How to check version of ansible installed

**#ansible –version**

**======================================================================**

Creating a basic inventory file:

Ansible uses an inventory file (basically, a list of servers) to communicate with your servers. Like a hosts file (at /etc/hosts) that matches IP addresses to domain names, an Ansible inventory file matches servers (IP addresses or domain names) to groups. Inventory files can do a lot more, but for now, we’ll just create a simple file with one server. Create a file at /etc/ansible/hosts (the default location for Ansible’s inventory file), and add one server to it:

**[example]**

**www.example.com**

In the above example the **[example]** is the group name, which would be used to specify in ansible cmd to enable ansible to work on this particular group of machines. [ansi2.example.com](http://www.example.com) is the machine belonging to **[example]** group **,** there can be no of machine in this group, we just need to specify only example in our ansible cmd to avoid the hassle of writing these n names again n again.

If our end machine are listening on different port than standard, then we need to mention them like this in hosts file

[**ansi2.example.com:2222**](http://www.example.com:2222)

we also have the feature of adding multiple groups in one parent group:

**[multi:children]**

**example**

**db**

Here above we have added already existing 2 groups “example and db“ in one parent group, so when we run cmd like

**# ansible-playbook multi**

the cmd would get executed on all hosts of both group.

Similiarly we can define multiple variables to be implemented in one parent group.

**[multi:vars]**

**ansible\_ssh\_user=vagrant**

**ansible\_ssh\_private\_key\_file=~/.vagrant.d/insecure\_private\_key**

We can also define a common variable for all the hosts in the inventory file, like suppose we always use tempuser to login to our staging and production environment, instead of this details in every group or pass it along at command execution. We can do it like this below and this would be used for every execution.

**[all:vars]**

**ansible\_ssh\_user=tempuser**

**ansible\_ssh\_private\_key\_file=~/id\_rsa**

Ansible can be run in 2 modes, 1st is ad-hoc mode in which the cmd is given as a part of activity to be performed, 2nd is thru playbook.

**# ansible example -a "free -m" -u [username]**

The above is the ad-hoc mode example, where

**-a = passing argument to be executed.**

**example = is the group name to perform activity on.**

**-u = to specify username to be used while logging remote hosts**

In this example, we quickly see memory usage (in a human readable format) on all the servers (for now, just one) in the example group, similarly we can keep on changing **“free -m”** with the cmd of our requirement. But imagine this if we have a long list of cmds to be executed this method wont be of any use, there the playbook comes into play.

One important thing to note here is since we just used one single cmd to get free space output, the above cm worked, by default ansible uses “command ” mode for execution of ad-hoc modes. iF we were to change our cmd to “hostname && df -h ” it wont work . we will have to use shell module of ansible.

**# ansible example -m shell -a "hostname && free -m" -u tempuser**

We can further make this above cmds more ansible suitable by using available modules in ansible.

**# ansible web-servers -s -m yum -a "name=ntp state=present"**

Web-servers = group to be actions performed on.

-m yum = yum module for installing packages

-s = sudo privileges.

Name = package name

State = present 🡪 will install the app if not present.

= absent 🡪 will remove app if present

= started 🡪 will start app if stopped

-k If our remote logging user requires sudo password for performing switch, we

should provide flag in above cmd so that ansible ask us.

Here below we have added  **--limit “192.168.1.1”** , what this basically does is, suppose u want to edit or install only on a specific server from group web-servers, --limit will only alter 192.168.1.1 server and install the ntp packages. If in your “**/etc/ansible/hosts**” file you have mentioned FQDN, then in cmd replace **--limit “FQDN”**.

**# ansible web-servers -s -m yum -a "name=ntp state=present" --limit “192.168.1.1”**

**Run-time inventory additions**

Just like static inventory files, it is important to remember that Ansible will parse this data once, and only once, per ansible or ansible-playbook execution. This is a fairly common stumbling point for users of cloud dynamic sources, where frequently a playbook will create a new cloud resource and then attempt to use it as if it were part of the inventory. This will fail, as the resource was not part of the inventory when the playbook launched. All is not lost though!

A special module is provided that allows a playbook to temporarily add inventory to the in-memory inventory object, the **add\_host** module. The add\_host module takes two options, name and groups. The name should be obvious, it defines the hostname that Ansible will use when connecting to this particular system. The groups option is a comma-separated list of groups to add this new system to. Any other option passed to this module will become the host variable data for this host. For example, if we want to add a new system, name it newmastery.example.name, add it to the web group, and instruct Ansible

**- name: add new node into runtime inventory**

**add\_host:**

**name: newmastery.example.name**

**groups: web**

**ansible\_ssh\_host: 192.168.10.30**

This new host will be available to use, by way of the name provided, or by way of the web group, for the rest of the ansible-playbook execution. However, once the execution has completed, this host will not be available unless it has been added to the inventory source itself. Of course, if this were a new cloud resource created, the next ansible or ansible-playbook execution that sourced inventory from that cloud would pick up the new member.

**One thing is important to note here in ansible would gather facts of host even if it is not the target host, i.e but still in mentioned in inverntory file.**

Let's take our previous inventory example and demonstrate the behavior of Ansible with and without a limit. If you recall, we have the special group all that we can use to reference all the hosts within an inventory. Let's assume that our inventory is written out in the current working directory in a file named mastery-hosts, and we will construct a playbook to demonstrate the host on which Ansible is operating

**cat mastery-hosts**

[web]

mastery.example.name ansible\_ssh\_host=192.168.10.25

[dns]

backend.example.name

[frontend:children]

web

[all:vars]

ansible\_ssh\_user=otto

Let's assume that our inventory is written out in the current working directory in a file named mastery-hosts, wand we will construct a playbook to demonstrate the host on which Ansible is operating.

Let's write this playbook out as mastery.yaml:

---

- name: limit example play

hosts: all

gather\_facts: false

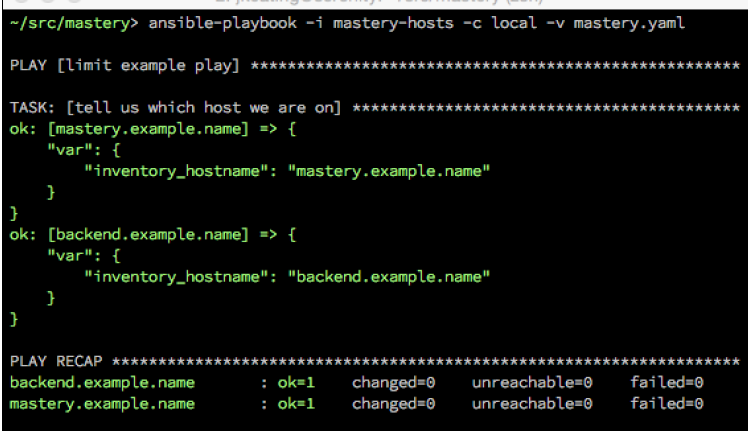
tasks:

**- name: tell us which host we are on**

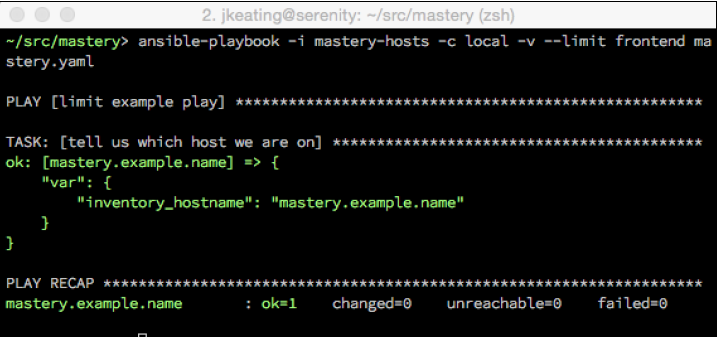
**debug:**

**var: inventory\_hostname**

The debug module is used to print out text, or values of variables. We'll use this module a lot in this book to simulate actual work being done on a host.



As we can see, both hosts **backend.example.name** and **mastery.example.name** were operated on. Let's see what happens if we supply a **limit**, specifically to limit our run to only frontend systems:



We can see that only mastery.example.name was operated on this time. While there are no visual clues that the entire inventory was parsed, if we dive into the Ansible code and examine the inventory object, we will indeed find all the hosts within, and see how the limit is applied every time the object is queried for items.

It is important to remember that regardless of the host's pattern used in a play, or the limit supplied at runtime, Ansible will still parse the entire inventory set during each run. In fact, we can prove this by attempting to access host variable data for a system that would otherwise be masked by our limit. Let's expand our playbook slightly and attempt to access the ansible\_ssh\_port variable from backend.example.name

---

- name: limit example play

hosts: all

gather\_facts: false

tasks:

- name: tell us which host we are on

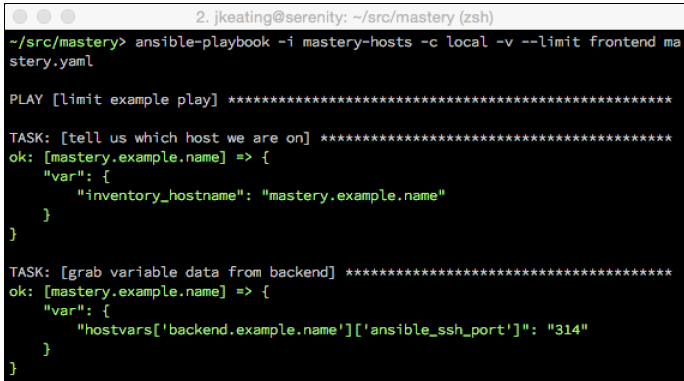
debug:

var: inventory\_hostname

**- name: grab variable data from backend**

**debug:**

**var: hostvars['backend.example.name']['ansible\_ssh\_port']**



We have successfully accessed the host variable data (by way of group variables) for a system that was otherwise limited out. This is a key skill to understand, as it allows for more advanced scenarios, such as directing a task at a host that is otherwise limited out. Delegation can be used to manipulate a load balancer to put a system into maintenance mode while being upgraded without having to include the load balancer system in your limit mask.

**Within a play, there are a few more operations.**

While a playbook is strictly ordered from top to bottom, a play has a more nuanced order of operations. Here is a list of the possible operations and the order in which they will happen:

• Variable loading

• Fact gathering

• The pre\_tasks execution

• Handlers notified from the pre\_tasks execution

• Roles execution

• Tasks execution

• Handlers notified from roles or tasks execution

• The post\_tasks execution

• Handlers notified from post\_tasks execution

A playbook is yaml based file which is passed as an parameter in ansible, and all cmd mentioned in it gets executed one by one.

Below is the eg yaml file.

**---**

**- hosts: all**

**sudo: yes**

**tasks:**

**- name: Ensure NTP (for time synchronization) is installed.**

**yum: name=ntp state=present**

**- name: Ensure NTP is running.**

**service: name=ntpd state=started enabled=yes**

Now we will go thru it one by one.

This first line is a marker showing that the rest of the document will be formatted in YAML.

2nd line tells Ansible to which hosts this playbook applies. all works here, we can also specify particular group in hosts file as well.

3rd Since we need privileged access to install NFS and modify system configuration, this line tells Ansible to use sudo for all the tasks in the playbook.

4th All the tasks after this line will be run on all hosts.

5th & 6th This line command is the equivalent of running yum install ntp, but is much more intelligent; it will check if ntp is installed, and, if not, install it.

7th & 8th This final task both checks and ensures that the ntpd service is started and running, and sets it to start at system boot.

Here we can do one more thing and avoid name tag in above yaml file,

**tasks:**

**- yum: pkg=ntp state=present**

**- service: name=ntpd state=started enabled=yes**

If u will compare this from above, u will notice the hyphen sign is added in yum & service line.

We can also use these module in ad-hoc mode;

**# ansible example -s -m service -a "name=ntpd state=started enabled=yes"**

**# ansible example -s -m yum -a "name=ntp state=present"**

**# ansible example -s -a "iptables -A INPUT -s 192.168.60.0/24 -p tcp -m tcp --dport 3306 -j ACCEPT"**

The “state” variable in service module can have multiple values like **state=stopped**

**Suppose we have pakage which is local on our mahine then we can install it as,**

**- name: Install JDK packages.**

**yum:**

**name: /tmp/jdk-8u111-linux-x64.rpm**

**state: present**

**Copy a file to the servers:**

**# ansible web-servers -m copy -a "src=/etc/hosts dest=/tmp/hosts"**

The above cmd would copy file from the source machine(on which ansible is running) /etc/hosts file to destination machine. **If we add a trailing slash to /etc/** then all file in the etc directory would get copied to destination server.

**Retrieve a file from the servers:**

**# ansible web-servers -s -m fetch -a "src=/etc/hosts dest=/tmp"**

The fetch module works almost exactly the same as the copy module, except in reverse. The major difference is that files will be copied down to the local dest in a directory structure that matches the host from which you copied them. For example, use the following command to grab the hosts file from the servers. Fetch will, by default, put the /etc/hosts file from each server into a folder in the destination with the name of the host i.e Ip address.

So, the web-servers server’s hosts file will end up in **/tmp/192.168.1.1/etc/hosts.**

In order to avoid this, we can make our file fetched directly into /tmp folder by making minute changes, add **flat=yes** **dest=/tmp/** (trailing slash)

**Create a file in remote host:**

**# ansible web-servers -m file -a "dest=/tmp/test/1.txt mode=644 state=touch"**

**Create a directory in remote host:**

**# ansible web-servers -m file -a "dest=/tmp/test mode=644 state=directory"**

**With modification in owner n group permission**

**# ansible web-servers -m file -a "dest=/tmp/pest owner=root group=nobody mode=644 state=directory"**

**Creating symlinks in remote machines:**

**# ansible web-servers -m file -a "src=/etc/hosts dest=/tmp/hosts owner=root group=root state=link"**

**NOTE: For this to work the source n destination file would be on remote hosts only!!!!**

**Delete directories and files**

**# ansible multi -m file -a "dest=/tmp/test state=absent"**

**Getting Information about a file on remote Host:**

If you need to check a file’s permissions, MD5, or owner, use Ansible’s stat module:

**# ansible multi -m stat -a "path=/etc/environment"**

This gives the same information you’d get when running the stat command, but passes back information in JSON.

**How to add normal Linux cmd in ansible playbook.**

The command module takes the command name followed by a list of space-delimited arguments.

The given command will be executed on all selected nodes. It will not be processed through the shell, so variables like $HOME and operations like "<", ">", "|", ";" and "&" will not work (use the shell module if you need these features).

**- name: gettig free space info**

**command: >**

**cp /etc/hosts /tmp/**

The greater-than sign (>) immediately following the command: module directive tells YAML automatically quote the next set of indented lines as one long string, with each line separated by a space”. It helps improve task readability in some cases.

We can pass on various options to it execution,

**chdir :** cd into this directory before running the command

**creates :** A file path is provided with it, so when it already exists, this command will not be run.

**removes:** when the file does not exist, this step will not be run.

**executable:** change the shell used to execute the command. Should be an absolute path to the executable.

Example:

**- name: This command will change the working directory to somedir/ and will only run when /path/to/database**

**File doesn't exist.**

**command: /usr/bin/make\_database.sh arg1 arg2**

**args:**

**chdir: somedir/**

**creates: /path/to/database**

Similiarly we have Shell Module which is capable to handly Wild characters and < > as command module is in capable of handling. All above options of command module is applicable to shell module as well.

**- name: Change the working directory to somedir/ before executing the command.**

**shell: somescript.sh >> somelog.txt**

**args:**

**chdir: somedir/**

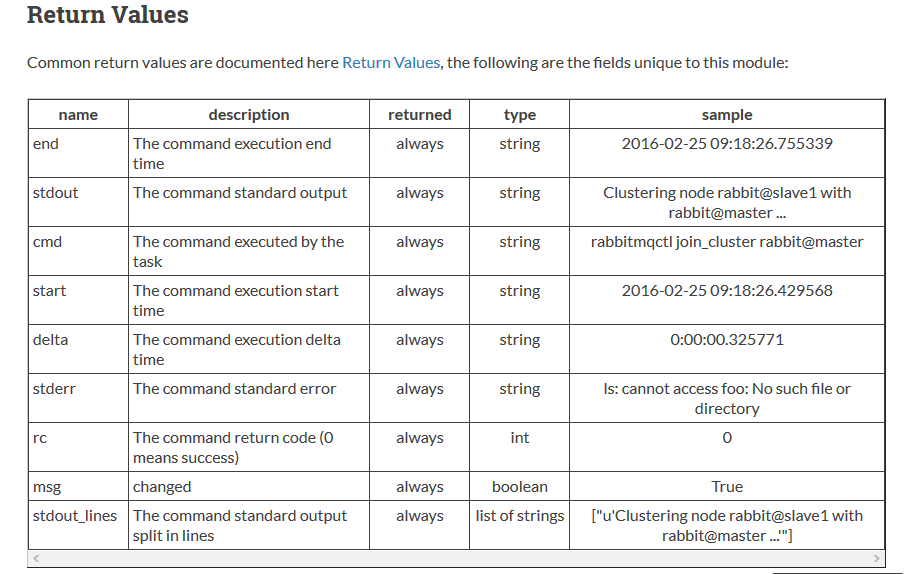
**- name: Run a command that uses non-posix shell-isms (in this example /bin/sh doesn't handle redirection and**

**wildcards together but bash does)**

**shell: cat < /tmp/\*txt**

**args:**

**executable: /bin/bash**



**To gain more usage of a particular module:**

**$ ansible-doc debug | cat -**

**Limiting playbooks to particular hosts and groups**

You can limit a playbook to specific groups or individual hosts by changing the hosts: definition.

The value can be set to all hosts, a group of hosts defined in your inventory, multiple groups of hosts (e.g. webservers, dbservers), individual hosts (e.g. atl.example.com), or a mixture of hosts. You can even do wild card matches, like \*.example.com, to match all subdomains of a top-level domain.

Do remember even for limiting our playbook to run on a particular group or host it need to be present in inventory file.

You can also limit the hosts on which the playbook is run via the ansible-playbook command:

**$ ansible-playbook playbook.yml --limit webservers**

In this case (assuming your inventory file contains a webservers group), even if the playbook is set to hosts: all, or includes hosts in addition to what’s defined in the webservers group, it will only be run on the hosts defined in webservers

You could also limit the playbook to one particular host

**# ansible-playbook playbook.yml --limit xyz.example.com**

We can also run playbook on a bunch of host

**# ansible-playbook playbook.yml --limit “xyz.example.com,dns1.example.com”**

And we also negate a particular host from a group of Hosts: In this scenario plabook wont run on host1.

**# ansible-playbook playbook.yml --limit 'all:!host1'**

We can also limit host with using wildcard characters, here all hosts with ansi1 group with ending ip address x.x.x.4 will be impacted.

**# ansible ansi -s -a "service ntpd restart" --limit "\*.4"**

If you want to see a list of hosts that would be affected by your playbook before you actually run it, use **--list-hosts**:

**$ ansible-playbook playbook.yml --list-hosts**

playbook: playbook.yml

play #1 (all): all TAGS: []

pattern: [u'all']

hosts (1):

**192.168.1.127**

By default, Ansible will run your commands in parallel, using multiple process forks, so the command will complete more quickly.

If you’re managing a few servers, this may not be much quicker than running the command serially, on one server after the other, but even managing 5 - 10 servers, you’ll notice a dramatic speedup if you use Ansible’s parallelism (which is enabled by default).

Run the same command again, but this time, add the argument “**-f 1”** to tell Ansible to use only one fork (basically, to perform the command on each server in sequence):

**$ ansible web-servers -a "hostname" -f 1**

We can increase this value as we please depending upon our network speed n dynamics -f 10, or -f 25. Like this would run ansible cmd parallel on 10 or 25 machines.

**Setting user and sudo options:**

If no user is defined alongside the hosts in a playbook, Ansible assumes you’ll connect as the user defined in your inventory file for a particular host, and then will fall back to your local user account name. You can explicitly define a remote user to use for remote plays using the --remote-user (-u) option:

**$ ansible-playbook playbook.yml --remote-user=johndoe**

In some situations, you will need to pass along your sudo password to the remote server to perform commands via sudo. In these situations, you’ll need use the --ask-sudo-pass **(-K)** option. You can also explicitly force all tasks in a playbook to use sudo with --sudo. Finally, you can define the sudo user for tasks run via sudo (the default is root) with the --sudo-user **(-U)** option.

**$ ansible-playbook playbook.yml --sudo --sudo-user=janedoe --ask-sudo-pass**

We can also run ansible with a diff user and also pass along its private ssh key for cmd executions:

**# ansible -m ping hosts --private-key=~/.ssh/keys/id\_rsa -u centos**

**Gathering Facts About Remote hosts:**

**# ansible web-server -m setup | less**

**Filtering out particular output from gathered facts**

**# ansible web-server -m setup -a "filter=ansible\_distribution\*"**

**Copying SSH key on remote host manually.**

**# ansible web-server -m authorized\_key -a "user=root key='ssh-rsa AAAA...XXX == root@hostname' "**

**Passing variable at ansible adhoc/playbook run time:**

Variables can be pesky, but sometimes are required to be passed in via the CLI. Any variable can be set via the command line. Often the command line is the be all, end all in variable overrides

**# ansible-playbook atmo\_playbook.yml -e "ATMOUSERNAME=atmouser"**

**# ansible-playbook atmo\_playbook.yml --extra-vars "version=1.23.45 other\_variable=foo"**

See below how can we make use of varibale to make our playbooks used in multiple location with diff host and usernames.

**---**

**- hosts: '{{ hosts }}'**

**remote\_user: '{{ user }}'**

**tasks:**

**- ...**

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

**Playbook Debugger::**

The stratergy module allows us to invoke a debugger while running playbook and give us feedback where our playbook failed

**- hosts: test**

**strategy: debug**

**tasks:**

**Busted Cache::**

Sometimes Ansible has a tendency to hold on to variables too long, which causes Ansible to think that a task/operation had already been done or changed when in fact it didn't. A simple fix is to flush the redis cache during a code execution.

This can be done like this:

**# ansible-playbook playbook.yml --flush-cache**

**Check for bad syntax::**

One can check to see if code contains any syntax errors by running the playbook.

Check for bad syntax:

**# ansible-playbook playbook.yml --syntax-check**

**List Task:**

To get a feel of what tasks would be executed upon running playbook ,,just run below cmd

**$ ansible-playbook --list-tasks playbook.yml**

Similiarly we can also list tags.

**$ ansible-playbook --list-tags playbook.yml**

**Step:**

This helps in over viewing each step as it is executed to find out where if something went wrong

**$ ansible-playbook --step oraclejdk-setup.yml**

**Take me right there!!! Aye**

Imagine in our playbook we failed at task 14 which goes by the name “copying to remote host”, so we don’t need to re run entirely playbook from step 1, this is very useful in development stages;

**# ansible-playbook --start-at-task="copying to remote host " playbook.yml**

**https://liquidat.wordpress.com/2016/02/29/useful-options-ansible-cli/**

**Running a playbook on a port other than default “22”…**

We will have to edit the hosts file and add entry over there, it can be can be done in 2 ways:

# vi /etc/ansible/hosts

**[webserver]**

node1.example**:2222**

**or**

node1.example.com **ansible\_port=2222**

**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**

**b-[a:f].example.com**

**Shell Module:**

**http://docs.ansible.com/ansible/shell\_module.html**

**- name: Execute the command in remote shell; stdout goes to the specified file on the remote.**

**shell: somescript.sh >> somelog.txt**

**- name: 1st chng dir to somedir/ and execute somescript.sh if somelog.txt does not exist**

**shell: sh somescript.sh >> somelog.txt**

**args:**

**chdir: somedir/**

**creates: somelog.txt**

**- name: Run a command that uses non-posix shell-isms (in this example /bin/sh doesn't handle redirection and**

**wildcards together but bash does)**

**shell: cat < /tmp/\*.txt**

**args:**

**executable: /bin/bash**

**- name: another way of enclosing cmd to be ran on remote machine specially if reges are used in it.**

**shell: ‘echo “” > /tmp/1.txt ’**

**- name: Run a command using a templated variable (always use quote filter to avoid injection)**

**shell: cat {{ myfile|quote }}**

**---**

**- hosts: ansi2**

**become: yes**

**gather\_facts: no**

**tasks:**

**- name: checking shell power**

**shell: "{{ item }}"**

**with\_items:**

**- "date >> /tmp/1.txt"**

**- "ip r l >> /tmp/1.txt"**

**register: hello**

**- debug: msg="the vairiable is {{ hello }}"**

**Run operations in the background:**

In these situations, you can tell Ansible to run the commands asynchronously, and poll the servers to see when the commands finish. When you’re only managing one server, this is not really helpful, but if you have many servers this can be really beneficial.

Ansible starts the command very quickly on all your servers (especially if you set a higher --forks value), then polls the servers for status until they’re all up to date.

**# ansible node -s -B 1 -a "yum -y install vim"**

192.168.1.127 | FAILED | rc=-1 >>

async task did not complete within the requested time

Like the above cmd would wait for 1 sec for It to complete and if still not done then kills it. In the above cmd we did not put **“-P (value)”**  which is basically polling for status of the completion of above cmd, by default it set value of 10 sec, so in actuality this would for 1 + 10 sec and then kills it off.

**# ansible ansi1 -s -B 1 -P 1 -a "yum -y update"**

Now above cmd would poll for result after 1 sec which is async time or grace period, so it would be 1 + 1 and the cmd is killed off.

**Fire and forget:**

When we set -P to zero, then ansible starts the cmd let it run on its own.

**# ansible ansi1 -s -B 600 -P0 -a "yum -y update"**

ansi1 | SUCCESS | rc=-1 >>

While a background task is running, you can also check on the status elsewhere using Ansible’s **async\_status** module, as long as you have the ansible\_job\_id value to pass in as jid: It was the id that was returned when you ran the original job in the background:

**$ ansible multi -s -m async\_status -a "jid=763350539037"**

How do we utilize this in our playbooks??

**- 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**

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:

**- name: simulate long running op, allow to run for 45 sec, fire and forget**

**command: /bin/sleep 15**

**async: 45**

**poll: 0**

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**

**We also have the option of combining diff tags if they are performing same task like installing multiple applications, copying files and etc.**

**---**

**- hosts: all**

**sudo: yes**

**tasks:**

**- name: Install Apache.**

**yum: name={{ item }} state=present**

**with\_items:**

**- httpd**

**- ntpd**

**- name: Start Services.**

**yum: name={{ item }} state=started**

**with\_items:**

**- httpd**

**- ntpd**

**- name: Copy configuration files.**

**copy:**

**src: "{{ item.src }}"**

**dest: "{{ item.dest }}"**

**owner: root**

**group: root**

**mode: 0644**

**with\_items:**

**- { src: "/path/to/config/httpd.conf", dest: "/etc/httpd/conf/httpd.conf" }**

**- { src: "/path/to/config/httpd-vhosts.conf", dest: "/etc/httpd/conf/httpd-vhosts.conf” }**

Note: Ansible’s copy module works very well for single or small groups of files, and recurses through directories automatically. If you are copying hundreds of files, or deeply-nested directory structures, copy will get bogged down. In these situations, consider using the **synchronize module** if you need to copy a full directory, or unarchive if you want to copy up an archive and have it expanded in place on the server.

**How to add a template on destination server or copy a file on destination server.**

**- name: Add Apache virtualhost for Drupal 8 development.**

**template:**

**src: "templates/drupal.dev.conf.j2"**

**dest: "/etc/apache2/sites-available/{{ domain }}.dev.conf"**

**owner: root**

**group: root**

**mode: 0644**

**notify: restart apache**

**We can use the File module to make a File Absent.(Delete) and also how to add a Symlink.**

**- name: Remove default virtualhost file.**

**file:**

**path: "/etc/apache2/sites-enabled/000-default"**

**state: absent**

**notify: restart apache**

**- name: Symlink Drupal virtualhost to sites-enabled.**

**file:**

**src: "/etc/apache2/sites-available/{{ domain }}.dev.conf"**

**dest: "/etc/apache2/sites-enabled/{{ domain }}.dev.conf"**

**state: link**

**We can also edit a particular line in a file on remote servers with the help of “lineinfile” module,**

**- name: Adding a line in file**

**lineinfile:**

**dest: /tmp/1.txt**

**regexp: "^apc.reg"**

**line: "apc.reg=1"**

**state: present**

The beauty of this module is that it will search a line starting with “apc.reg” if it finds one with the same value as defined in playbook it will keep it untouch, if value is different then it would delete old line and add the new one as per playbook, if the line is not present al together then it would add a new line.

If our line was commented out from in original file, then it would add a new line in it.

Taking this one step further how about we want to add something only after a particular tag in destination file.

**- name: Making LVM work on particular defined partition only.**

**lineinfile:**

**dest: "/etc/lvm/lvm.conf"**

**insertafter: devices {**

**line: 'filter = [ "a|^/dev/sdb$|", "r|.\*/|" ]'**

**state: present**

**Similiarly if we have multiple entries to make using above technique,**

**- name: Adding Entries in "/etc/cinder/cinder.conf"**

**lineinfile:**

**dest: "/etc/cinder/cinder.conf"**

**insertafter: "{{ item.inserts }}"**

**state: present**

**line: "{{ item.lines }}"**

**with\_items:**

**- { inserts: '\[database\]', lines: 'connection = mysql://cinder:cinder@controller/cinder' }**

**- { inserts: '\[DEFAULT\]', lines: 'rpc\_backend = rabbit' }**

**- { inserts: '\[oslo\_messaging\_rabbit\]', lines: 'rabbit\_host = controller' }**

**- { inserts: '\[oslo\_messaging\_rabbit\]', lines: 'rabbit\_userid = openstack' }**

**- { inserts: '\[oslo\_messaging\_rabbit\]', lines: 'rabbit\_password = openstack' }**

**tags: cinder.conf**

**NOTE:** while writing lines tags in above cmd, u might notice it has multiple special characters which can cause issues, so we can use single inverted comma to make it work like shown above. Also remember if a similar exist in destination file it would add a new line keeping the old one as it is.

**Adding a Block of Data on remote server..**

**- name: Setting Up Fio on all machines.**

**blockinfile:**

**dest: /root/fio.ini**

**block: |**

**[random-rw]**

**rw=randrw**

What it does is it copies this data to remote machine to a particular file, if it does not exist there, if it exit exactly then no modification is done, if the data exist but is slightly diffferent then it gets over written by new data , be a little careful while using it.

**https://docs.ansible.com/ansible/blockinfile\_module.html**

**Changing permissions on Folder and its files recursively.**

**- name: Set permissions on files directory.**

**file:**

**path: "{{ drupal\_core\_path }}/sites/default/files"**

**mode: 0777**

**state: directory**

**recurse: yes**

**Note: if the directory does not exists, it will be created.**

**Include a varibale, variables file**

To make our playbook more efficient and readable, let’s begin the playbook (named playbook.yml) by instructing Ansible to load in variables from a separate vars.yml file:

To begin with there 5 types of variable

1] inventory variable (**/etc/ansible/hosts**)

2] role variable

3] play variable (**vars\_files, vars\_prompt**)

4] task variable (**gather\_facts**, **set\_fact add\_host**)

5] exra variable (**--extra-vars**)

**vars\_files:**

**- vars.yml**

Take example of Symlink above where {{ domain }} is used in code, that is basically a variable that is going to be used a lot in our yaml file. Here the variable file tags come into action which help us to de clutter out main yaml file.

We will edit the **“vars.yml”** and add the domain variable values

**domain: “**[**www.example.com**](http://www.example.com)**”**

we can also add variables and its value in playbook.yml file it self,

**vars:**

**- docroot: /var/www/serversforhackers.com/public**

**- username: john**

Tasks can easily be included in a similar way. In the tasks: section of your playbook, you can add include directives like so:

**tasks:**

**- include: included-playbook.yml**

Just like with variable include files, tasks are formatted in a flat list in the included file.

**---**

**- name: Add profile info for user.**

**…………..**

We can even access variables of a particular host by using the “**hostvars**”. For example, to access the foobar variable for examplehost:

**{{ hostvars['examplehost']['foobar'] }}**

Passing variable at cmd prompt:

# ansible-playbook playbook.yml --extra-vars "foo=bar owner=fred"

We can also define variable using the **set\_fact.**

**tasks:**

**- name: set a variable**

**set\_fact:**

**task\_var\_name: "defined variable"**

Asking for variable value at cmd prompt.

**- name: get the operators name**

**pause:**

**prompt: "Please enter your name"**

**register: opname**

Now since there are multiple ways to define a variable what would happen if we define a variable at 2 location , which value would be used.

1st Extra Variable.

2nd Connnection variable like **“ansible\_ssh\_user, ansible\_ssh\_host”** but even this can be overridden by **task variable.**

3rd Most everything else

• Command line switches

• Play variables

• Task variables

• Role variables (not defaults)

**Pre\_tasks and Post\_tasks**

Ansible lets you run tasks before or after the main set of tasks using **pre\_tasks** and **post\_tasks**. In this case, we need to ensure that our apt cache is updated before we run the rest of the playbook, so we have the latest package versions on our server.

**---**

**- hosts: all**

**sudo: yes**

**pre\_tasks:**

**- name: Update machine**

**yum: name=\* state=latest update\_cache=yes**

**post\_tasks:**

**- name: reboot machine**

**shell: reboot**

**tasks:**

**- name: Ensure vim is installed.**

**yum: name=vim state=present**

**Dynamic includes:**

Until Ansible 2.0.0, includes were processed when your playbook run started, so you couldn’t do things like load a particular include when some condition was met. Ansible 2.0.0 evaluates includes during playbook execution, so you can do something like the following:

# Include extra tasks file, only if it's present at runtime.

**- name: Check if extra\_tasks.yml is present.**

**stat: path=extras/extra\_tasks.yml**

**register: extra\_tasks\_file**

**connection: local**

**- include: tasks/extra-tasks.yml**

**when: extra\_tasks\_file.stat.exists**

If the file tasks/extra-tasks.yml is not present, Ansible skips the include. You can even use a with\_items loop (or any other with\_\* loop) with includes. Includes evaluated during playback execution can make your playbooks much more flexible!

**Playbook includes:**

Playbooks can even be included in other playbooks, using the same include syntax in the top level of your playbook. For example, if you have two playbooks—one to set up your webservers (web.yml), and one to set up your database servers (db.yml), you could use the following playbook to run both at the same time:

**- hosts: all**

**remote\_user: root**

**tasks:**

**[...]**

**- include: web.yml**

**- include: db.yml**

One more option is to use it like this with “tags” options

**- include: tasks/drush.yml tags=drush or**

**# ansible-playbook playbook.yml --tags=drush**

So when we run this cmd only file with tags drush would be run .

**Ansible vault:**

Secrets are meant to stay secret. Whether they are login credentials to a cloud service or passwords to database resources, they are secret for a reason. Should they fall into the wrong hands, they can be used to discover trade secrets and private customer data, create infrastructure for nefarious purposes, or worse.

we cover how to keep your secrets safe with Ansible.

• Encrypting data at rest

• Protecting secrets while operating

The facility that provides this service is Vault, which allows for encrypting text files so that they are stored "at rest" in encrypted format. Without the key or a significant amount of computing power, the data is indecipherable.

The key lessons to learn while dealing with encrypting data at rest are:

• Valid encryption targets

• Creating new encrypted files

• Encrypting existing unencrypted files

• Editing encrypted files

• Changing the encryption password on files

• Decrypting encrypted files

• Running the ansible-playbook command to reference encrypted files

**Things Vault can encrypt**

This is essentially any YAML (or JSON) file that Ansible uses during its operation.

This can include:

• group\_vars/ files

• host\_vars/ files

• include\_vars targets

• vars\_files targets

• --extra-vars targets

• role variables

• Role defaults

• Task files

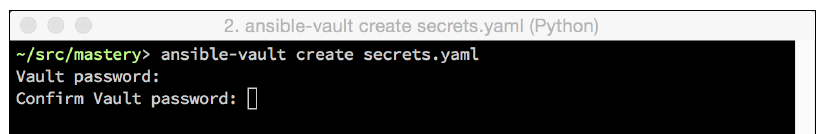
• Handler files

If the file can be expressed in YAML and read by Ansible, it is a valid file to encrypt with Vault.

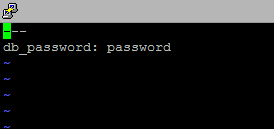
**Creating new encrypted files**

To create a new file, you'll need to know two things ahead of time. The first is the password Vault should use to encrypt the file, and the second is the file name. Once provided with this information, ansible-vault will launch a text editor, whichever editor is defined in the environment variable EDITOR. Once you save the file and exit the editor, ansible-vault will use the supplied password as a key to encrypt the file with the AES256 cipher.

Let's walk through a few examples of creating encrypted files. First, we'll create one and be prompted for a password; then we will provide a password file; and finally we'll create an executable to deliver the password .



Here as shown above we are creating the secrets.yaml file. The file does not exist now, it is being created and asking us for a password to protect the file. Once that is done it will open the file. And we can put in our secret content.



Now when ever we open this file, we would need to provide password. Otherwise it wont open. Plus even if we cat the file, it would be all encrypted.

Now we can even create a password file to be used with ansible-vault. Then we can reference this file while calling ansible-vault to create/edit another encrypted file:

As this would help us share our ansible code with other with out worrying about that they would be able to see our db password and this would also help us to automate execution of our ansible tasks.

**# echo "password" > password\_file**

**# ansible-vault create --vault-password-file password\_file more\_secrets.yml**

Editing a existing file associated with a password\_file.

**# ansible-vault edit --vault-password-file=./password\_file more\_secrets.yml**

**Similiarly we can encrypt existing file :**

As with create, encrypt expects a password (or password file) and the path to a file. In this case, however, the file must already exist. Let's demonstrate this by encrypting an existing file, a\_vars\_file.yaml

**# ansible-vault encrypt a\_vars\_file.yaml**

New Vault password:

Confirm New Vault password:

**How to change password of a existing encryted file.**

**# ansible-vault rekey secrets.yml**

Vault password:

New Vault password:

Confirm New Vault password:

Rekey successful

**How to change the password file of a existing encrypted file.**

**# ansible-vault rekey --vault-password-file=./password\_file --new-vault-password-file=./password2 chaman.yml**

**How to decrypt a Encrypted file.**

If, at some point, the need to encrypt data files goes away, ansible-vault provides a subroutine that can be used to remove encryption for one or more encrypted files.

**# ansible-vault decrypt --vault-password-file=./password\_file chaman.yml**

**How do we decrypt/Execute a file at ansible playbook cmd line.**

**# ansible-playbook -i mastery-host --vault-password-file=./password\_file chaman.yml**

**# ansible-playbook -i mastery-host --ask-vault-pass chaman.yml**

**How to make ansible avoid logging or showing secret data on screen on in logs!**

As shown above even after using vault there might come a scenario where ansible would dump or show sensitive information in clear text on remote host; in order to avoid this we will make use of no\_log module which would basically suppress any form of display on that particular task executiion.

**tasks:**

**- name: print the variable**

**debug:**

**var: something**

**no\_log: true**

**REGISTER & WHEN:**

The “**REGISTER**” directive basically stores the output of a sub-play which can be used later on.

The **“WHEN”** directive processes the execution of a sub play depending upon the outcome to previous task using register values, or looking on variable boolean values(true, false), or by looking at the anisble\_facts.

**Register:**

In Ansible, any play can ‘register’ a variable, and once registered, that variable will be available to all subsequent tasks. Registered variables work just like normal variables or host facts.

Many times, you may need the output (stdout or stderr) of a shell command, and you can get that in a variable using the following syntax:

**- shell: my\_command\_here**

**register: my\_command\_result**

Later, you can access **stdout** (as a string) with **my\_command\_result.stdout**, and **stderr** with **my\_- command\_result.stderr.**

**When:**

The "Add Nginx Repository" Task registers "ppastable". The register directive basically stores the output of Nginx repository addition task. Then we use that to inform the Install Nginx Task to only run when the registered "ppastable" Task is successful. This allows us to conditionally stop Ansible from running a Task.

**- name: Add Nginx Repository**

**apt\_repository: repo='ppa:nginx/stable' state=present**

**register: ppastable**

**- name: Install Nginx**

**apt: pkg=nginx state=installed update\_cache=true**

**when: ppastable | success**

**register: nginxinstalled**

**notify:**

**- Start Nginx**

One of the most helpful extra keys you can add to a play is a 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.

Let’s take a look at a simple use of

**- yum: name=mysql-server state=present**

**when: is\_db\_server**

The above statement assumes you’ve defined the **is\_db\_server** variable as a boolean **(true or false**) earlier, and will run the play if the value is true, or skip the play when the value is false.

If you only define the is\_db\_server variable on database servers (meaning there are times when the variable may not be defined at all), you could run tasks conditionally like so:

**- yum: name=mysql-server state=present**

**when: (is\_db\_server is defined) and is\_db\_server**

“when is even more powerful if used in conjunction with variables registered by previous tasks”. As an example, we want to check the status of a running application, and run a play only when that application reports it is ‘ready’ in its output:

**- command: my-app --status**

**register: myapp\_result**

**- command: do-something-to-my-app**

**when: "'ready' in myapp\_result.stdout"**

some examples of uses of when in real-world playbooks:

# From our Node.js playbook register a command's output, then see if the path to our app is in the output. Start the app if it's not present.

**- command: forever list**

**register: forever\_list**

**- command: forever start /path/to/app/app.js**

**when: "forever\_list.stdout.find('/path/to/app/app.js') == -1"**

# Copy a file to the remote server if the hosts file doesn't exist.

**- stat: path=/etc/hosts**

**register: hosts\_file**

**- copy: src=path/to/local/file dest=/path/to/remote/file**

**when: hosts\_file.stat.exists == false**

# Run 'ping-hosts.sh' script if 'ping\_hosts' variable is true

**---**

**- hosts: all**

**vars:**

**ping\_hosts: true**

**tasks:**

**- command: /usr/local/bin/ping-hosts.sh**

**when: ping\_hosts**

# Run 'git-cleanup.sh' script if variable is\_app\_server is set to true and branch we're interested in is missing fromgit's

list of branches in our project.

**- command: chdir=/path/to/project git branch**

**register: git\_branches**

**- command: /path/to/project/scripts/git-cleanup.sh**

**when: "(is\_app\_server == true) and ('interesting-branch' not in git\_branches.stdout)"**

# Downgrade PHP version if the current version contains '7.0'.

**- shell: php --version**

**register: php\_version**

**- shell: yum -y downgrade php\***

**when: "'7.0' in php\_version.stdout"**

uptill now this were our defined variable which resulted in ansible to take action. We can also use ansible own dynamic variable library to get variable info depending upon type of Guest machine and take action.

The below play would call for variable **“ansible\_os\_family”** of ansible library and would take appropriate action, we don’t need to provide any input.

**- name: "shut down Debian flavored systems"**

**command: /sbin/shutdown -t now**

**when: ansible\_os\_family == "Debian"**

Similiarly we can group conditions. So here if distribution is centos with major version 6 or the distri is debian with major ver 7, then only machine would be shutdown.

**- 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")**

The above same cmd can be written like this if we have multiple scenarios that need to be true to enable further executions.

**- name: "shut down CentOS 6 systems"**

**command: /sbin/shutdown -t now**

**when:**

**- ansible\_distribution == "CentOS"**

**- ansible\_distribution\_major\_version == "6"**

Suppose we want to ignore the error of one statement and then decide to do something conditionally based on success or failure:

- 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**

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**

so our below play would see is epic variable defined with a true value then only execution would take place.

Then a conditional execution might look like:

**tasks:**

**- shell: echo "This certainly is epic!"**

**when: epic**

Similiarly we can avoid execution by saying when variable is not defined.

**- 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:

**- 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**

**Changed\_when and failed\_when**:

Just like when, you can use **changed\_when and failed\_when** to influence Ansible’s reporting of when a certain task results in changes or failures.

It is difficult for Ansible to determine if a given command results in changes, so if you use the **command or shell module** without also using changed\_when, Ansible will always report a change. Most Ansible modules report whether they resulted in changes correctly, but you can also override this behavior by invoking changed\_when yourself.

Suppose when u run “yum update” on a remote machine, this give u a output “Changed=1” which is not right as even there was no update on remote host,, so in order to over come this false positive we can use Changed\_when modue..

**- name: Updating machine**

**shell: "yum update"**

**register: update**

**changed\_when: "'No Packages marked for Update' not in update.stdout"**

You can see we used register to store the results of the command, then we checked whether a certain string was in the registered variable’s stdout. Only when Update doesn’t do anything will it print “No Packages marked for Update“, so we use that string to tell Ansible if the task resulted in a change. this would give u a output as Changed=0 as there was nothing to update on remote host, which would be correct.

Many command-line utilities print results to stderr instead of stdout, so **failed\_when** can be used to tell Ansible when a task has actually failed and is not just reporting its results in the wrong way.

**register: import**

**failed\_when: "import.stderr and 'already exists' not in import.stderr"**

In this case, we only want Ansible to report a failure when the command returns an error, and that error doesn’t contain ‘already exists’.

**Ignore Errors:**

Sometimes there are commands that should be run always, and they often report errors. Or there are scripts you might run that output errors left and right, and the errors don’t actually indicate a problem, but they’re just annoying (and they cause your playbooks to stop executing).

For these situations, you can add **ignore\_errors: true** to the task, and Ansible will remain blissfully unaware of any problems running a particular task. Be careful using this, though; it’s usually best if you can find a way to work with and around the errors generated by tasks so playbooks do fail if there are actual problems.

**ignore\_errors: yes**

**Creates Module :**

Ansible has the creates option in the command module. Give it a filename (directories will not work) and if it already exists Ansible will skip the action.

If a command creates a file after it is first run, or removes a file after it is first run, you can use the creates or removes argument with command or shell. Then it won’t run a second time

**- name: trivially create a file**

**shell: echo "hello" > /tmp/hello**

**args:**

**creates: /tmp/hello**

Another Example:

We need to expand the Apache Solr archive, then copy it into place. For both of these steps, use the built-in tar and cp utilities (with the appropriate options) to do the work. Setting creates tells Ansible to skip these steps in subsequent runs, since the Solr war file will already be in place.

**- name: Copy Solr into place.**

**command: >**

cp -r {{ download\_dir }}/solr-{{ solr\_version }} {{ solr\_dir }}

**args:**

creates={{ solr\_dir }}/dist/solr-{{ solr\_version }}.war

More Elaborate Example by using **with\_items** Tag:

# Use shell so commands are passed in correctly. So upon execution of this when the file /tmp/pappu.txt exist on destination the sub play wont get executed.

**- name: copy file to node2**

**shell:**

cp -r {{ item.src }} {{ item.dest }}

**with\_items:**

- { src: /etc/hosts, dest: /tmp/pappu.txt }

**args:**

**creates: /tmp/pappu.txt**

**Roles:**

Roles usually have a directory structure like this:

rolename

- files

- handlers

- meta

- templates

- tasks

- vars

Within each directory, Ansible will search for and read any Yaml file called main.yml automatically and execute it.

The most appropriate way to create a role is by using this command, where adminpod is the role name itself.

**# ansible-galaxy init adminpod**

**/root/deli-adminpod-ansible/roles/adminpod\_worker**

**[root@ansible adminpod\_worker]# ls -l**

**drwxr-xr-x. 2 root root 4096 Feb 6 09:25 defaults**

**drwxr-xr-x. 2 root root 4096 May 16 10:01 files**

**drwxr-xr-x. 2 root root 4096 Feb 6 09:25 handlers**

**drwxr-xr-x. 2 root root 4096 May 16 10:02 tasks**

**drwxr-xr-x. 2 root root 4096 May 17 08:56 templates**

**drwxr-xr-x. 2 root root 4096 Feb 6 09:25 vars**

Every directory within your role directory has its unique functioning,

**Role scaffolding :**

**The bare Minimum.**

Instead of requiring you to explicitly include certain files and playbooks in a role, Ansible automatically includes any main.yml files inside specific directories that make up the role. There are only two directories required to make a working Ansible role**:**

**role\_name/**

**meta/**

**tasks/**

If you create a directory structure like the one shown above, with a main.yml file in each directory, Ansible will run all the tasks defined in tasks/main.yml if you call the role from your playbook using the following syntax:

**#vi /etc/ansible/playbook.yml**

**---**

**- hosts: all**

**roles:**

**- pappu** 🡪 role name

**#ansible-playbook playbook.yml**

So all cmd in pappu role (tasks/main.yml), would get executed.

Going a step further, suppose our playbook look like this, in which a vairable is sought out “node\_npm\_modules” we can define its value in “defaults/main.yml”

# vi tasks/main.yml

- name: Install npm modules required by our app.

npm: name={{ item }} global=yes state=latest

with\_items: node\_npm\_modules

#cat defaults/main.yml

node\_npm\_modules:

- forever

- async

- request

So now when u run playbook, with role pappu, it will execute role “pappu ” on appservers.

# cat playbook.yml

---

- hosts: appservers

roles:

- pappu

1st: tasks/main.yml will get executed,

2nd: In that it will refrence “node\_npm\_modules” that will lead it to “defaults/main.yml” and so on.

# ansible-playbook playbook.yml

We can define multiple roles in out playboook and they all can be executed on “appservers”

---

- hosts: appservers

roles:

- pappu

- chappu

Each one of the roles lives in its own isolated world, and can be shared with other servers and groups

of servers in your infrastructure

it would also work if we created a role manually with just 2 folders in it, like meta and task,,the main.yml file in this folder will be automatically executed. When this role is called upon.

Generally there is default playbook.yml file which is usually run to call the specified role When running a role’s tasks, Ansible picks up variables defined in a role’s **vars/main.yml** file and **defaults/main.yml.**

Another fine example of roles,,

# cat playbooks/update.yml

---

- hosts: all

roles:

- { role: apt-update, when: "ansible\_os\_family == 'Debian'" }

- { role: yum-update, when: "ansible\_os\_family == 'RedHat'" }

**Meta:**

The main.yml file within the meta directory contains Role meta data, including dependencies.

If this Role depended on another Role, we could define that here. For example, I have the Nginx Role depend on the SSL Role, which installs SSL certificates.

**---**

**dependencies:**

**- { role: ssl }**

If I called the "nginx" Role, it would attempt to first run the "ssl" Role. And then run it own role.

Otherwise we can omit this file, or define the Role as having no dependencies:

**---**

**dependencies: [ ]** - - > no space between square brackets

**Files:**

First, within the files directory, we can add files that we'll want copied into our remote servers. So in our “files dir” we can have file like this..

**[root@ansible adminpod\_worker]# cd files/**

**[root@ansible files]# ll**

**-rw-r--r--. 1 root root 199 Feb 6 ssh\_config\_header**

In files dir, we won't find a main.yml.

**Handlers:**

Handlers are special kinds of tasks you run at the end of a group of tasks by adding the notify option to any of the tasks in that group. The handler will only be called if one of the tasks notifying the handler makes a change to the server (and doesn’t fail), and it will only be notified at the end of the group of tasks.

**handlers:**

**- name: restart apache**

**service: name=apache2 state=restarted**

**- name: stop ntpd**

**service: name=ntpd state=stopped**

To call this handler, add the option **notify: restart apache** after defining the rest of a play. We’ve defined this handler so we can restart the apache2 service after a configuration change, which will be explained below.

Note: By default, Ansible will stop all playbook execution when a task fails, and won’t even notify any handlers that may need to be triggered. In some cases, this leads to unintended side effects. If you want to make sure handlers always run after a task uses notify to call the handler, even in case of playbook failure, add **--force-handlers** to your **ansible-playbook command.** Or you can use the meta module to do so (e.g. - meta: flush\_handlers).

Handlers can be included just like tasks, within a playbook’s handlers section. For example:

**handlers:**

**- include: included-handlers.yml**

This can be helpful in limiting the noise in your main playbook, since handlers are usually used for things like restarting services or loading a configuration, and can distract from the playbook’s primary purpose.

Inside of the handlers directory, we can put all of our Handlers that were once within the nginx.yml Playbook. Inside of handlers/main.yml:

**---**

**- name: Start Nginx**

**service: name=nginx state=started**

**- name: Reload Nginx**

**service: name=nginx state=reloaded**

Once these are in place, we can reference them from other files freely. We can also allow one set of code allow to call 2 diff handlers ,,

**- name: Rebuild application configuration.**

**command: /opt/app/rebuild.sh**

**notify:**

**- restart apache**

**- restart memcached**

And also one handler can call another handler as well like below,

**handlers:**

**- name: restart apache**

**service: name=apache2 state=restarted**

**notify: restart memcached**

**- name: restart memcached**

**service: name=memcached state=restarted**

So when `**restart memcached**` is called it will run itseld then call `**restart memcached**` to run as well.

when running your playbook. Handlers won’t run on any hosts that became unreachable during the playbook’s run.

**Template:**

Template files can contain template variables, based on Python's Jinja2 template engine. Files in here should end in .j2, but can otherwise have any name. Similar to files, we won't find a main.yml file within the templates directory.

Our template file could also contain variables which we will define later in the vars/main.yml file.

**Variables:**

Variables in Ansible work just like variables in most other systems. Variables always begin with a letter ([A-Za-z]), and can include any number of underscores (\_) or numbers ([0-9]).

Valid variable names include foo, foo\_bar, foo\_bar\_5, and fooBar, though the standard is to use all lowercase letters, and typically avoid numbers in variable names (no camelCase or UpperCamelCase). Invalid variable names include \_foo, foo-bar, 5\_foo\_bar, foo.bar and foo bar.

In an inventory file, a variable’s value is assigned using an equals sign, like so:

foo=bar

In a playbook or variables include file, a variable’s value is assigned using a colon, like so:

foo: bar

We can also pass variable to ansible in cmd line while running playbook ,

**# ansible-playbook example.yml --extra-vars "foo=bar"**

**# ansible-playbook example.yml -e "foo=bar"**

Variables may also be included in a separate file, using the vars\_files section:

**---**

**- hosts: example**

**vars\_files:**

**- vars.yml**

**tasks:**

**- debug: msg="Variable 'foo' is set to {{ foo }}"**

--- # Variables file 'vars.yml' in the same folder as the playbook.

**foo: bar**

There is also a vars directory containing a main.yml file which simply lists variables we'll use. This provides a convenient place for us to change configuration-wide settings.

Here's what the vars/main.yml file might look like:

**---**

**domain: serversforhackers.com**

**ssl\_key: /etc/ssl/sfh/sfh.key**

**ssl\_crt: /etc/ssl/sfh/sfh.crt**

These are three variables which we can use elsewhere in this Role. We saw them used in the template above, but we'll see them in our defined Tasks as well.

There is more option of setting ENVIRONMENT variable on remote hosts ,,

**- name: Add an environment variable to the remote user's shell.**

**lineinfile:**

**dest=~/.bash\_profile**

**regexp=^ENV\_VAR=**

**line=ENV\_VAR=value**

**- name: Get the value of the environment variable we just added.**

**shell: 'source ~/.bash\_profile && echo $ENV\_VAR'**

**register: foo**

**- name: Print the value of the environment variable.**

**debug: msg="The variable is {{ foo.stdout }}"**

**- name: print to stdout**

**command: echo "hello"**

**register: hello**

**- debug: msg="{{ hello.stdout }}"**

**- debug: msg="{{ hello.stderr }}"**

**- name: Print mosh version**

**debug: "msg=Mosh Version: '{{ mosh\_version.stdout }}'"**

There are many different places you can store environment variables, including .bashrc, .profile, and .bash\_profile in a user’s home folder. In our case, since we want the environment variable to be available to Ansible, which runs a pseudo-TTY shell session, in which case .bash\_profile is used to configure the environment.

If your application requires many environment variables (as is the case in many Java applications), you might consider using copy or template with a local file instead of using lineinfile with a large list of items.

You can also set the environment for just one play, using the environment option for that play. As an

example, let’s say you need to set an http proxy for a certain file download. This can be done with:

**- name: Download a file, using example-proxy as a proxy.**

**get\_url:**

**url=http://www.example.com/file.tar.gz**

**dest=~/Downloads/**

**environment:**

**http\_proxy: http://example-proxy:80/**

That could be rather cumbersome, though, especially if you have many tasks that require a proxy or some other environment variable. In this case, you can pass an environment in via a variable in your playbook’s vars section (or via an included variables file), like so:

**vars:**

**var\_proxy:**

**http\_proxy: http://example-proxy:80/**

**https\_proxy: https://example-proxy:443/**

**[etc...]**

**tasks:**

**- name: Download a file, using example-proxy as a proxy.**

**get\_url: url=http://www.example.com/file.tar.gz dest=~/Downloads/**

**environment: var\_proxy**

Now we set the proxy just for one task , i.e for downloading the tar.gz file, to set it system wide , for entire time of the playbook ,,

**# In the 'vars' section of the playbook (set to 'absent' to disable proxy):**

**proxy\_state: present**

**# In the 'tasks' section of the playbook:**

**- name: Configure the proxy.**

**lineinfile:**

**dest: /etc/environment**

**regexp: "{{ item.regexp }}"**

**line: "{{ item.line }}"**

**state: "{{ proxy\_state }}"**

**with\_items:**

**- { regexp: "^http\_proxy=", line: "http\_proxy=http://example-proxy:80/" }**

**- { regexp: "^https\_proxy=", line: "https\_proxy=https://example-proxy:443/" }**

**- { regexp: "^ftp\_proxy=", line: "ftp\_proxy=http://example-proxy:80/" }**

Doing it this way allows me to configure whether the proxy is enabled per-server (using the proxy\_state variable), and with one play, set the http, https, and ftp proxies. You can use a similar kind of play for any other types of environment variables you need to set system-wide.

You can test remote environment variables using the ansible command: **ansible test -m shell -a 'echo $TEST'**. When doing so, be careful with your use of quotes and escaping you might end up using double quotes where you meant to use single quotes, or vice-versa, and end up printing a local environment variable instead of one from the remote server!

Taking Input from user and setting it as Variable and using it in further tasks:

**- hosts: cinder**

**vars\_prompt:**

**- name: "partition"**

**prompt: “Provide drive infotmation”**

**private: no**

**tasks:**

**- name: listing directory.**

**shell: ls -l {{ partition }}**

When our playbook is run, It will ask us to provide input for **"partition".** And would substitute it value later on.

Here “prompt” would be the display msg, and our key value pair would be partition: {{ value provided by us}}.

Private: is basically if we want our variable pass on would be visibl eto naked eyes or do we want it to show as \*\*\*\*\* like a password.

Similiarly we can set our playbook to ask multiple vairables from us and utilize it in execution.

- name: "HDD\_info"

prompt: Provide HDD infotmation

private: no

- name: "dir path"

prompt: pass in dir path

private: yes

Variables may also be added via Ansible inventory files, either inline with a host definition, or after a group:

[**washington]**

**app1.example.com proxy\_state=present**

**app2.example.com proxy\_state=absent**

For example, to apply a set of variables to the host app1.example.com, create a blank file named **app1.example.com** at the location **/etc/ansible/host\_vars/app1.example.com**, and add variables as you would in an included vars\_files YAML file:

**---**

**foo: bar**

**baz: qux**

To apply a set of variables to the entire washington group, create a similar file in the location **/etc/ansible/group\_vars/washington** (substitute washington for whatever group name’s variables you’re defining).

**--**

# file: group\_vars/washington

**ntp: ntp-atlanta.example.com**

**backup: backup-atlanta.example.com**

So now when we run a ansible playbook with **“ -l Washington** ” the playbook can pull variable like **ntp, backup** from the above mentioned group\_vars file.

Similiarly we can set a host based variable file, so that whenever a playbook is run on a individual host, variable value would be pull from **“host\_vars/ansi1.example.com”** file.

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**

Registered Variables:

Accessing varibales:

# In your playbook.

tasks:

- debug: var=ansible\_eth0

There is one more extremely useful way of getting variable dynamically through the use of **“include\_vars”** what it does is it takes input from ansible setup module which gather facts and inherit variable values from there making it dynamic it in nature. For eg, suppose I have 2 different type of os like redhat and debian, I want ansible to check the type of os end mahine is and accordingly run a specific playbook.

**[root@ns0 ansible]# pwd**

**/etc/ansible**

**[root@ns0 ansible]#**

**[root@ns0 ansible]# ll | grep -E 'setup\*|RedHat|include'.yml**

-rw-r--r-- 1 root root 210 Jun 10 05:02 include.yml

-rw-r--r-- 1 root root 20 Jun 10 05:01 RedHat.yml

-rw-r--r-- 1 root root 62 Jun 10 04:47 setup-Redhat.yml

**[root@ns0 ansible]# cat include.yml**

**---**

**- hosts: all**

**tasks:**

**- name: Getting os info**

**include\_vars: "{{ ansible\_os\_family }}.yml"**

**- include: setup-Redhat.yml**

**when: ansible\_os\_family == 'RedHat'**

**- debug: msg="{{ package }}"**

**[root@ns0 ansible]#**

**[root@ns0 ansible]#**

**[root@ns0 ansible]# cat RedHat.yml**

**---**

**package: httpd**

**[root@ns0 ansible]#**

**[root@ns0 ansible]#**

**[root@ns0 ansible]# cat setup-Redhat.yml**

**---**

**- name: htttp install**

**yum: name={{ package }} state=present**

Also we can extract variable value by using debug module like below where we are asking value for ssh port of a host.

**- name: grab variable data from backend**

**debug:**

**var: hostvars['backend.example.name']['ansible\_ssh\_port']**

===============================================================================================

**Ansible-Vault:**

If you use Ansible to fully automate the provisioning and configuration of your servers, chances are you will need to use passwords or other sensitive data for some tasks, whether it’s setting a default admin password, synchronizing a private key, or authenticating to a remote service.

Some projects store such data in a normal variables file, in version control with the rest of the playbook, but in this case, the data is easily accessed by anyone with a copy of the project. It’s better to treat passwords and sensitive data specially, and there are two primary ways to do this:

For most projects, Ansible’s built-in Vault is adequate,

1] You take any YAML file you would normally have in your playbook and store it in the vault.

2] Ansible encrypts the vault (‘closes the door’), using a key (a password you set).

3] You store the key (your vault’s password) separately from the playbook in a location only you control or can

access.

4] You use the key to let Ansible decrypt the encrypted vault whenever you run your playbook.

As any \*.yml file can be our ansible vault file. Like for eg, our var.yml file containing our MYSQL Db password, we wont want it to be viewable by everyone, so we can encrypt it,

**$ ansible-vault encrypt var.yml**

When running this you would be asked for a secure password,, after applying this when we open the file we would see all encrypted content in it, i.e not readable.

Now when we run the playbook we will have to provide password with out which our playbook wont run.

**$ ansible-playbook test.yml ----------- > this wont work…**

**$ ansible-playbook test.yml --ask-vault-pass**

Vault password:

We can afterwards edit the encrypted file, change its password, view it or decrypt it permanently.

**# ansible-vault edit**

We can also provide password to ansible via file,, we will have to treat this file with utmost care, i.e chmod 600 vault\_pass.txt,,, put it in secure location on ur machine where no one can access it and then pass it along while running cmd like this below,

**# ansible-playbook test.yml --vault-password-file ~/.ansible/vault\_pass.txt**

**http://docs.ansible.com/ansible/latest/playbooks\_conditionals.html**

**Delegation:**

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**

If we look at the above code, you will see 1st our local machine ran a local script to bring the inventory\_hostname server out of load balancing , installed a package on “ inventory\_hostname” and again add it to the load balancing pool.

We can also replace “delagate\_to” with “local\_action” .

**- name: take out of load balancer pool**

**local\_action: command /usr/bin/take\_out\_of\_pool {{ inventory\_hostname }}**

**Important:**

Look at tag Serial : 5,, its an important feature, which tells ansible to how much machine to work at a time,, like above cmd would start ansible at 5 machine groups at a time , after fininshing on 5 then move on to next 5.

We can also increase value as we move ahead like, like 1st it would work on 1 machine, then increase it to simultaeniously to 5, after completing on 5 then increase it to 10,, and hence forth 10 on groups,,

**Serial:**

**- 1**

**- 5**

**- 10**

It can also accept in Percentage…

**- name: test play**

**hosts: webservers**

**serial:**

**- "10%"**

**- "20%"**

**- "100%"**

**Max Failure Percentage:**

In the below example, if more than 3 of the 10 servers in the group were to fail, the rest of the play would be aborted.

**- hosts: webservers**

**max\_fail\_percentage: 30**

**serial: 10**

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.

**Run Once:**

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:

**- 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”.

**Local Playbooks:**

It may be useful to use a playbook locally, rather than by connecting over SSH. 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**

We can also add connection in our playbook like

**- hosts: 127.0.0.1**

**connection: local**

**Interrupt execution on any error:**

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

1] disable traffic on load balancers (must be turned off simultaneously)

2] gracefully stop the service

3] upgrade software (this step includes tests and starting the service)

4] enable traffic on the load balancers (which should be turned on simultaneously)

The service can’t be stopped with “alive” load balancers; they must be disabled first. Because of this, the second stage can’t be played if any server failed in the first stage.

**---**

**- hosts: load\_balancers\_dc\_a**

**any\_errors\_fatal: True**

**tasks:**

**- name: 'shutting down datacenter [ A ]'**

**command: /usr/bin/disable-dc**

**- hosts: frontends\_dc\_a**

**tasks:**

**- name: 'stopping service'**

**command: /usr/bin/stop-software**

**- name: 'updating software'**

**command: /usr/bin/upgrade-software**

**- hosts: load\_balancers\_dc\_a**

**tasks:**

**- name: 'Starting datacenter [ A ]'**

**command: /usr/bin/enable-dc**

In this example Ansible will start the software upgrade on the front ends only if all of the load balancers are successfully disabled.

**Pausing playbook execution with wait\_for:**

**- name: Wait for webserver to start.**

**local\_action:**

**module: wait\_for**

**host: "{{ inventory\_hostname }}"**

**port: "{{ webserver\_port }}"**

**delay: 10**

**timeout: 300**

**state: started**

The above task waits until webserver\_port is open on inventory\_hostname, as checked from the host running the Ansible playbook, with a 5-minute timeout (and 10 seconds before the first check, and between checks).

**Prompts:**

Under rare circumstances, you may require the user to enter the value of a variable that will be used in the playbook. If the playbook requires a user’s personal login information, or if you prompt for a version i.e it can be dynamic in nature so in this scenario it would be very helpful use vars\_prompt.

As a simple example, you can request a user to enter a username and password that could be used to login to a network share

**---**

**- hosts: all**

**vars\_prompt:**

**- name: share\_user**

**prompt: "What is your network username?"**

**- name: share\_pass**

**prompt: "What is your network password?"**

**private: yes**

**encrypt: "sha512\_crypt"**

**confirm: yes**

**salt\_size: 7**

There are a few special options you can add to prompts:

• private: If set to yes, the user’s input will be hidden on the command line.

• default: You can set a default value for the prompt, to save time for the end user.

• confirm: makes u enter password twice.

• encrypt / salt\_size: These values can be set for passwords so you can verify the entry , and encrypt it using a salt with the specified size and crypt scheme.

**Tags:**

If you have a large playbook it may become useful to be able to run a specific part of the configuration without running the whole playbook.

Both plays and tasks support a “tags:” attribute for this reason. You can ONLY filter tasks based on tags from the command line with **--tags** or --**skip-tags**. Adding “tags:” in any part of a play (including roles) adds those tags to the contained tasks.

**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 run just 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 " configuration "**

You can apply the same tag name to more than one task, in the same file or included files. This will run all tasks with that tag.

**- 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**

tags can be applied to entire play and roles as well, like,,,

**- hosts: all**

**tags:**

**- bar**

**tasks:**

**roles:**

**- { role: webserver, port: 5000, tags: [ 'web', 'foo' ] }**

There is a special always tag that will always run a task, unless specifically skipped

**tasks:**

**- debug: msg="Always runs"**

**tags:**

**- always**

**Includes:**

**1] Dynamic includes:**

Until Ansible 2.0.0, includes were processed when your playbook run started, so you couldn’t do things like load a particular include when some condition was met. Ansible 2.0.0 evaluates includes during playbook execution, so you can do something like the following:

**# Include extra tasks file, only if it's present at runtime.**

**- name: Check if extra\_tasks.yml is present.**

**stat: path=extras/extra-tasks.yml**

**register: extra\_tasks\_file**

**connection: local**

**- include: tasks/extra-tasks.yml**

**when: extra\_tasks\_file.stat.exists**

If the file tasks/extra-tasks.yml is not present, Ansible skips the include.

**2] Handler includes**

Handlers can be included just like tasks, within a playbook’s handlers section. For example:

**handlers:**

**- include: included-handlers.yml**

This can be helpful in limiting the noise in your main playbook, since handlers are usually used for things like restarting services or loading a configuration, and can distract from the playbook’s primary purpose.

**3] Playbook includes**

Playbooks can even be included in other playbooks, using the same include syntax in the top level of your playbook. For example, if you have two playbooks—one to set up your webservers (web.yml), and one to set up your database servers (db.yml), you could use the following playbook to run both at the same time:

**tasks:**

**[...]**

**- include: web.yml**

**- include: db.yml**

**Inventory File:**

The default location of inventory file is “/etc/ansible/hosts” but it can changed and be later called upon while executing the playbook using this syntax **”ansible-playbook playbook.yml -i ~/path/to/file ”** . As stated earlier we can define groups in square brackets. Like this

**[web-server]**

**Hosts1**

**Hosts2**

**…………**

And one more feauture of it is we can define variable for a particular group as well in ventory file which is not recommended but still

**[web-server:vars]**

**ansible\_ssh\_user=tempuser**

**foo=bar**

We can also dile varibles beside host like this,

**[web-server]**

**hosts1.com ansible\_ssh\_user=tempuser**

We can also define a parent group and add childres to it.

**[netmagic:children]**

**netmagic\_bm**

**netmagic\_vm**

**[netmagic\_bm]**

**153.254.108.165**

**[netmagic\_vm]**

**153.254.108.163**

when we call like this

**# ansible-playbook playbook.yml -l netmagic**

The playbook would be executed on “**netmagic\_bm & netmagic\_vm**” both of them.

**Script (module)**- Runs a local script on a remote node after transferring it.

* The script module takes the script name followed by a list of space-delimited arguments.
* The local script at path will be transferred to the remote node and then executed.
* The given script will be processed through the shell environment on the remote node.
* This module does not require python on the remote system, much like the raw module.

# Example from Ansible Playbooks

**- script: /some/local/script.sh --some-arguments 1234**

# Run a script that creates a file, but only if the file is not yet created

**- script: /some/local/create\_file.sh --some-arguments 1234**

**args:**

**creates: /the/created/file.txt**

# Run a script that removes a file, but only if the file is not yet removed

**- script: /some/local/remove\_file.sh --some-arguments 1234**

**args:**

**removes: /the/removed/file.txt**

meaning of create : a filename, when it already exists, this step will **not** be run.

meaning of remove : a filename, when it does not exist, this step will **not** be run.

**Searching for content**

It is fairly common in Ansible to search a string for a substring. In particular, the common administrator task of running a command and grepping the output for a particular key piece of data is a reoccurring construct in many playbooks.

Let's try using the search filter:

**- name: check database version**

**command: neutron-manage current**

**register: neutron\_db\_ver**

**- name: upgrade db**

**command: neutron-manage db\_sync**

**when: not neutron\_db\_ver.stdout | search('juno')**

This version is much cleaner to follow and does not mask errors from the first task. The search filter searches a string and will return True if the substring is found anywhere within the input string. If an exact complete match is desired instead, the match filter can be used. Full Python regex syntax can be utilized inside the search / match string.

**Ansible - other host facts:**

**Fact Caching:**

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'] }}**

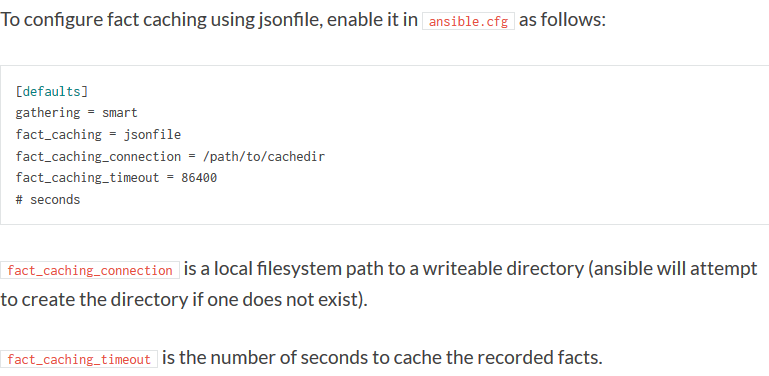
With “Fact Caching” disabled, in order to do this, Ansible must have already talked to ‘asdf.example.com’ in the current play, or another play up higher in the playbook. This is the default configuration of ansible.

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.



**Setting up Ansible/jinja variables:**

The default behavior from ansible and ansible.cfg is to fail if variables are undefined, but you can turn this off.

**{{ variable | mandatory }}**

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.

**Accessing Complex Variable Data:**

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 of a remote host:

**{{ ansible\_eth0["ipv4"]["address"] }}**

Or

**{{ ansible\_eth0.ipv4.address }}**

Similarly, this is how we access the first element of an array:

**{{ foo[0] }}**

We can get this details by the setup module of ansible (gather\_fatcs: yes). To get a list of all such variables we can run

**# ansible example -m setup**

In a playbook we need to mention **gather\_facts: yes** in the beginning top most of playbook before tasks:

**Example #1**

First, the author creates a “users” dictionary:

---

users:

- name: john

email: john@example.com

- name: jane

email: jane@example.com

- name: fred

email: fred@example.com

password: 123!abc

when run the below cmd to, we would be picking u particular content.

**- set\_fact:**

**emails: "{{ users | selectattr('password', 'undefined') | map(attribute='email') | list }}"**

1] The “users” dictionary is being passed to the selectattr( ) filter.

2] The selectattr( ) filter will cycle through all of the dictionaries (also known as objects) inside of “users”, looking

for which ones do and do not have the “password” attribute specified.

3] Because the ‘undefined’ Test has been passed as a parameter, only the objects which do not have a “password”

attribute defined will be recorded.

4] The map( ) filter is then used to return the “email” attribute of all objects that were recorded in the previous

step.

**Example #2**

how **selectattr( )** can be used with both Jinja ( **equalto( )** ) and Ansible ( **match( )** and **search( )** ) tests.

- hosts: localhost

connection: local

gather\_facts: no

vars:

network:

addresses:

private\_ext:

- type: fixed

addr: 172.16.2.100

private\_man:

- type: fixed

addr: 172.16.1.100

- type: floating

addr: 10.90.80.10

We can obtain the “floating” value using any one of following filters:

**- debug: msg={{ network.addresses.private\_man | selectattr("type", "equalto", "floating") | map(attribute='addr') | list }}**

**- debug: msg={{ network.addresses.private\_man | selectattr("type", "match", "^floating$") | map(attribute='addr') | list }}**

**- debug: msg={{ network.addresses.private\_man | selectattr("type", "search", "^floating$") | map(attribute='addr') | list }}**

**# Example setting host facts using complex arguments**

We can even set a variable using the above method by dynamically getting info and setting it up as a variable.

**- set\_fact:**

one\_fact: something

other\_fact: "{{ local\_var \* 2 }}"

**another\_fact: "{{ some\_registered\_var.results | map(attribute='ansible\_facts.some\_fact') | list }}"**

**Extracting values from containers:**

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 }}**

The results of the above expressions would be:

['x', 'z']

**{{ ['x','y']| map('extract', {'x': 42, 'y': 31}) |list }}**

[42, 31]

**Return the union of two lists:**

**{{ [1, 2, 3] | union([2, 3, 4]) | join(', ') }}**

Returns:

1, 2, 3, 4

**Intersect:**

Return the intersection of two lists.

**{{ [1, 2, 3] | intersect([2, 3, 4]) | join(', ') }}**

Returns:

2, 3

**Difference:**

Return the difference of two lists.

**{{ [1, 2, 3] | difference([2, 3, 4]) | join(', ') }}**

Returns:

1

**List files:**

Return a recursive list of files under a specific path.

**{{ '/etc/salt/' | list\_files | join('\n') }}**

Returns:

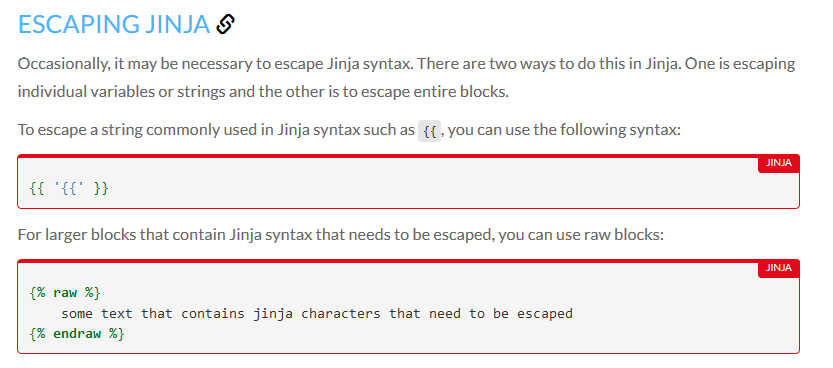
/etc/salt/master

/etc/salt/proxy

/etc/salt/minion

/etc/salt/pillar/top.sls

/etc/salt/pillar/device1.sls



**Configure MySQL:**

**- name: Remove the MySQL test database.**

**mysql\_db: db=test state=absent**

**- name: Create a database for Drupal.**

**mysql\_db: "db={{ domain }} state=present"**

MySQL installs a database named test by default, and it is recommended that you remove

the database as part of MySQL’s included mysql\_secure\_installation tool. The first step in

configuring MySQL is removing this database. Next, we create a database named {{ domain }} the database is named the same as the domain we’re using for the Drupal site.

<https://relativkreativ.at/articles/how-to-use-ansibles-lineinfile-module-in-a-bulletproof-way>

Removing unwanted lines can be done with this task:

- name: Remove lines with unwanted occurrences of ListenAddress

lineinfile: dest=/etc/ssh/sshd\_config

regexp="^ListenAddress (?!1.2.3.4)"

state=absent

What's not so obvious here is the (?!1.2.3.4): This is a regular expression construct called negative lookahead:

The regular expression ^ListenAddress (?!1.2.3.4) matches lines starting with ListenAddress, followed by a space, then not followed by 1.2.3.4. For our task, this means that all ListenAddress directives are removed except those which specify the IP address 1.2.3.4.

**.**

**Let's edit our example sshd\_config file to make the SSH daemon listen on multiple IP addresses.**

**- name: Remove lines with unwanted occurrences of ListenAddress**

**lineinfile: dest=/etc/ssh/sshd\_config**

**regexp="^ListenAddress (?!46.101.70.239|10.19.0.6)"**

**state=absent**

**Other options for ansible-playbook:**

The ansible-playbook command also allows for some other common options:

• --inventory=PATH (-i PATH): Define a custom inventory file (default is the default Ansible

inventory file, usually located at /etc/ansible/hosts).

• --verbose (-v): Verbose mode (show all output, including output from successful options).

You can pass in -vvvv to give every minute detail.

• --extra-vars=VARS (-e VARS): Define variables to be used in the playbook, in "key=value,key=value"

format.

• --forks=NUM (-f NUM): Number for forks (integer). Set this to a number higher than 5 to

increase the number of servers on which Ansible will run tasks concurrently.

• --connection=TYPE (-c TYPE): The type of connection which will be used (this defaults to

ssh; you might sometimes want to use local to run a playbook on your local machine, or on

a remote server via cron).

• --check: Run the playbook in Check Mode (‘Dry Run’); all tasks defined in the playbook will

be checked against all hosts, but none will actually be run.

======================================================================

#ansible all –m setup

Note:

If Ansible reports No hosts matched or returns some other inventory-related error,try setting the ANSIBLE\_HOSTS environment variable explicitly: export ANSIBLE\_- HOSTS=/etc/ansible/hosts. Generally Ansible will read the file in /etc/ansible/hosts automatically, but depending on how you installed Ansible, you may need to explicitly set ANSIBLE\_HOSTS for the ansible command to work correctly.

By default, Ansible will run your commands in parallel, using multiple process forks, so thecommand will complete more quickly. If you’re managing a few servers, this may not be muchquicker than running the command serially, on one server after the other, but even managing 5-10 servers, you’ll notice a dramatic speedup if you use Ansible’s parallelism (which is enabled by default).

If we want to run cmds sequentially, i.e one server at a time add “-f 1” tag, we can also set it to assign no of servers to run this cmd at a time like 10 server, 20 …….. “-f 10” .

If we want to get the enviromental details of the target hosts run

# ansible example -m setup.

- name: Configure kernel parameters

lineinfile:

dest: /etc/sysctl.conf

regexp: "^{{ item.property }}="

line: "{{ item.property }}={{ item.value }}"

with\_items:

- { property: 'kernel.shmall', value: '2097152' }

- { property: 'kernel.shmmax', value: '134217728' }

- { property: 'fs.file-max', value: '65536' }

In ansible, I can use something like:

debug:var="{{lookup('pipe', 'date +%Y%m%d')}}"

The pipe is an ansible lookup plugin that will calculate the output of the shell command you specify in lookup's second parameter, and pipe it to the left side of your lookup. You can specify any shell command there.

Therefore in your example, the output of shell command date +%Y%m%d should be pipe'd to the debug module and var should be set to this output value.

I generally use pipe lookup to set a fact for timestamp so that I can append timestamp in a variable at the end for any resource names, like this -

- set\_fact: timestamp="{{ lookup('pipe', 'date +%Y%m%d%H%M%S') }}"

You can also use an array variable to define your items:

---

- hosts: all

sudo: true

vars:

packages: [ 'vim', 'git', 'curl' ]

tasks:

- name: Install Package

apt: name={{ item }} state=latest

with\_items: packages

- name: Installing Python module

pip: name=MySQL-python

- name: Server configuration

lineinfile: dest=/etc/my.cnf line={{ item }} mode=0644 create=yes

with\_items:

- bind-address=0.0.0.0

- log-bin

- server\_d={{ ansible\_all\_ipv4\_addresses[0].split('.')[3] }}

- log-basename=log{{ ansible\_all\_ipv4\_addresses[0].split('.')[3] }}

- name: Client configuration

lineinfile: dest=/root/.my.cnf line={{ item }} mode=0600 create=yes

with\_items:

- "[client]"

- user=root

- password={{ rootpwd }}

- name: Securing root account

mysql\_user: name=root password={{ rootpwd }} state=present host=localhost

- name: Remove anonymous users

raw: mysql -e "DELETE FROM mysql.user WHERE user='';"

- name: Set root permission

raw: mysql -e "DELETE FROM mysql.user WHERE user='root' AND host!='localhost';"