

```
=====
```

```
vi etc/sudoers.d
```

```
ansible ALL=(ALL) NOPASSWD:ALL
```

```
--- generate ssh keys on master =====
```

```
ssh-keygen -t rsa
```

copy the pub file and paste in .ssh/authorized\_keys in client node before that run the below commands on client node

```
mkdir -p .ssh
```

```
cd .ssh/  
vi authorized_keys  
chmod 700 .ssh/  
cd .ssh/  
chmod 640 authorized_keys
```

```
=====
```

to set the idle time

```
echo 'ClientAliveInterval 50' | sudo tee --append /etc/ssh/sshd_config  
/bin/systemctl restart sshd.service
```

```
=====
```

Inventories -- two types

static and dynamic

```
--group  
--variables
```

default inventory location is /etc/ansible

alternatively inventory location can be specified through the ansible.cfg file

ex:

```
[webservers]  
host1  
host2
```

```
[dbservers]  
host3  
host4
```

```
[servers:children] --nested groups  
webservers  
dbservers
```

-- you can create your own inventory file --

```
vi inventory
```

```
ansible -i inventory all --list-all
```

```
=====ad hoc commands ===== ----
```

```
ansible all -m ping
```

```
ansible all -m shell -a "df -h" -- to list the directories
```

```
-a we need to use before the command
```

```
ansible all -m yum -a "name=httpd state=present" -- to install the package
```

```
ansible all -m yum -a "name=httpd state=removed" -- to remove the package
```

```
to install package -- present
```

```
to remove the package -- removed
```

```
-- to start the service
```

```
ansible all -m service -a "name=http state=started"
```

```
service start --started
```

```
service stop -- stopped
```

```
service restart -- restarted
```

```
-----copy module --
```

```
to copy a file
```

```
ansible all -m copy -a "src=/root/divyansh dest=/root/"
```

```
=====
```

```
user is the module to create
```

```
to create the user
```

```
ansible all -m user -a "name=divyansh password=divyansh"
```

```
or ansible all -m user -a "name=divyansh state=present"
```

```
to remove
```

```
ansible all -m user -a "name=divyansh state=absent"
```

```
=====
```

```
How to create a directory
```

```
ansible all -m file -a "name=sample state=directory mode=777"
```

```
here mode we are using it for setting up the permissions
```

```
to delete the directory
```

```
ansible all -m file -a "name=sample state=absent"
```

=====

to create a file

```
ansible all -m shell -a "name=subuli state=file mode=777"
```

to list the available modules -- ansible-doc -l

if you want to know more about module --ansible-doc user

=== To check playbook syntax === ansible-playbook --syntax-check <ymlfilename>

ansible-playbook -C <yml file> dry run -- use -C option

if you want to know more details of the output use -vvvv option at the end

=====

remote\_user: the name of the remote user

become: to enable or disable the privilege escalation

become\_method: to allow using an alternative escalation solution

become\_user: the target user used for privilege escalation

=====variables in ansible=====

you can define variables in playbook -- but this is not recommended, as playbook should be static and reusable

instead of defining variable in playbook , we can define variable in below

inventory (deprecated)

inclusion files

local fact

variable precedence --

Variables scope:

Global Scope: this is when a variable is set from inventory or the command line

Play Scope: this is applied when it is set from a play

Host scope: this is applied when set in inventory or using a host variable

inclusion file

--> when the same variable is set at different levels, the most specific level gets precedence

host

playbook

global scope

---> when a variable is set from the command line, it will overwrite anything else

ex: ansible-playbook site.yml -e "web\_package=apache"

---> inventory at hostlevel variables define

```
[web_servers]
```

```
web_server_1 ansible_user=centos http_port=80
web_server_2 ansible_user=ubuntu http_port=8080
```

at hostgroup level:

```
[web_servers:vars]
ansible_user=centos
http_port=80
```

Register a variable:

Ansible register variable or ansible register module is used to capture or store the output of the command or task. By using the register module, you can store that output into any variable.

array as variable syntax:

```
- hosts: all
  vars:
    <variablename>:
      - <arrayvalue1>
      - <arrayvalu2>
      - <arrayvalue3>
```

Disctionary as variable:

```
- hosts: all
  vars:
    <variablename>:
      <key>: <value>
```

=====ansible vault=====

Ansible Vault is a feature that allows users to encrypt values and data structures within Ansible projects.

commands:

```
create: to create ansible vault file in the encrypted format
view: to view data of encrypted file
edit: to edit encrypted file
encrypt: to encrypt an unencrypted file (meaning if you want to encrypt your
existing file)
decrypt : to decrypt an encripted file
```

```
--ask-vault-pass : to provide password while running playbook
```

```
--vault-password-file: to pass a vault password through a file
```

example: here vault.yaml is the file name

- 1) ansible-vault create vault.yaml
- 2) ansible-vault view vault.yaml

- 3) ansible-vault edit vault.yaml
- 4) ansible-vault decrypt vault.yaml
- 5) ansible-vault encrypt vault.yaml

=====

=====Ansible facts:=====

Notes:

---> Ansible facts are variables that are automatically set and discovered by ansible on managed hosts  
 ----> facts contain information about hosts that can be used in conditionals  
 ---> for instance, before installing specific software you can check that a managed host runs a specific kernel version  
 ----> By default, all playbooks perform fact gathering before running the actual plays  
 ---> you can run facts gathering in an adhoc command using the setup module  
 ---> to show facts, use debug module to print the value of the ansible\_facts variable

Ansible Gathered Facts or playbook variables belongs to one of the following types

Dictionary  
 List  
 AnsibleUnsafeText

How to know the data type of variable (or) fact: {{ <the variable name> | type\_debug }}

custom facts:  
 steps to create custom facts:

- 1) create /etc/ansible/facts.d on remote machine
- 2) Inside of facts.d place one or more custom facts files (facts file extension should be \*.fact)
- 3) the output of \*.fact file should be json
- 4) the \*.fact file should have execution permission

```
#!/bin/bash
git_ver=$(git --version | awk '{ print $3 }')
echo "{\"Git_Version\":\"${git_ver}\"}"
```

example playbook is here: ansible\_local is the default variable that we can use to fetch the custom facts

```
- hosts: all
  tasks:
    - name: fetching custom facts from remote machine
      debug:
        msg: "the custom fact is {{ansible_local}}"
```

-----

Conditionals  
 Loops  
 Roles  
 User mailing list  
 filters-- Jinja2 filters and their templates

====conditions====

when:

syntax

tasks:

module: your code

when: <expression>

if your expression is true then will execute the module otherwise it will not execute

Register: Register key is used to store the response to a variable after completion of each task

we have to use at the end of each task as follows

---delegate to -- the below task execute only on local host even if you are using hosts:all in header

---

- name: name of the playbook

- hosts: all

- tasks:

- command:

- touch: /tmp/vardhan.txt

- delegate\_to: localhost

when can be used to test multiple conditions as well

use and or or and group the conditions with parentheses

ex: when: ansible\_distribution == "Centos" or(and)

ansible\_distribution == "RedHat"

the when keyword also supports a list, and when using a list, all of the conditions must be true

complex conditional statements can group conditions using parentheses

=====handlers=====

--> handlers allow you to configure playbooks in a way that one task will only run if another task has been running successfully

--> in order to run the handler, a notify statement is used from the main task to trigger the handler

--> handlers typically are used to restart services or reboot hosts

--->handlers are executed after running all tasks in a play

---> handlers will only run if a task has changed something, so if an ok result instead of changed result is reported, the handler will not run

--> if one of the task fails, the handler will not run, but this may be overwritten using force\_handlers:True

--> one task may trigger more than one handler

=====Blocks=====

--> A block is a logical group of tasks

--> it can be used to control how tasks are executed

--> one block can, for instance, be enabled using a single "when"

--> Blocks can also be used in error condition handling

- use "block" to define the main tasks to run

- use "rescue" to define tasks that run if tasks defined in the block fail

- use "always" to define tasks that will run, regardless of the success or failure of the "block" and "rescue" tasks

--> Notice that items cannot be used on blocks

example:

tasks:

- block:

```

- name: upgrade the database
shell:
cmd: /usr/local/lib/upgrade-database
rescue:
- name: revert the database upgrade
shell:
cmd: /usr/local/lib/revert-database
always:
- name: always restart the database
service:
name: mariadb
state: restarted

```

#### =====Understanding failure handling==

```

--> Ansible looks at the exit status of task to determine whether it has failed
--> When any task fails, ansible aborts the rest of the play on that host and
continues with the next host
--> Different Solutions can be used to change that behaviour
--> use "ignore_errors" in task to ignore failures
--> use "force_handlers" to force a handler that has been triggered to run, even
if(another) task fails

```

#### == Defining Failure States==

```

--> As ansible only looks at the exit status of a failed task, it may think a task
was successful where that is not the case
--> to be more specific., use "failed_when" to specify what to look for in command
output to recognize a failure

```

#### =====using the fail module=====

```

--> The "failed_when" keyword can be used in task to identify when a task has
failed
--> The "fail" module can be used to print a message that informs why a task has
failed
--> To use "failed_when" or "fail" , the result of the command must be registered,
and the registered variable output must be analyzed
--> When using the "fail" module, the failing task must have "ignore_errors" set to
yes

```

#### =====loop===== loop=with\_\*

The loop keyword is the current keyword replaced by with\_items

The syntax will be probably be deprecated in future versions of ansible

```

with_items: equivalent to loop keyword
with_file: the item contains a file, which contents is used to loop through
with_sequence: generates a list of values based on numeric sequence

```

#### =====Tags=====

```

ansible-playbook tags.yml --skip-tags "tagname"    -- to skip the tags

ansible-playbook tags.yml --tags "tagname"         -- to run only specified tag in the
playbook

```

example playbook:

```
---
```

```

- name: demonstrating tags
  hosts: all
  tasks:
    - name: tagging
      yum:
        name: zip

```

```

    state: latest
  tags: prod
- name: installing httpd
  yum:
    name: httpd
    state: latest

```

## SPECIAL TAGS

Ansible has a special tag that can be assigned in a playbook: `always`. This tag causes the task to always be executed even if the `--skip-tags` option is used, unless explicitly skipped with `--skip-tags always`.

There are three special tags that can be used from the command-line with the `--tags` option:

1. The `tagged` keyword is used to run any tagged resource.
2. The `untagged` keyword does the opposite of the tagged keyword by excluding all tagged resources from the play.
3. The `all` keyword allows administrators to include all tasks in the play. This is the default behavior of the command line.

## Error handlings

### ===Specifying When a Task Reports "Changed" Results

When a task makes a change to a managed host, it reports the changed state and notifies

handlers. When a task does not need to make a change, it reports `ok` and does not notify handlers.

The `changed_when` directive can be used to control when a task reports that it has changed.

For example, the `shell` module in the next example is being used to get a Kerberos credential

which will be used by subsequent tasks. It normally would always report `"changed"` when it runs. To

suppress that change, `changed_when: false` is set so that it only reports `"ok"` or `"failed"`.

```

- name: get Kerberos credentials as "admin"
  shell: echo "{{ krb_admin_pass }}" | kinit -f admin
  changed_when: false

```

### Specifying Task Failure Conditions

You can use the `failed_when` directive on a task to specify which conditions indicate that the

task has failed. This is often used with `"run command"` modules that may successfully execute a

command, but the command's output or exit code may indicate a failure.

For example, you can run a script that outputs an error message and use that message to define

the failed state for the task. The following snippet shows how the `failed_when` keyword can be used in a task:

```

tasks:

```



```
- shell: /usr/local/bin/create_users.sh
  register: command_result
  failed_when: "'Password missing' in command_result.stdout"
```

```
=====Jinja2
templates=====
{% ... %} for control statements
{{ ... }} for expressions
{# ... #} for comments
```

```
=====roles
```

Ansible Roles: it is a place the group of playbooks in one file. it will help reduce the code complexity, and increases the code reusability

```
--> ansible-galaxy init lamp
    lamp will be created successfully
default directories/files that will create once you create a role
```

```
lamp/
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
├── vars
│   └── main.yml
```

Default: The main.yml file in this directory contains the default values of role variables that can be overwritten when the role is used.

files: This directory contains static files that are referenced by role tasks.

Handlers: The main.yml file in this directory contains the role's handler definitions.

meta: The main.yml file in this directory contains information about the role, including author, license, platforms, and optional role dependencies.

Tasks: The main.yml file in this directory contains the role's task definitions.

Templates: This directory contains Jinja2 templates that are referenced by role tasks

tests: This directory can contain an inventory and test.yml playbook that can be used to test the role

vars: The main.yml file in this directory defines the role's variable values.

CONTROLLING ORDER OF EXECUTION

Normally, the tasks of roles execute before the tasks of the playbooks that use them. Ansible provides a way of overriding this default behavior: the `pre_tasks` and `post_tasks` tasks.

The `pre_tasks` tasks are performed before any roles are applied. The `post_tasks` tasks are performed after all the roles have completed.

---

example:

```
- hosts: remote.example.com
```

```
pre_tasks:
```

```
- debug:
```

```
msg: 'hello'
```

```
roles:
```

```
- role1
```

```
- role2
```

```
tasks:
```

```
- debug:
```

```
msg: 'still busy'
```

```
post_tasks:
```

```
- debug:
```

```
msg: 'goodbye'
```

=====Ansible galaxy=====

- Ansible Galaxy is a repository for Ansible Roles that are available to drop directly into your Playbooks to streamline your automation projects

to create a role use the below command

```
ansible-galaxy init <rolename>
```

===Ansible tower===

```
sed -i 's|admin_password=. *|admin_password=Bangalore@123|g' inventory
```

```
0U3gI7BNSNlyxrfDBcAN/J30I1eKeAaTVkTONTUZ
```

```
sed -i 's|secret_key=. *|secret_key=0U3gI7BNSNlyxrfDBcAN/J30I1eKeAaTVkTONTUZ|g'
```

```
inventory
```

```
d52f3955c7d20940d827f4431c09c07336380436
```