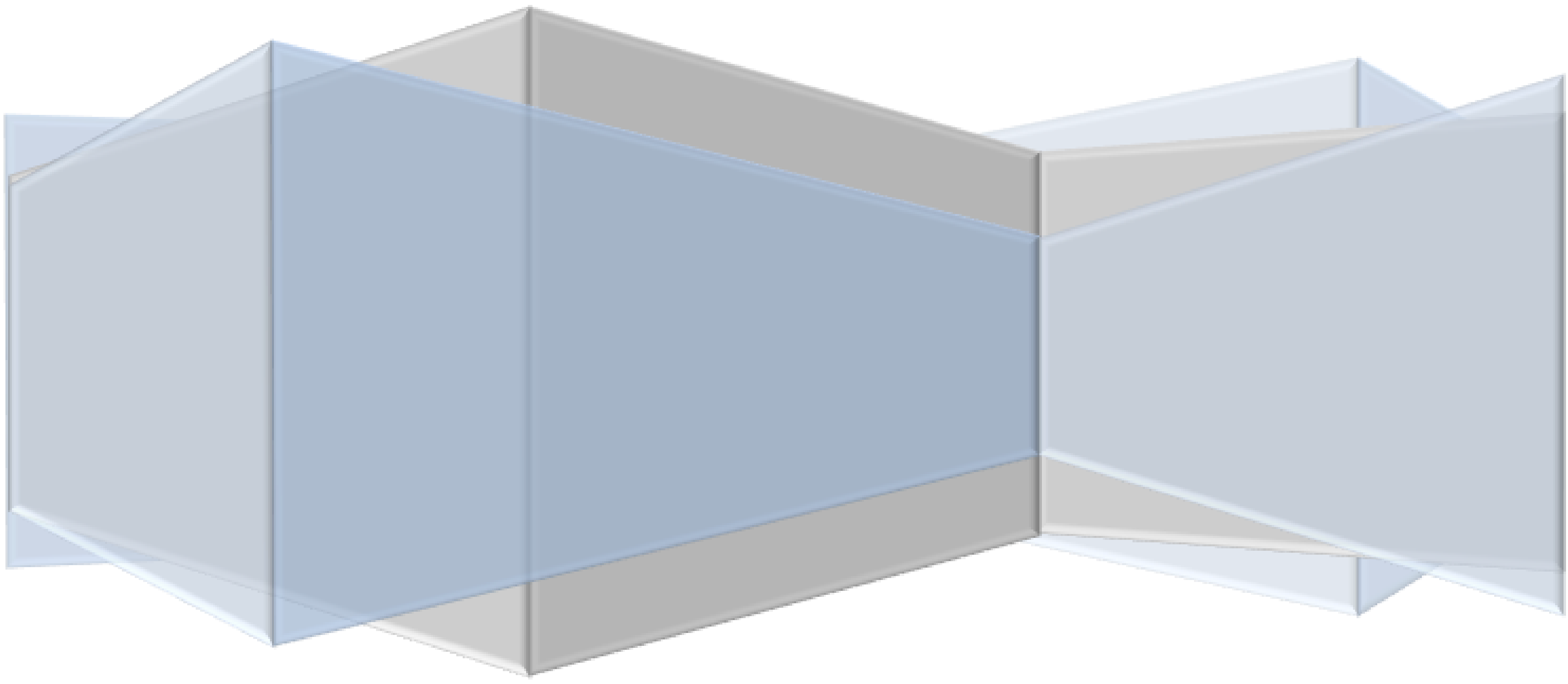
Using Ansible

Ganesh Palnitkar



Index

Download and install Ansible RPM from,
https://releases.ansible.com/ansible/rpm/release/epel-7-x86_64/

To start working with Ansible server and nodes, we have to make sure to comply pre-requisites, like establish ssh key sharing from Ansible server to nodes. To have Python running on the nodes.

```
vagrant@ansibleserver:~/project$ ssh-keygen
```

$$\underbrace{\texttt{vagrant@ansibleserver:\textasciitilde/project\$ssh-copy-id -i}}_{\text{\{source of ssh key\}}}\ \underbrace{\texttt{\textasciitilde/.ssh/id\_rsa.pub}}_{\text{\{Location of ssh key\}}}\ \underbrace{\texttt{vagrant@192.16.33.11}}_{\text{\{Target host and user\}}}$$

Now to allow Ansible to connect to the `localhost` (for local change), i.e. the ansible control server, we will copy the public key of the user to `authorized_keys` file in `~/.ssh` folder.

```
$ cp ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys
```

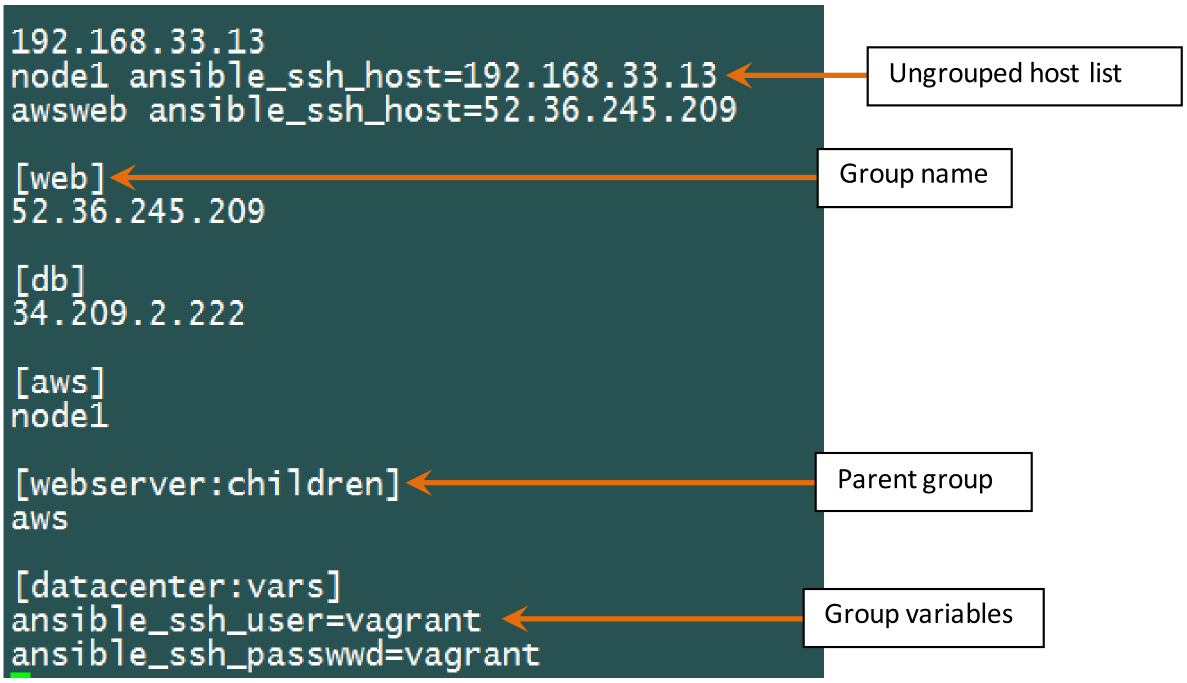
Ansible Components on ansible control server:

1) **Inventory file**
2) **Ansible configuration file**

Inventory file is the one in which we write all target (remote) servers that we want to manage using Ansible. This file can be updated manually or using a plugin, can be updated dynamically.

Inventory is maintained in two ways**, static inventory** and **dynamic inventory.**

Static inventory file is maintained as shown below.



```
192.168.33.13
node1 ansible_ssh_host=192.168.33.13
awsweb ansible_ssh_host=52.36.245.209

[web]
52.36.245.209

[db]
34.209.2.222

[aws]
node1

[webserver:children]
aws

[datacenter:vars]
ansible_ssh_user=vagrant
ansible_ssh_passwwd=vagrant
```

A sample inventory file looks like the one shown below. This file should be written in YAML format.

YAML format of inventory file:

```
all:
  children:
    master:
      children:
        dbserver:
          hosts:
            192.168.33.30:
              package: apache2
              username: user1
            ubuntu-2:
              package: apache2
              username: user1
        webserver:
          hosts:
            192.168.33.30: {}
            localhost:
              package: apache2
              username: user1
            ubuntu-2: {}
    ungrouped:
      hosts:
        ubuntu-1: {}
```

Graph format of inventory:

```
@all:
  |--@master:
  |  |--@dbserver:
  |  |  |--192.168.33.30
  |  |  |--ubuntu-2
  |  |--@webserver:
  |  |  |--192.168.33.30
  |  |  |--localhost
  |  |  |--ubuntu-2
  |--@ungrouped:
  |  |--ubuntu-1
```

Inventory Management is important for effective and efficient use of Ansible to control your entire environment.

We can break it up in to the environments, like production and test etc. Thus, maintaining separate inventory files for specific environments.

Dynamic Inventory: Dynamic Inventory is maintained / generated using a some sort of a script, like a simple **nmap** command can also be used for gathering hosts in selected subnet, or the **ec2.py** file that can help in gathering a list of all hosts in a ec2 environment.

Understanding the Ansible defaults in the `ansible.cfg` file:

To know about what all options can be set in the ansible configuration file, visit, www.docs.ansible.comand in 'getting started' look for configuration file details. Here we can see all options that can be set in the configuration file as defaults.

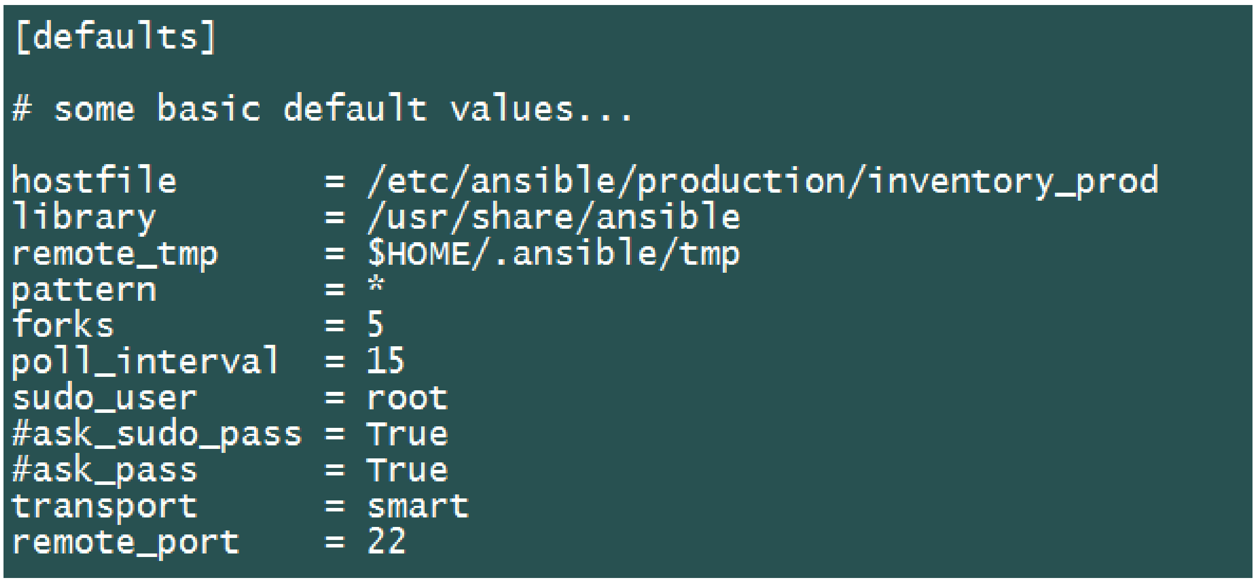
The options set in the ansible.cgf in the current directory have the least precedence. The option set in the environment variable has the highest precedence.

To test this, we can set the default key value, 'host_key_checking = False' in the ansible.cfgfile, and then try running the ping module on a target remote server. With this default key-value setting, we can override the requirement to check the host_key.

Few more default settings that we can modify and test are,

If we have python 3 installed on a specific remote server, we can update the inventory file and provide a behavioural pattern to be followed for that remote server. This patter is as mentioned below.

`192.168.33.12 ansible_python_interpreter=usr/bin/python2.7` … in this case the python 2.7 is installed in the 'usr/bin' directory. This can be set as per the specific system settings.



```
[defaults]

# some basic default values...

hostfile       = /etc/ansible/production/inventory_prod
library        = /usr/share/ansible
remote_tmp     = $HOME/.ansible/tmp
pattern        = *
forks          = 5
poll_interval  = 15
sudo_user      = root
#ask_sudo_pass = True
#ask_pass      = True
transport      = smart
remote_port    = 22
```

The ‘ansible.cgf’ file has ansible setting that one can modify to suit the environment.

This config file ‘ansible.cgf’ can be copied into the production folder and changes made in the file are then applied only for the playbooks located inside the folder.

Various Ansible commands / utilities

1) `ansible-config` … lists all defaults configuration values from ansible.cfg file. We can view , dump on screen ot list as per the contents of the cfg file.
2) `ansible-connection` … allows user to specify method to be used for connection to host
3) `ansible-console` … A REPL that allows for running ad-hoc tasks against a chosen inventory from a nice shell with built-in tab completion (based on dominis' ansible-shell).
4) `ansible-doc` … list and provides access to module documentation pages.
5) `ansible-galaxy` … gets us access to the public library of roles. Also helps in creating and managing roles in ansible-environment.
6) `ansible-inventory` … helps in managing inventories, static inventory . output can be displayed in YAML format or in graphical representation.
7) `ansible-playbook` … helps in running playbook from command line.
8) `ansible-pull` … Used to pull a remote copy of ansible on each managed node, each set to run via cron and update playbook source via a source repository
9) `ansible-test` … ansible-test provides ways to run different types of tests on your Collections, broadly these tests are of types as, 1) Sanity Tests 2) Unit Tests 3) Integration Tests
10) `ansible-vault` … helps in accessing and managing vaults objects.

Ansible Modules

- Core modules … modules supported by Ansible
- Extras … module updated created by community members and not supported by Ansible
- Deprecated … module that will be removed soon.

`$ ansible-doc -l`… to display all available core modules on ansible repo.
`$ ansible-doc<module name>…` man page for a module
`$ ansible-doc -s <name>…` help with some snippets on who to use a module inside a play book.
Core modules are categorized into multiple groups, like package deployment, network config, virtual machine, etc.
Common module that we can discuss are, 'copy' module, 'fetch', 'apt', 'yum', 'service' module etc.
Let's use the module to install webserver on a centos machine.
We will use the yum module and provide input parameters for the module to work.
Using below command over command line,

```
$ ansible<hostname>-i hosts -m yum -a "name=httpd state=present" --become

$ ansible<hostname> -i hosts -m service -a "name=httpd enabled=yes
state=started" --become
```

On the AWS DB node try running the yum module for package installation.

```
$ ansible<hostname> -i hosts -m apt -a "name=mariadb-server state=latest"
--become

$ ansible<hostname> -i hosts -m service -a "name=mariadb state=started" --
become
```

Understanding the setup module
'Setup' module is a module to gather facts from a system.

```
$ ansible web -i hosts -m setup -a "filter=ansible_os_family"
```

Module usage with example:
File Module:

```
$ ansible web -i inventory -m file -a "name=<path/filename> state touch"

$ ansible web -i inventory -m file -a "path=<path/filename> state touch"
```

Copy Module:

```
$ ansible web -i inventory -m copy -a "src=index.html
dest=/var/www/html/index.html"
```

Template Module

```
$ ansible web -i inventory -m template -a
"src=<path/file.j2>dest=<filepath>"
```

User module

```
$ ansible web - i inventory –m user –a “name=ganeshhp comment=Ganesh
Palnitkar gid=4234 uid=4010”
```

Package Modules

```
$ ansible web – i inventory –m yum –a “name=ntp state=latest”

$ ansible web – i inventory –m apt –a “name=apache2 state=latest”
```

Service module

```
$ ansible web –i inventory –m service –a “name=apache2 state=started
enabled=yes”
```

`$ ansible web -m shell –a <shell command> --become` ….. Here we can pass a shell command to be executed on a remote machine.

Host /group Target pattern

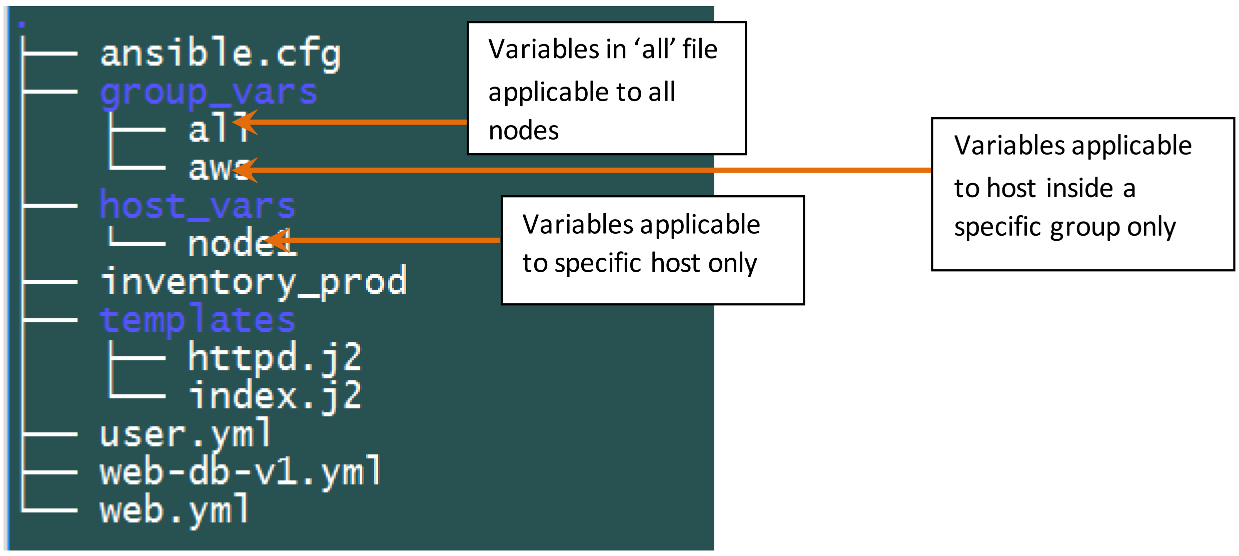
group1:group2 … grp1 OR grp2
!group1 …. Not grp1
web*.autofact.com ….. wildcard
group1:&group2 … host machines that are only common with both groups only be applied with the change.

Variable and Facts:

The Config file has all default parameters listed with default value. Values for nearly all parameters can be changed with uncommenting the parameter declaration.

Using **Variables** in **Ansible** inventory management will help in managing the inventory effectively.

This can be done by creating a folder structure as mentioned below.



```
.
├── ansible.cfg
├── group_vars
│   ├── all
│   └── aws
├── host_vars
│   └── node1
├── inventory_prod
├── templates
│   ├── httpd.j2
│   └── index.j2
├── user.yml
├── web-db-v1.yml
└── web.yml
```

Variables in 'all' file applicable to all nodes
Variables applicable to host inside a specific group only
Variables applicable to specific host only

Here variables mentioned in the 'all' file has the least precedence. Vars mentioned in the 'db' file which is the group_var directory, has the 2nd highest precedence and the variables mentioned in the web1 file inside the host_vars directory has the highest precedence.

Let's test this by using the user module to create a user using the 'username' variable mentioned inside these files.

Using user module to create user on remote node.

```
$ ansible webserver -i hosts -m user -a "name={{username}} password=12345" --become
```

In the '**all' var** file update the below lines. Once updated run the user create module using command mentioned above.

```
- - -
'username': 'ganesh'
```

Update the '**aws' file** with below lines. Once updated run the user create module using command mentioned above. Here name of the file must match with name of the group mentioned in the inventory file.

```
- - -
'username': 'ganesh_aws'
```

Update the **'node1' file** with below lines. Once updated run the user create module using command mentioned above. Here the file name must match with the hostname for which we want to apply the variables.

```
- - -
'username': 'ganesh_node1'
```

Calling variables inside the playbook,

```
---
- name:
  hosts:
  tasks:
  - name: manage users
    user:
      name: "{{ username }}"
      state: present
```

Here the `username` is the variable we are calling.

Overriding variables from the command line,

```
$ ansible-playbook playbook.yml -e "username=newuser"
```

Here **`-e`** options allows us to pass additional variable at run time.

Capturing command output with `register` variable.

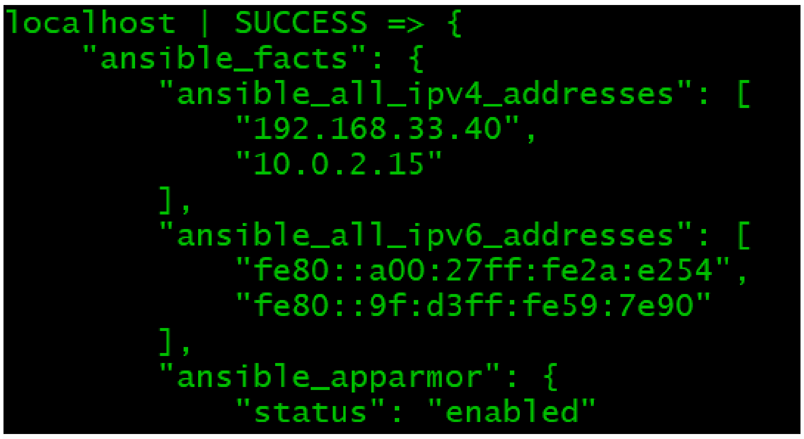
```
---
- name:
  hosts:
  tasks:
  - name: manage users
    package:
      name: apache2
      state: present
    register: install_logs

  - debug: var=install_logs
```

`$ ansible <hostname> -i inventory -m setup -a "filter=ansible_eth*"` ….
This will run the ohai profiler on the remote server and gather facts and return those to the ansible server.

Core Facts:

Here the **setup module** gives us the **core system facts** gathered as shown below.



```
localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "192.168.33.40",
            "10.0.2.15"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::a00:27ff:fe2a:e254",
            "fe80::9f:d3ff:fe59:7e90"
        ],
        "ansible_apparmor": {
            "status": "enabled"
```

Above image is a truncated output of entire core system facts.

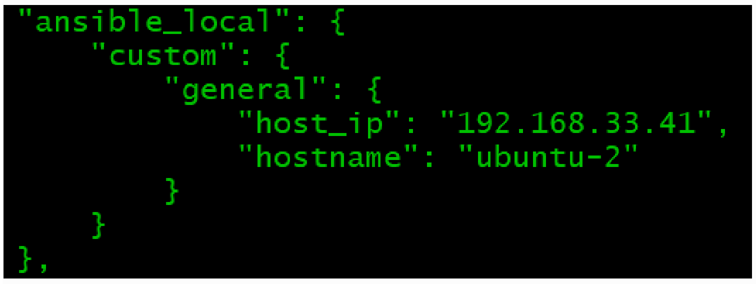
Custom Facts

Along with core system facts, we can also setup **custom facts** by creating a folder structure and file inside it, as shown below.

`/etc/ansible/facts.d/custom.fact`

Here the file name `custom.fact` can be any with file extension default as **`.fact`**

Once we setup this we can try running the setup module and gather facts. The facts will get added with custom facts as well, as shown below.



```
"ansible_local": {
    "custom": {
        "general": {
            "host_ip": "192.168.33.41",
            "hostname": "ubuntu-2"
        }
    }
},
```

To access these custom facts in a template or playbook as,

```
{{ ansible.local[‘custom’][‘general’][‘hostname’] }} or,

{{ ansible.local[‘custom’][‘general’][‘host_ip’] }}
```

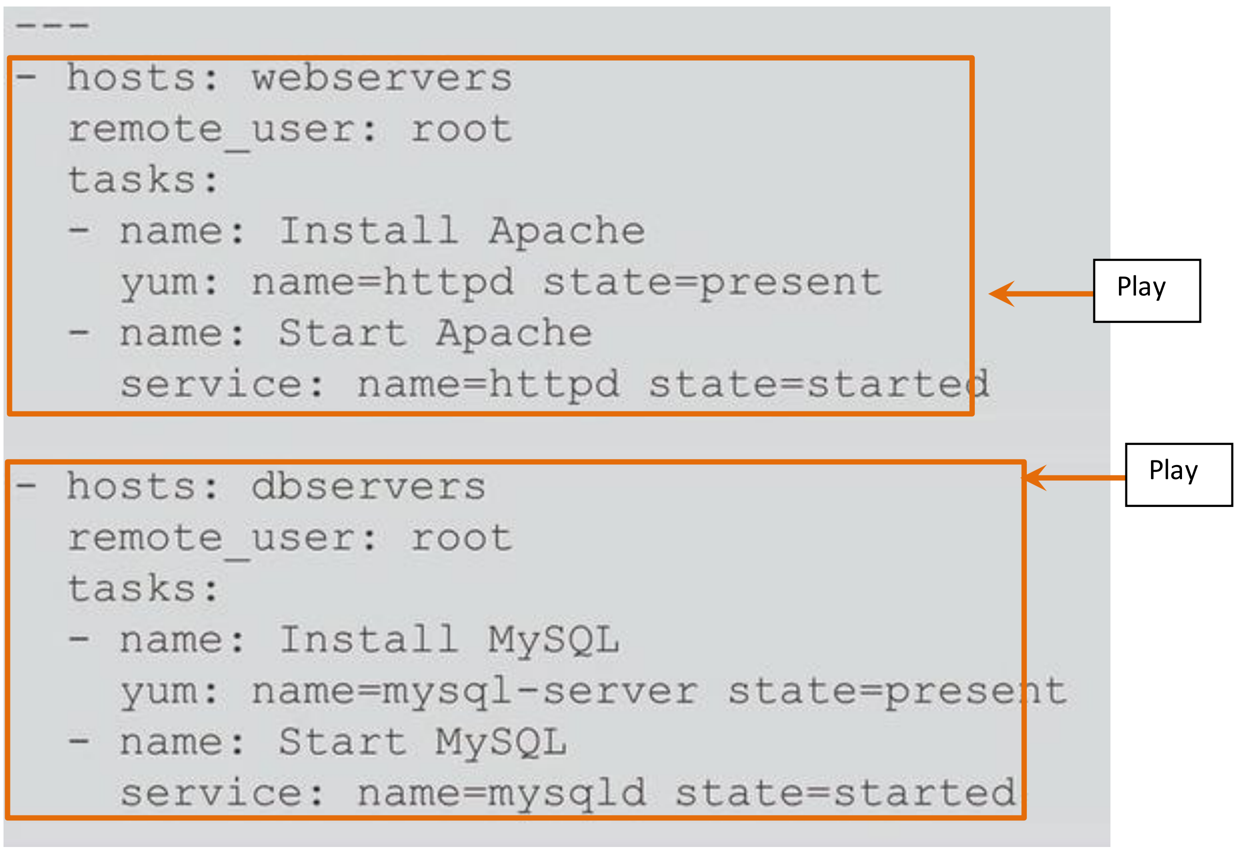
Variables can be used in action lines. Suppose you defined a variable called `vhost` in the `vars` section, you could do this:

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

Play and Playbooks

- Plays help to map the hosts to tasks.
- A play can have multiple tasks
- A playbook can have multiple plays.

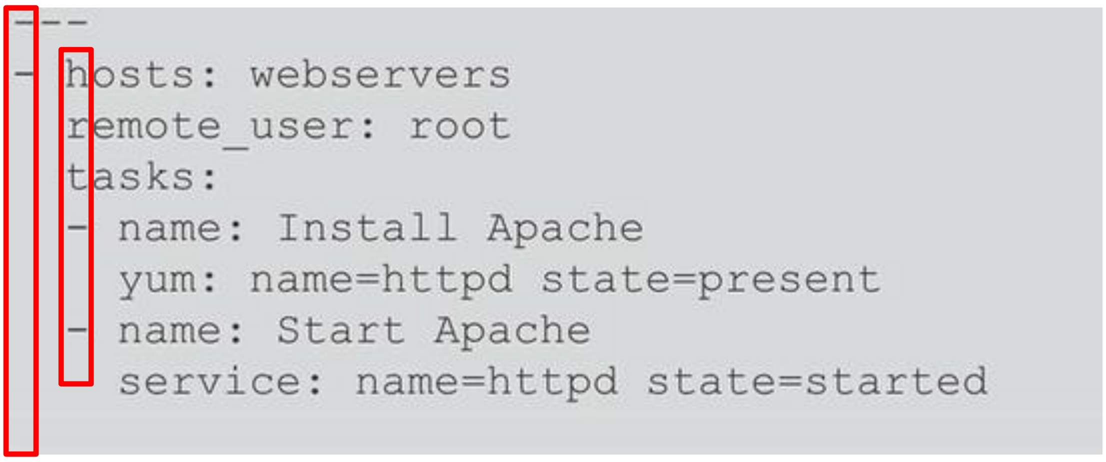
A sample playbook, each coloured rectangle represents a play. Each play is mapped to the host or a group, parent group etc.



```
---
- hosts: webservers
  remote_user: root
  tasks:
  - name: Install Apache
    yum: name=httpd state=present
  - name: Start Apache
    service: name=httpd state=started

- hosts: dbservers
  remote_user: root
  tasks:
  - name: Install MySQL
    yum: name=mysql-server state=present
  - name: Start MySQL
    service: name=mysqld state=started
```

Using **white space** / **indentation** is very specific and has to be followed while writing the play / playbook.



```
---
- hosts: webservers
  remote_user: root
  tasks:
  - name: Install Apache
    yum: name=httpd state=present
  - name: Start Apache
    service: name=httpd state=started
```

Tasks are executed in the order – top down. Thus we have to be careful while specifying the tasks in the play.

Tasks use modules. Each tasks use a module, or in other word a task can can one module at a time.

To execute a playbook use the command,

```
$ ansible-playbook <playbook.yml>
```

Optional parameters to pass while executing playbook,

`$ ansible-playbook playbook.yml --step` … tasks to tasks implementation, like a debug mode of playbook execution.

`$ ansible-playbook --limit playbook.retry <playbook-file-name>` … here the playbook.ertry is the file created after a failed playbook execution on a host.

`$ ansible-playbook playbook.yml --become-user –K –u –e` … here, while running a playbook, if we want to pass a remote-user name, we use the **`–u`** option, **`-K`** is used for passing password and **`-e`** is used for passing extra variable at run time.

A sample playbook, note the indentation and syntax used for writing tasks and modules.

```
-   hosts: webserver
    remote_user: root
    become: yes
    tasks:
    - name: ensure apache is at the latest version
      yum: name=httpd state=present
    - name: start the apache service
      service: name=httpd state=started enabled=yes

-   hosts: dbserver
    remote_user: root
    become: yes
    tasks:
    - name: ensure MySQL is installed
      apt: name=mariadb-server state=present
    - name: ensure that MySQL service is started
      service: name=mariadb state=started
```

Changing order of task implementation using Order instruction

We can **control the order** in which hosts are implemented with tasks. The default is to follow the order supplied by the inventory:

```
    hosts: all
    order: sorted
    gather_facts: False
    tasks:
    - name:
```

Possible values for order are:

inventory:

The default. The order is ‘as provided’ by the inventory

reverse_inventory:

As the name implies, this reverses the order ‘as provided’ by the inventory

sorted:

Hosts are alphabetically sorted by name

reverse_sorted:

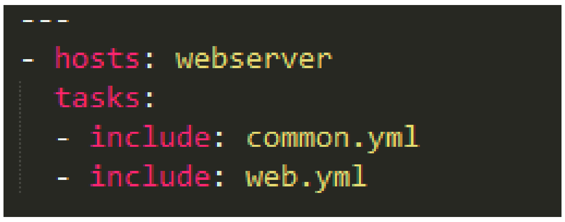
Hosts are sorted by name in reverse alphabetical order

shuffle:

Hosts are randomly ordered each run

Call a playbook inside other playbook.

One can also call a playbook inside a play., like shown below.,



```
---
- hosts: webserver
  tasks:
  - include: common.yml
  - include: web.yml
```

Tasks control using Handlers:

Notify Function can be used in the Playbook that calls the Handler. Handlers are list of Tasks listed under the declaration as ***handlers*** that are invoked only on a certain condition in the execution of tasks.

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

For failed task in the playbook execution one can use below command so as Ansible will confirm on running a particular task before executing it.

```
$ ansible-playbookplaybook.yml --step.
```

The step asks you on before executing each task and you could choose (N)o/(y)es/(c)ontinue.

Parallelism:

While we run a playbook on a group of hosts, by default the each task is executed on all hosts serially and then the next task is taken for implementation. We can change the order of execution and set the batch size by using the `serial` declaration and value can be set to number of hosts in one group of parallel execution. This can also be set by using the percentage value as stated below.

```
---
- name:
  hosts:
  serial: 2 … or
  serial: “50%”  … or,
  serial:
       - 1
       - 5
       - 10
```

Roles:

In order to create roles, create a Directory named as Role inside the Ansible directory.

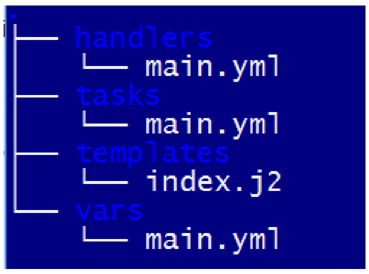
The role folder structure can also be created using the ansible-galaxy utility.

About ansible-galaxy

```
$ ansible-galaxy init <role name>
```

This command will create a default folder structure with required files placed in it.

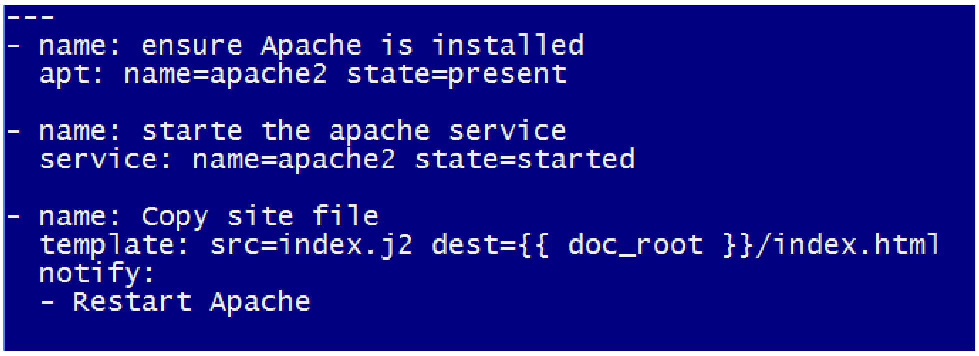
As shown below create directories and files inside the roles directory. Here 'webserver' is the role.



```
├── handlers
│   └── main.yml
├── tasks
│   └── main.yml
├── templates
│   └── index.j2
└── vars
    └── main.yml
```

The tasks will be located in the tasks directory and written in main.yml file.

Roles hold the tasks inside the tasks folder in the main.yml file. So the main.yml file inside the tasks folder would appear as shown below.,



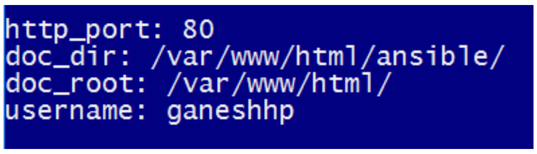
```
---
- name: ensure Apache is installed
  apt: name=apache2 state=present

- name: starte the apache service
  service: name=apache2 state=started

- name: Copy site file
  template: src=index.j2 dest={{ doc_root }}/index.html
  notify:
  - Restart Apache
```

The file will only contain tasks and no hosts statement or vars definitions, etc.

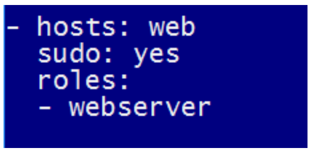
The 'vars' folder thus has the main.yml file to store all variables for the role.



```
http_port: 80
doc_dir: /var/www/html/ansible/
doc_root: /var/www/html/
username: ganeshhp
```

Similar to this the template folder would contain the template file, like index.j2, etc. and main.yml file inside 'handler' folder will have action statement as notified in 'notify' command.

In order to call the role inside a playbook we can use below syntax. The playbook in such case will be located outside the 'roles' folder. And will typically have syntax similar to one mentioned below.



```
- hosts: web
  sudo: yes
  roles:
  - webserver
```

Here in this file we are calling the role with the 'roles' statement and then mentioned the role name.

Ansible Galaxy for remote Role repository

https://galaxy.ansible.com

Ansible provides a repository of ready-to-use roles for almost all requirements that you can think of.

Just pull the role to your Ansible controller and start using.

On the Galaxy web link we can explore all different roles categorized as, 'Most starred', 'Most watched', 'Most Downloaded', etc.

To install a particular role on to the Ansible control server, we can use the command,

Some ansible-galaxy commands

```
$ ansible-galaxy { init, remove, delete, list, search, import, setup,
login, info, install}
```

```
$ ansible-galaxy install <role name>
```

Ansible Vault

Securing a file or a string.

```
$ ansible-vault [create|decrypt|edit|encrypt|encrypt_string|rekey|view]
[options] [vaultfile.yml]
```

Common Options

`--ask-vault-pass`
ask for vault password

`--new-vault-id <NEW_VAULT_ID>`
the new vault identity to use for rekey

`--new-vault-password-file`
new vault password file for rekey

`--vault-id`
the vault identity to use

`--vault-password-file`
vault password file

`--version`
show program's version number and exit

`-h, --help`
show this help message and exit

`-v, --verbose`
verbose mode (-vvv for more, -vvvv to enable connection debugging)

To create a new encrypted key file,

`$ ansible-vault create abc.yml`(a file abc.yml is create which will have encrypted contents stored)

To encrypt existing file

```
$ ansible-vault encrypt existing.yml
```

To update or rekey files

```
$ ansible-vault rekey abc.yml
```

Editing an encrypted file

```
$ ansible-vault edit abc.yml
```

Viewing an encrypted file

```
$ ansible-vault view abc.yml
```

Decrypting files

```
$ ansible-vault decrypt abc.yml
```

To use an encrypted file while executing a playbook.

```
$ ansible-playbook –i hosts abc.yml --ask-vault-pass
```

This will prompt the user for supplying vault password.

Variable_Prompt

When running a playbook, you may wish to prompt the user for certain input, and can do so with the ‘vars_prompt’ section.
A common use for this might be for asking for sensitive data that you do not want to record.

Here’s an example…
in this example, the value for the name argument is the variable **key**. Value entered against the prompt is the **value** for the variable key.

```
---
- hosts: all
  vars_prompt:

    - name: username
      prompt: "What is your username?"
      private: no

    - name: password
      prompt: "What is your password?"

  tasks:

    - debug:
        msg: 'Logging in as {{ username }}'

    - user:
        name: “{{ username }}”
        password: “{{ password }}”
        state: present
```

Shell and Command Modules:

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

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

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

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

Or this:

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

Ignoring Failed Command or Shell task

Generally playbooks will stop executing any more steps on a host that has a task fail. Sometimes, though, you want to continue on. To do so, write a task that looks like this:

```
- name: This will not be counted as a failure
  command: ‘/bin/yum install none’
  ignore_errors: yes
```

We can also define the criteria to set for identifying failure in logs search.
You may check for failure by searching for a word or phrase in the output of a command:

```
- name: Fail task when the command error output prints FAILED
  command: /usr/bin/example-command -x -y -z
  register: command_result
  failed_when: "'FAILED' in command_result.stderr"
```

Or, based on the return code:

```
- name: Fail task when both files are identical
  raw: diff foo/file1 bar/file2
  register: diff_cmd
  failed_when: diff_cmd.rc == 0 or diff_cmd.rc >= 2
```

Conditional Statements:

When statement:

```
tasks:
 - name: “take an action on certain conditional fulfilment”
   when: ansible_facts[‘os_family’] == “RedHat”
   command: /sbin/shutdown –t now
```

or we can have conditional check in a group as statement below.

```
tasks:
  - name: "shut down CentOS 6 and Debian 7 systems"
    command: /sbin/shutdown -t now
    when: (ansible_facts['distribution'] == "CentOS" and ansible_facts['distribution_major_version'] ==
"6") or
          (ansible_facts['distribution'] == "Debian" and ansible_facts['distribution_major_version'] ==
"7")
```

Using conditional when with loop:

```
tasks:
    - command: echo {{ item }}
      loop: [ 0, 2, 4, 6, 8, 10 ]
      when: item > 5
```

Applying ‘**when’** conditional statement to roles and include.

```
- import_tasks: other_tasks.yml # note "import"
  when: x is not defined


hosts: webservers
  roles:
     - role: debian_stock_config
       when: ansible_facts['os_family'] == 'Debian'
```

Register variables: Register is default variable that can be initialized dynamically during command, module execution.

```
- name: test play
  hosts: all

  tasks:

      - shell: cat /etc/motd
        register: motd_contents

      - shell: echo "motd contains the word hi"
        when: motd_contents.stdout.find('hi') != -1
```

Or, we can validate ‘register’ variable contents for emptiness.

```
- name: check registered variable for emptiness
  hosts: all

  tasks:

      - name: list contents of directory
        command: ls mydir
        register: contents

      - name: check contents for emptiness
        debug:
          msg: "Directory is empty"
        when: contents.stdout == ""
```

Tags:

If we have large number of tasks and you want to selectively run a task or number of tasks or a play then we can make use tags to selectively execute a task or a play.

```
tasks:
- yum:
    name:
    - httpd
    - memcached
    state: present
  tags:
  - packages

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

Once we have tagged the task or play , we can call the task or play by using its tag identity.

```
$ ansible-playbook playbook.yml --tags "configuration,packages"
```
or,

```
$ ansible-playbook example.yml --skip-tags "packages"
```

Few handy commands on Ansible...

```
$ ansible-doc –t <type_of_module_from_list> --list
```

Types of docs we can list...

```
{become,cache,callback,cliconf,connection,httpapi,inventory,lookup,netco
nf,shell,module,strategy,vars}]
```

Below commands can help to list all default config values that are defined in the Ansible.cfg file.

```
$ ansible-config –version {list dump view}
```

Using inventory command to list all devices from inventory file.

```
$ ansible-inventory --list
```

Using Ansible for Windows management.

Pre-Requisites before we get started for managing Windows hosts using Ansible.

- Supported Microsoft Windows operating system versions:
  - Windows Server (2008, 2008 R2, 2012, 2012 R2, 2016, or 2019)
  - Windows 7, 8.1, or 10
- To manage a Windows-based server, Ansible must connect and run code
  - WinRM must be enabled
  - Ansible must be able to authenticate to the managed host
  - The managed host must have PowerShell 3.0 or newer and .NET Framework 4.0 or newer (Windows Server 2012 and later and Windows 8.1 and later has the right software pre-installed)

On the Ansible Control server install the python package, pywinrm using below command.

```
$ sudo yum install python-pip

$ sudo pip install pywinrm
```

Update the inventory file on the Ansible control server by adding the windows machine entry.

```
[windows]
172.31.42.206

[windows:vars]
ansible_connection=winrm
ansible_user=<username>
ansible_password=<password>
ansible_winrm_server_cert_validation=ignore
ansible_winrm_transport=basic
ansible_winrm_port=5985
```

The above variable declaration can also be done inside ‘group_vars’ folder by creating variable file named after the group name… e.g. group_vars → windows

Now, on the windows machine that we want to manage using Ansible, make few changes as show below.

```
ps> winrm quickconfig–q

ps> winrm set winrm/config/service ‘@{AllowUnencrypted=”true”}’

ps> winrm set winrm/config/service/path ‘@{Basic=”true”}’
```

Testing and Debugging Ansible Code:

1) **Syntax** Check: We can check the syntax of playbook without executing the playbook.

   ```
   $ ansible-playbook playbook_name.yml --syntax-check
   ```

   If the syntax check returns successful, then we will see only the playbook name. In case of any error in syntax, error will get displayed on screen.

2) **Dry-run** of playbook: We can dry-run a playbook to check if the results return the expected output. But in this case, the results shown are not actually implemented on target hosts.

   ```
   $ ansible-playbook --check playbook_name.yml
   ```

   We will see the logs just like a normal run logs would be, but there will not be any change done on target hosts.

   Not all modules fully support check mode. Some modules that do not support the check mode, **will not** make any changes but will not report what changes that might have been made.

   For example, the **command / shell** module doesn't support the **check** option. This is because the results of **shell or command** module can only be verified when the actual command is executed on target and in such case the results may vary from hosts to host. So in such case of check option, the task gets skipped.

   Check mode can also be extended as the tasks arguments as well. Example as below,

   ```
   tasks:
   -  name: check mode example
      yum:
         name: httpd
         state: absent
      register: pkg_results
      check_mode: yes
   ```

   In this case the playbook will be executed normally without setting the `--check_mode` option, but still above tasks will be executed in check mode only, without any results implemented on target hosts.

3) Adding **verbosity**: we can add verbosity option at the time of running a playbook with **–v** switch. We can also change the verbosity level by adding **–vv** upto **–vvvvv.** With this we can easily get very detailed output about tasks implantation logs.

   ```
   $ ansible-playbook playbook_name.yml –vv
   ```

4) **Test Modules** in Ansible:

   Debug module: The debug module prints a message or variable to the Ansible output. The debug module is useful for debugging variables or expressions.

```
$ - debug:
    msg: system {{ inventory_hostname }} has uuid {{
ansible_product_uuid }}
```

Parameter	Description
msg	A customized message to be printed. This can contain variables.
var	A variable to print. No need for {{ }}. Mutually exclusive with msg.

var and **msg** parameters cannot be used together in one task using debug module.

```
tasks:
-   name: check mode example
    command:
       cmd: '/usr/bin/date +%s'
    register: date_results

-   name: Display date value gathers by register parameter
    debug:
       var: date_results.stdout

-   name: display results with customized message
    debug:
       msg: "The system has current time as {{ date_results.stdout
    }}
```

5) **Assert Module**: The assert module checks that the given expressions are true with an optional custom message. This module helps in validating variable and conditions are what you are expecting.

Parameter	Description
that	A list of string expressions, same as **when** statement
quiet	A Boolean to avoid verbose output
success_msg	A custom message used when the asserted expression pass.
fail_msg	A custom message used when the asserted expressions fails.

```
-   name: checking assert module usage
    assert:
       that:
          variable <= 100
       fail_msg: "variable value must be between 0 to 100"
       success_msg: "variable value is less than 100"
```

6) **Fail module**: Fail module forces the Ansible playbook to fail with custom message. The fail module is useful when testing certain conditions with `when` statement.

```
- fail:
    msg: the hosts may not be the right candidate
    when: package_name == ‘apache2’
```

7) **Meta tasks**: **Meta tasks** are special tasks that can influence Ansible internal execution or state. Meta tasks can be used anywhere in the playbook. Meta is not a module thus can't be used in loop.
Useful tasks for debugging are `clear_facts` and `clear_host_errors.`

```
-   name: clear gathered facts from all targeted hosts
    meta: clear_facts

-   meta: clear_host_errors
```

8) Playbook Debugger:

- Ansible includes a debugger as part of strategy plugins.
- We can check or set value of variable, update module arguments, and rerun a task with new variables to help resolve the cause of failure.
- **The debugger** can be enabled by setting the value in ansible configuration (ansible.cfg) file or as an environment variable globally or at runtime.

```
[defaults]
enable_task_debugger = true
```

or,

```
ANSIBLE_ENABLE_TASK_DEBUGGER=True ansible-playbook –i hosts playbook.yml
```

- While running the playbook, in case of any failed or unreachable task will start the debugger unless it is disabled explicitly in playbook.
- In the playbook the debugger keyword to be defined can have below values.

Choice	Description
Always	Always invokes the debugger regardless of outco
Never	Never invoke debugger regardless of outcome
on_failed	Invoke the debugger only of the tasks fails.
On_unreachable	Invoke the debugger only if the host if unreachable.
On_skipped	Invoke the debugger only if the tasks gets skipped.

- The debugger can be enabled at **tasks level or play level** as shown below.

```
- name: execute a command
  command: false
  debugger:  on_failed
```

or,

```
- name: play-name
  hosts: all
  debugger: on_skipped
  tasks:
```

In below example we can specify debugger to be invoked at play level and also at task level. In this case the **task level debugger has higher precedence**.

```
- name: debugger example
  hosts: all
  debugger: never
  tasks:
- name: execute command
  command: some_command
  debugger: on_failed
```

Once the debugger is invoked with the related setting in Play or Task, we have following commands to use inside the debugger.

Command	Shortcut	Action
print	p	Print information about the task
task.args[*key*] = *value*	no shortcut	Update module arguments
task_vars[*key*] = *value*	no shortcut	Update task variables (you must `update_task` next)
update_task	u	Recreate a task with updated task variables
redo	r	Run the task again
continue	c	Continue executing, starting with the next task
quit	q	Quit the debugger

9) **Ansible-lint:** Validating the YAML syntax in playbook using **ansible-lint.** To install lint utility we will be using python package manager.

```
$ sudo pip install ansible-lint or.
$ sudo apt install ansible-lint
```

One can set rules for ansible-lint to verify correctness of a file or all files inside a folder (role). While using the **ansible-lint** utility, we can also set exclusions if any.
Ansible-lint configuration file can have setting about, defaults etc. sample file as shown below.

```
$ ansible-lint playbook.yaml
$ ansible-lint roles/*
```

The default ansible rules are located in `ansible-lint/lib/ansiblelint/rules`