large and active community it may take awhile to get your contributions in! See the notes about priorities in a later section for understanding our work queue. Be patient, your request might not get merged right away, we also try to keep the devel branch more or less usable so we like to examine Pull requests carefully, which takes time.

Patches should always be made against the `devel` branch.

Keep in mind that small and focused requests are easier to examine and accept, having example cases also help us understand the utility of a bug fix or a new feature.

Contributions can be for new features like modules, or to fix bugs you or others have found. If you are interested in writing new modules to be included in the core Ansible distribution, please refer to the module development documentation.

Ansible's aesthetic encourages simple, readable code and consistent, conservatively extending, backwards-compatible improvements. Code developed for Ansible needs to support Python 2.6+, while code in modules must run under Python 2.4 or higher. Please also use a 4-space indent and no tabs, we do not enforce 80 column lines, we are fine with 120-140. We do not take 'style only' requests unless the code is nearly unreadable, we are "PEP8ish", but not strictly compliant.

You can also contribute by testing and revising other requests, especially if it is one you are interested in using. Please keep your comments clear and to the point, courteous and constructive, tickets are not a good place to start discussions (ansible-devel and IRC exist for this).

Tip: To easily run from a checkout, source `./hacking/env-setup` and that's it – no install required. You're now live! For more information see hacking/README.md.

Other Topics

Ansible Staff

Ansible, Inc is a company supporting Ansible and building additional solutions based on Ansible. We also do services and support for those that are interested. We also offer an enterprise web front end to Ansible (see Tower below).

Our most important task however is enabling all the great things that happen in the Ansible community, including organizing software releases of Ansible. For more information about any of these things, contact info@ansible.com

On IRC, you can find us as jimi_c, abadger1999, Tybstar, bcoca, and others. On the mailing list, we post with an @ansible.com address.

Mailing List Information

Ansible has several mailing lists. Your first post to the mailing list will be moderated (to reduce spam), so please allow a day or less for your first post.

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 AnsibleFest coming up, which is our official conference series.

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 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.

Release Numbering

Releases ending in ".0" are major releases and this is where all new features land. Releases ending in another integer, like "0.X.1" and "0.X.2" are dot releases, and these are only going to contain bugfixes.

Typically we don't do dot releases for minor bugfixes (reserving these for larger items), but may occasionally decide to cut dot releases containing a large number of smaller fixes if it's still a fairly long time before the next release comes out.

Releases are also given code names based on Van Halen songs, that no one really uses.

Tower Support Questions

Ansible Tower is a UI, Server, and REST endpoint for Ansible, produced by Ansible, Inc.

If you have a question about Tower, visit support.ansible.com rather than using the IRC channel or the general project mailing list.

IRC Channel

Ansible has several IRC channels on Freenode (irc.freenode.net):

- #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-notices - Mostly bot output from things like Github, etc.

Notes on Priority Flags

Ansible was one of the top 5 projects with the most OSS contributors on GitHub in 2013, and has over 1400 contributors to the project to date, not to mention a very large user community that has downloaded the application well over a million times.

As a result, we have a LOT of incoming activity to process.

In the interest of transparency, we're telling you how we sort incoming requests.

In our bug tracker you'll notice some labels - P1, P2, P3, P4, and P5. These are our internal priority orders that we use to sort tickets.

With some exceptions for easy merges (like documentation typos for instance), we're going to spend most of our time working on P1 and P2 items first, including pull requests. These usually relate to important bugs or features affecting large segments of the userbase. So if you see something categorized "P3 or P4", and it's not appearing to get a lot of immediate attention, this is why.

These labels don't really have definition - they are a simple ordering. However something affecting a major module (yum, apt, etc) is likely to be prioritized higher than a module affecting a smaller number of users.

Since we place a strong emphasis on testing and code review, it may take a few months for a minor feature to get merged.

Don't worry though – we'll also take periodic sweeps through the lower priority queues and give them some attention as well, particularly in the area of new module changes. So it doesn't necessarily mean that we'll be exhausting all of the higher-priority queues before getting to your ticket.

Every bit of effort helps - if you're wishing to expedite the inclusion of a P3 feature pull request for instance, the best thing you can do is help close P2 bug reports.

Community Code of Conduct

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 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 greg@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 our 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

Contributors License Agreement

By contributing you agree that these contributions are your own (or approved by your employer) and you grant a full, complete, irrevocable copyright license to all users and developers of the project, present and future, pursuant to the license of the project.

CHAPTER

TEN

ANSIBLE GALAXY

"Ansible Galaxy" can either refer to a website for sharing and downloading Ansible roles, or a command line tool for managing and creating roles.

Topics

The Website

The website Ansible Galaxy, is a free site for finding, downloading, and sharing community developed Ansible roles. Downloading roles from Galaxy is a great way to jumpstart your automation projects.

Access the Galaxy web site using GitHub OAuth, and to install roles use the 'ansible-galaxy' command line tool included in Ansible 1.4.2 and later.

Read the "About" page on the Galaxy site for more information.

The ansible-galaxy command line tool

The ansible-galaxy command has many different sub-commands for managing roles both locally and at galaxy.ansible.com.

Note: The search, login, import, delete, and setup commands in the Ansible 2.0 version of ansible-galaxy require access to the 2.0 Beta release of the Galaxy web site available at https://galaxy-qa.ansible.com.

Use the `--server` option to access the beta site. For example:

```
$ ansible-galaxy search --server https://galaxy-qa.ansible.com mysql --author␣
↪geerlingguy
```

Additionally, you can define a server in ansible.cfg:

```
[galaxy]
server=https://galaxy-qa.ansible.com
```

Installing Roles

The most obvious use of the ansible-galaxy command is downloading roles from the Ansible Galaxy website:

```
$ ansible-galaxy install username.rolename
```

roles_path

You can specify a particular directory where you want the downloaded roles to be placed:

```
$ ansible-galaxy install username.role -p ~/Code/ansible_roles/
```

This can be useful if you have a master folder that contains ansible galaxy roles shared across several projects. The default is the roles_path configured in your ansible.cfg file (/etc/ansible/roles if not configured).

Installing Multiple Roles From A File

To install multiple roles, the ansible-galaxy CLI can be fed a requirements file. All versions of ansible allow the following syntax for installing roles from the Ansible Galaxy website:

```
$ ansible-galaxy install -r requirements.txt
```

Where the requirements.txt looks like:

```
username1.foo_role
username2.bar_role
```

To request specific versions (tags) of a role, use this syntax in the roles file:

```
username1.foo_role,version
username2.bar_role,version
```

Available versions will be listed on the Ansible Galaxy webpage for that role.

Installing Multiple Roles From Multiple Files

At a basic level, including requirements files allows you to break up bits of configuration policy into smaller files. Role includes pull in roles from other files.

```
ansible-galaxy install -r requirements.yml
```

Content of requirements.yml

```
# from github
- src: yatesr.timezone

- include: webserver.yml
```

Content of the webserver.yml file.

```
# from github
- src: https://github.com/bennojoy/nginx

# from github installing to a relative path
- src: https://github.com/bennojoy/nginx
```

Advanced Control over Role Requirements Files

For more advanced control over where to download roles from, including support for remote repositories, Ansible 1.8 and later support a new YAML format for the role requirements file, which must end in a 'yml' extension. It works like this:

```
ansible-galaxy install -r requirements.yml
```

The extension is important. If the .yml extension is left off, the ansible-galaxy CLI will assume the file is in the "basic" format and will be confused.

And here's an example showing some specific version downloads from multiple sources. In one of the examples we also override the name of the role and download it as something different:

```
 # 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: v1.4

 # 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: 0.1.0
```

As you can see in the above, there are a large amount of controls available to customize where roles can be pulled from, and what to save roles as.

You can also pull down multiple roles from a single source (just make sure that you have a meta/main.yml file at the root level).

```
meta\main.yml
common-role1\tasks\main.yml
common-role2\tasks\main.yml
```

For example, if the above common roles are published to a git repo, you can pull them down using:

```
# multiple roles from the same repo
- src: git@gitlab.company.com:mygroup/ansible-common.git
  name: common-roles
  scm: git
  version: master
```

You could then use these common roles in your plays

```
---
- hosts: webservers
  roles:
    - common-roles/common-role1
    - common-roles/common-role2
```

Roles pulled from galaxy work as with other SCM sourced roles above. To download a role with dependencies, and automatically install those dependencies, the role must be uploaded to the Ansible Galaxy website.

See also:

Playbook Roles and Include Statements All about ansible roles

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

irc.freenode.net #ansible IRC chat channel

Building Role Scaffolding

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 rolename
```

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
```

Note: .travis.yml and tests/ are new in Ansible 2.0

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.

Search for Roles

The search command provides for querying the Galaxy database, allowing for searching 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.
```

Note: The format of results pictured here is new in Ansible 2.0.

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.rolename
    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

The list command shows the name and version of each role installed in 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

The remove command will delete a role from roles_path:

```
$ ansible-galaxy remove username.rolename
```

Authenticate with Galaxy

To use the import, delete and setup commands authentication with Galaxy is required. The login command will authenticate the user,retrieve a token from Galaxy, and store it in the user's home directory.

```
$ 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:
Succesfully logged into Galaxy as dsmith
```

As depicted above, the login command prompts for a GitHub username and password. It does NOT send your password to Galaxy. It actually authenticates with GitHub and creates a personal access token. It then sends the personal access token to Galaxy, which in turn verifies that you are you and returns a Galaxy access token. After authentication completes the GitHub personal access 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. Log into GitHub, go to Settings and click on Personal Access Token to create a token.

Note: The login command in Ansible 2.0 requires using the Galaxy 2.0 Beta site. Use the `--server` option to access https://galaxy-qa.ansible.com. You can also add a *server* definition in the [galaxy] section of your

ansible.cfg file.

Import a Role

Roles can be imported using ansible-galaxy. The import command expects that the user previously authenticated with Galaxy using the login command.

Import any GitHub repo you have access to:

```
$ ansible-galaxy import github_user github_repo
```

By default the command will wait for the role to be imported by Galaxy, 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
```

Use the –branch option to import a specific branch. If not specified, the default branch for the repo will be used.

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 under My Imports.

Note: The import command in Ansible 2.0 requires using the Galaxy 2.0 Beta site. Use the `--server` option to access https://galaxy-qa.ansible.com. You can also add a *server* definition in the [galaxy] section of your ansible.cfg file.

Delete a Role

Remove a role from the Galaxy web site using the delete command. You can delete any role that you have access to in GitHub. The delete command expects that the user previously authenticated with Galaxy using the login command.

```
$ ansible-galaxy delete github_user github_repo
```

This only removes the role from Galaxy. It does not impact the actual GitHub repo.

Note: The delete command in Ansible 2.0 requires using the Galaxy 2.0 Beta site. Use the `--server` option to access https://galaxy-qa.ansible.com. You can also add a *server* definition in the [galaxy] section of your ansible.cfg file.

Setup Travis Integrations

Using the setup command you can enable notifications from travis. The setup command expects that the user previously authenticated with Galaxy using the login command.

```
$ ansible-galaxy setup travis github_user github_repo xxxtravistokenxxx

Added integration for travis github_user/github_repo
```

The setup command requires your Travis token. The Travis 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 calculated hash is stored in Galaxy and used to verify notifications received from Travis.

The setup command enables Galaxy to respond to notifications. Follow the Travis getting started guide to enable the Travis build process for the role repository.

When you create your .travis.yml file add the following to cause Travis to notify Galaxy when a build completes:

```
notifications:
    webhooks: https://galaxy.ansible.com/api/v1/notifications/
```

Note: The setup command in Ansible 2.0 requires using the Galaxy 2.0 Beta site. Use the `--server` option to access https://galaxy-qa.ansible.com. You can also add a *server* definition in the [galaxy] section of your ansible.cfg file.

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 you want disabled. Use the –list option to get the ID.

CHAPTER

ELEVEN

TESTING STRATEGIES

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.

Note: 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.

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).

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
```

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.

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.

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: webservers
  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
      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: webservers
  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.

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
```

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'.

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.

See also:

About Modules All the documentation for Ansible modules

Playbooks An introduction to playbooks

Delegation, Rolling Updates, and Local Actions Delegation, useful for working with loud balancers, clouds, and locally executed steps.

User Mailing List Have a question? Stop by the google group!

irc.freenode.net #ansible IRC chat channel

CHAPTER

TWELVE

FREQUENTLY ASKED QUESTIONS

Here are some commonly-asked questions and their answers.

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 task or play level:

```
environment:
  PATH: "{{ ansible_env.PATH }}:/thingy/bin"
  SOME: value
```

Note: 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.

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.

Note: Ansible 2.0 has deprecated the "ssh" from `ansible_ssh_user`, `ansible_ssh_host`, and `ansible_ssh_port` to become `ansible_user`, `ansible_host`, and `ansible_port`. If you are using a version of Ansible prior to 2.0, you should continue using the older style variables (`ansible_ssh_*`). These shorter variables are ignored, without warning, in older versions of Ansible.

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
bar.example.com
```

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.

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.

How do I configure a jump host to access servers that I have no direct access to?

With Ansible 2, 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*.

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.

How do I handle python pathing not having a Python 2.X in /usr/bin/python on a remote machine?

While you can write ansible modules in any language, most ansible modules are written in Python, and some of these are important core ones.

By default Ansible assumes it can find a /usr/bin/python on your remote system that is a 2.X version of Python, specifically 2.4 or higher.

Setting of an inventory variable 'ansible_python_interpreter' on any host will allow Ansible to auto-replace the interpreter used when executing python modules. Thus, you can point to any python you want on the system if /usr/bin/python on your system does not point to a Python 2.X interpreter.

Some Linux operating systems, such as Arch, may only have Python 3 installed by default. This is not sufficient and you will get syntax errors trying to run modules with Python 3. Python 3 is essentially not the same language as Python 2. Ansible modules currently need to support older Pythons for users that still have Enterprise Linux 5 deployed, so they are not yet ported to run under Python 3.0. This is not a problem though as you can just install Python 2 also on a managed host.

Python 3.0 support will likely be addressed at a later point in time when usage becomes more mainstream.

Do not replace the shebang lines of your python modules. Ansible will do this for you automatically at deploy time.

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.

Where does the configuration file live and what can I configure in it?

See *Configuration file*.

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, or set an environment variable:

```
export ANSIBLE_NOCOWS=1
```

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.

How do I see all the inventory vars defined for my host?

You can see the resulting vars you define in inventory running the following command:

```
ansible -m debug -a "var=hostvars['hostname']" localhost
```

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 %}
```

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.

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 webservers 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 was 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['webservers'][0]]['ansible_eth0']['ipv4']['address'] }}
```

Notice how we're pulling out the hostname of the first machine of the webservers 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[['webservers'][0]] }}

- debug: msg={{ hostvars[headnode].ansible_eth0.ipv4.address }}
```

Notice how we interchanged the bracket syntax for dots – that can be done anywhere.

How do I copy files recursively onto a target host?

The "copy" module has a recursive parameter, though if you want to do something more efficient for a large number of files, take a look at the "synchronize" module instead, which wraps rsync. See the module index for info on both of these modules.

How do I access shell environment variables?

If you just need to access existing variables, use the 'env' lookup plugin. For example, to access the value of the HOME environment variable on management machine:

```
---
# ...
  vars:
     local_home: "{{ lookup('env','HOME') }}"
```

If you need to set environment variables, see the Advanced Playbooks section about environments.

Ansible 1.4 will also make remote environment variables available via facts in the 'ansible_env' variable:

```
{{ ansible_env.SOME_VARIABLE }}
```

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 OS X) 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.encrypt(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 *Vault* to encrypt sensitive data.

Can I get training on Ansible or find commercial support?

Yes! See our services page for information on our services and training offerings. Support is also included with *Ansible Tower*. Email 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.

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*.

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!

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 *Vault*.

In Ansible 1.8 and later, 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.

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 previouslly you could use variables without specifying (like *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, if you need to use a dynamic variable use the hostvars or vars dictionary as appropriate:

```
{{ hostvars[inventory_hostname]['somevar_' + other_var] }}
```

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.

See also:

Ansible Documentation The documentation index

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

CHAPTER

THIRTEEN

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 *playbook documentation*.

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.

Globbing 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 The concept that change commands should only be applied when they need to be applied, and that it is better to describe the desired state of a system than the process of how to get to that state. As an analogy, the path from North Carolina in the United States to California involves driving a very long way West but if I were instead in Anchorage, Alaska, driving a long way west is no longer the right way to get to California. Ansible's Resources like you to say "put me in California" and then decide how to get there. If you were already in California, nothing needs to happen, and it will let you know it didn't need to change anything.

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. These are how such things as `with_items`, a basic looping plugin, are implemented. There are also lookup plugins like `with_file` which load data from a file and ones for querying environment variables, DNS text records, or key value stores. Lookup plugins can also be accessed in templates, e.g., `{{ lookup('file','/path/to/file') }}`.

Loops Generally, Ansible is not a programming language. It prefers to be more declarative, though various constructs like `with_items` allow a particular task to be repeated for multiple items in a list. Certain modules, like yum and apt, are actually optimized for this, and can install all packages given in those lists within a single transaction, dramatically speeding up total time to configuration.

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

See also:

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.

Tasks *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.

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.).

See also:

Frequently Asked Questions Frequently asked questions

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

CHAPTER

FOURTEEN

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 *Playbooks* at the same time to see how this is used in practice.

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']
```

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 | will include the newlines. Using a > will ignore newlines; 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

ignore_newlines: >
            this is really a
            single line of text
            despite appearances
```

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.

Gotchas

While YAML is generally friendly, the following is going to result in a YAML syntax error:

```
foo: somebody said I should put a colon here: so I did
```

You will want to quote any hash values using colons, like so:

```
foo: "somebody said I should put a colon here: so I did"
```

And then the colon will be preserved.

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 }}"
```

The same applies for strings that start or contain any YAML special characters ‘‘ [] { } : > | ‘‘ .

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"
```

See also:

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

CHAPTER

FIFTEEN

PORTING GUIDE

Playbook

- backslash escapes When specifying parameters in jinja2 expressions in YAML dicts, backslashes sometimes needed to be escaped twice. This has been fixed in 2.0.x so that escaping once works. The following example shows how playbooks must be modified:

```
# 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_messsage[:-1] }}"
```

```
- debug:
    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. Previouslly 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: foo.yml
    a: 1
```

Should now be:

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

  Should be:

```
- include: 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 previouslly 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.:

```
with_items: "{{vars['myvar_' + res_of_name]}}"
```

  Or *hostvars[inventory_hostname]['myvar_' + rest_of_name]* if appropriate.

- 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 indistiguishable from a string which makes it difficult to display a proper error.

CHAPTER

SIXTEEN

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

    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

CHAPTER

SEVENTEEN

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. It may mean playing tricks on Ansible.

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. The `fileglob` lookup plugin is pretty simple to understand:

```
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:
    # 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
```

Note: 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

CHAPTER

EIGHTEEN

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: https://github.com/ansible/ansible/blob/devel/docsite/rst/developing_api.rst