Infrastructure automation with ANSIBLE

Chapter 1: What is Ansible

Ansible is an open source configuration management and orchestration utility. It can automate and standardize the configuration of remote hosts and virtual machines. Its orchestration functionality allows Ansible to coordinate the launch and graceful shutdown of multitiered applications. Because of this, Ansible can perform rolling updates of multiple systems in a way that results in zero downtime.

Instead of writing custom, individualized scripts, system administrators create high-level plays in Ansible. A play performs a series of tasks on the host, or group of hosts, specified in the play. A file that contains one or more plays is called a playbook.

Ansible's architecture is agentless. Work is pushed to remote hosts when Ansible executes. Modules are the programs that perform the actual work of the tasks of a play. Ansible is immediately useful because it comes with hundreds of core modules that perform useful system administrative work.

Ansible was originally written by Michael DeHaan, the creator of the Cobbler provisioning application. Ansible has been widely adopted, because it is simple to use for system administrators. Developers ease into using Ansible because it is built on Python. Ansible is supported by DevOps tools, such as Vagrant and Jenkins.

There are many things that Ansible can not do. Ansible can not audit changes made locally by other users on a system. For example, who made a change to a file? The following list provides some other examples of items that Ansible can not perform.

• Ansible can add packages to an installation, but it does not perform the initial minimal installation of the system. Every system can start with a minimal installation, either via Kickstart or a base cloud starter image, then use Ansible for further configuration.

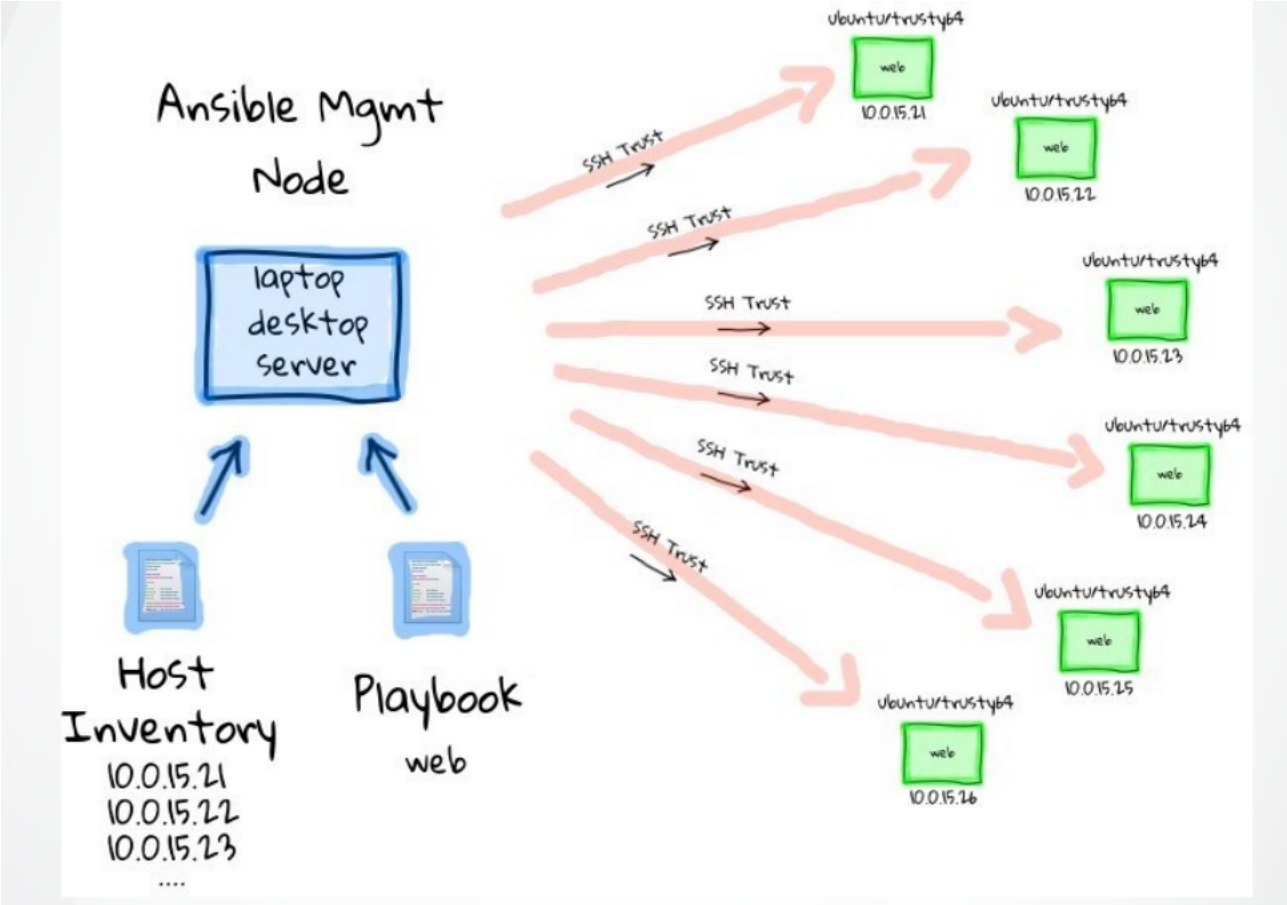
• Although Ansible can remediate configuration drift, it does not monitor for it.

• Ansible does not track what changes are made to files on the system, nor does it track what user or process made those changes. These types of changes are best tracked with a version control system or the Linux Auditing System.

Architecture

There are two types of machines in the Ansible architecture: the control node and managed hosts. Ansible software is installed on the control node and all of its components are maintained on it. The managed hosts are listed in a host inventory, a text file on the control node that includes a list of managed host names or IP addresses.

System administrators log into the control node and launch Ansible, providing it with a playbook and a target host to manage. Instead of a single system to control, a group of hosts or a wildcard can be specified. Ansible uses SSH as a network transport to communicate with the managed hosts. The modules referenced in the playbook are copied to the managed hosts. Then they are executed, in order, with the arguments specified in the playbook. Ansible users can write their own custom modules, if needed, but the core modules that come with Ansible can perform most system administration tasks.



The following lists describes the Ansible components that are maintained on the control node.

Ansible configuration
Ansible has configuration settings that defines how it behaves. These settings include such things as the remote user to execute commands, and passwords to provide when executing remote commands with sudo. Default configuration values can be overridden by environment variables or values defined in configuration files.

Host inventory
The Ansible host inventory defines which configuration groups hosts belong to. The inventory can define how Ansible communicates with a managed host and it also defines host and group variable values.

Core modules
Core modules are the modules that come bundled with Ansible, There are over 400 core modules.

Custom modules
Users can extend Ansible's functionality by writing their own modules and adding them to the Ansible library. Modules are typically written in Python, but they can be written in any interpreted programming language (shell, Ruby, Python, etc.).

Playbooks
Ansible playbooks are files written in YAML syntax that define the modules, with arguments, to apply to managed nodes. They declare the tasks that need to be performed.

Connection plugins
Plugins that enable communication with managed hosts or cloud providers. These include native SSH, paramiko SSH, and local. Paramiko is a Python implementation of OpenSSH for Red Hat Enterprise Linux 6 that provides the ControlPersist setting that improves Ansible performance.

Plugins
Extensions that enhance Ansible's functionality. Examples include e-mail notifications and logging.

Prerequiste on the Control Node

- Ansible software is installed on the control node.
- A machine acting as a control node must have Python 2.6 or 2.7 installed.
- Linux will run Ansible software.
- Windows is not supported for the control node at this time.

Prerequiste on the Managed Host

- SSH must be installed and configured to allow incoming connections.
- Managed hosts must have Python 2.4 or later installed.
- The python-simplejson package must also be installed on Red Hat Enterprise Linux 5 managed hosts. It is not required on Red Hat Enterprise Linux 6 and 7 managed hosts, since Python 2.5 (and newer versions) provide its functionality by default.

Understanding Ansible Deployment and Orchestration

One of Ansible's strengths is in how it simplifies the configuration of software on servers. When Ansible accesses managed hosts, it can discover the version of O/S running on the remote server. The installed applications and applied software subscriptions can be compared to determine if the host is properly entitled. Ansible playbooks can be used to consistently build development, test, and production servers. Kickstart can get bare-metal servers running enough to let Ansible take over and build them further. They can be provisioned to a corporate baseline standard, or they can be built for a specific role within the datacenter.

Ansible is commonly used to finish provisioning application servers. For example, a playbook can be written to perform the following steps on a newly installed base system:

1. Configure software repositories.
2. Install the application.
3. Tune configuration files. Optionally download content from a version control system.
4. Open required service ports in the firewall.

5. Start relevant services.

6. Test the application and confirm it is functioning.

Ansible is also simple tool to use for updating applications in parallel/serial for Orchestrating zero-downtime rolling upgrades

For example, a playbook can be developed to execute the following steps on application servers:

1. Stop system and application monitoring.

2. Remove the server from load balancing.

3. Stop relevant services.

4. Deploy, or update, the application.

5. Start relevant services.

6. Confirm the services are available and add the server back to load balancing.

7. Start system and application monitoring.

Connection Plugins

Control Persisit
A feature that improves Ansible performance by eliminating SSH connection overhead, when multiple SSH commands are executed in succession.

Majority of the new O/S's uses default ssh which have "ControlPersist" feature.

Paramiko
Paramiko is a python implementation of ssh with ControlPersist feature, can be used on RHEL6.

Local
This plugin is used to connect and manage "control node" itself.

Winrm
Winrm connection plugin used for managing windows machines.

Docker
Ansible 2, we have docker plugin as well, which can connect from Docker host to containers.

Chapter 2: Ansible Setup

Unlike other configuration management utilities, such as Puppet and Chef, Ansible uses an agentless architecture. Ansible itself only needs to be installed on the host or hosts from which it will be run. Hosts which will be managed by Ansible do not need to have Ansible installed. This installation involves relatively few steps and has minimal requirements.

Ansible installation on the control node requires only that Python 2, version 2.6 or later be installed. Ansible does not yet use Python 3. To see whether the appropriate version of Python is installed on ur linux system, use the yum command

```
[root@control ~]# yum list installed python
Loaded plugins: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: centos.crazyfrogs.org
 * epel: mirrors.ircam.fr
 * extras: miroir.univ-paris13.fr
 * updates: ftp.pasteur.fr
Installed Packages
python.x86_64                                2.7.5-86.el7                                @base
[root@control ~]#
```

Managed hosts do not need to have any special Ansible agent installed. They do need to have Python 2, version 2.4 or later installed. If the version of Python installed on the managed host is earlier than Python 2.5, then it must also have the python-simplejson package installed.

The Ansible control node communicates with managed hosts over the network. Multiple options are available, but an SSH connection is used by default. Ansible normally connects to the managed host by using the same user name as the one running Ansible on the control node.

To ensure security, SSH sessions require authentication at the initiation of each connection. Relying on password authentication for each connection to each managed hosts quickly becomes unwieldy when the number of managed hosts increases. Therefore, in enterprise environments, key-based authentication is a much more desirable option.

Ansible is available on EPEL Repository for majority of linux distributions, Once you have the repo, you can simply use “yum” to install ansible.

```
[root@control ~]#
[root@control ~]# yum install ansible -y
```

When Ansible has been installed, the ansible command becomes available for execution on the command line. The various options for the ansible command can be displayed with the –help option, or its abbreviated -h counterpart.

```
[root@control ~]# ansible -h
usage: ansible [-h] [--version] [-v] [-b] [--become-method BECOME_METHOD]
               [--become-user BECOME_USER] [-K] [-i INVENTORY] [--list-hosts]
               [-l SUBSET] [-P POLL_INTERVAL] [-B SECONDS] [-o] [-t TREE] [-k]
               [--private-key PRIVATE_KEY_FILE] [-u REMOTE_USER]
               [-c CONNECTION] [-T TIMEOUT]
               [--ssh-common-args SSH_COMMON_ARGS]
               [--sftp-extra-args SFTP_EXTRA_ARGS]
               [--scp-extra-args SCP_EXTRA_ARGS]
               [--ssh-extra-args SSH_EXTRA_ARGS] [-C] [--syntax-check] [-D]
               [-e EXTRA_VARS] [--vault-id VAULT_IDS]
               [--ask-vault-pass | --vault-password-file VAULT_PASSWORD_FILES]
               [-f FORKS] [-M MODULE_PATH] [--playbook-dir BASEDIR]
               [-a MODULE_ARGS] [-m MODULE_NAME]
               pattern

Define and run a single task 'playbook' against a set of hosts

positional arguments:
  pattern               host pattern

optional arguments:
```

The --version option is helpful for identifying the version of Ansible installed.

```
[root@control ~]# ansible --version
ansible 2.9.6
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/root/.ansible/plugins/modules', u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /bin/ansible
  python version = 2.7.5 (default, Aug  7 2019, 00:51:29) [GCC 4.8.5 20150623 (Red Hat 4.8.5-39)]
[root@control ~]#
```

As per the output, we can see the version of ansible installed in 2.9.6 and it also tell us about the configuration file the ansible CLI is currently reading.

The behavior of an Ansible installation can be customized by modifying settings housed in Ansible's configuration file. Ansible will select its configuration file from one of several possible locations on the control node.

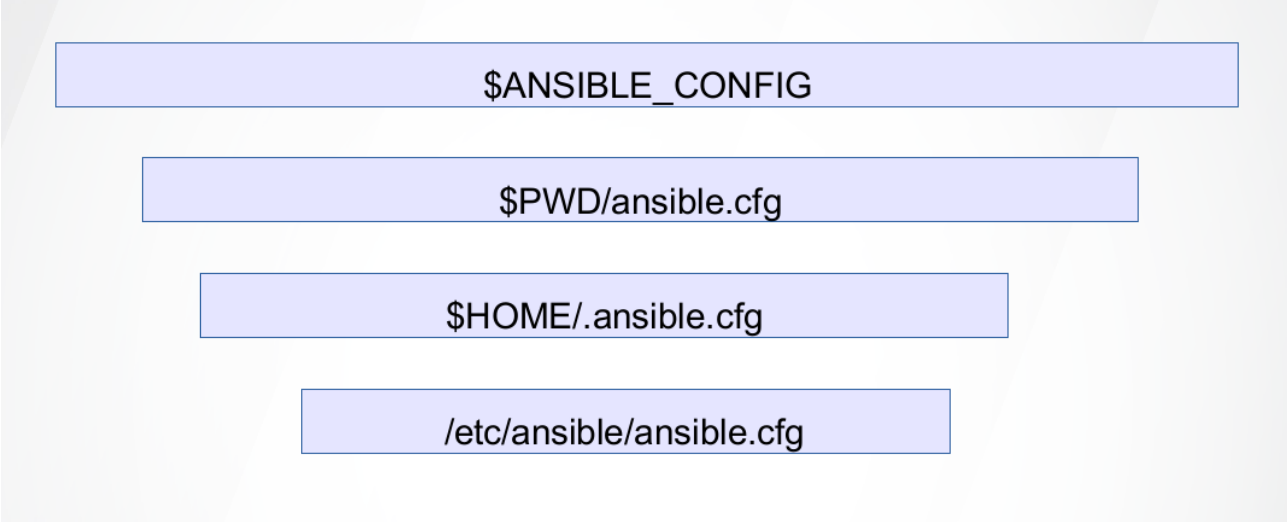
/etc/ansible/ansible.cfg
When installed, the ansible package provides a base configuration file located at /etc/ansible/ansible.cfg. This file will be used if no other configuration file is found.

$HOME/.ansible.cfg
Ansible will look for a ~/.ansible.cfg in the user's home directory. This configuration will be used instead of the /etc/ansible/ansible.cfg if it exists and if there is no ansible.cfg in the current working directory.

$PWD/ansible.cfg
If an ansible.cfg file exists in the directory in which the ansible command is executed, it is used instead of the global file or the user's personal file. This allows administrators to create a directory structure where different environments or projects are housed in separate directories with each directory containing a configuration file tailored with a unique set of settings.

$ANSIBLE_CONFIG
Users can make use of different configuration files by placing them in different directories and then executing Ansible commands from the appropriate directory, but this method can be restrictive and hard to manage as the number of configuration files grows. A more flexible option is to define the location of the configuration file with the $ANSIBLE_CONFIG environment variable. When this variable is defined, Ansible uses the configuration file the variable specifies instead of any of the previously mentioned configuration files.

The preference of the configuration files are reversed in order mentioned above.



Due to the multitude of locations where Ansible configuration files can be placed, it may be confusing which configuration file is being used by Ansible, especially when multiple files exist on the control node. To clearly identify the configuration file in use, execute the ansible command with the --version option. In addition to displaying the version of Ansible installed, it also displays the currently active configuration file.

```
[root@control ~]# ansible --version
ansible 2.9.6
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/root/.ansible/plugins/modules', u'/usr/share/ansible/plugins/modules']
```

Writing your ansible configuration file

The ansible configuration file in more like an INI format file consisting for several sections containing setttings defined as key/value pairs. Sections are enclosed with square brackets.

A sample configuration file with some settings is displayed below

```
[usa@control project]$ cat ansible.cfg
[defaults]
inventory=hosts
remote_user=india


[privilege_escalation]
become=false
become_method=sudo
become_user=root
become_ask_pass=false
[usa@control project]$
```

Inventory
A host inventory defines the path of the file which contains the details of the hosts Ansible manages. Hosts may belong to groups which are typically used to identify the hosts' role in the datacenter. A host can be a member of more than one group.

There are two ways in which host inventories can be defined. A static host inventory may be defined by a text file, or a dynamic host inventory may be generated from outside providers.

An Ansible static host inventory is defined in an INI-like text file, in which each section defines one group of hosts (a host group). Each section starts with a host group name enclosed in square brackets ([]). Then host entries for each managed host in the group are listed, each on a single line. They consist of the host names or IP addresses of managed hosts which are members of the group. In the following example, the host inventory defines three host groups, group1, group2 and group3

Sample Static host inventory file looks like this.

```
[usa@control project]$ cat hosts
[group1]
machine1.example.com
machine2.example.com

[group2]
192.168.0.10
192.168.0.20
192.168.0.30

[group3]
database.google.com
webserver.google.com

[mgroup:children]
group2
group3
[usa@control project]$
```

Ansible host inventories can include groups of host groups. This is accomplished with the :children suffix.

Now that you configured the basic ansible configuration and inventory file, it’s time to validate the configuration. The same can be done by listing the hosts in the sample inventory file.

The standard pattern of the command to list the hosts inside the inventory file looks like

 # ansible <host-pattern> -i <path of inventory file > --list-hosts

The -i flag allow us to use another inventory file which is not defined inside the configuration file. Sample examples to find the hosts using above configuration

>> To find specific host in the inventory file

```
[usa@control project]$ ansible machine1.example.com --list-hosts
  hosts (1):
    machine1.example.com
[usa@control project]$
```

>> To list all the hosts in the inventory file

```
[usa@control project]$ ansible all --list-hosts
  hosts (7):
    machine1.example.com
    machine2.example.com
    192.168.0.10
    192.168.0.20
    192.168.0.30
    database.google.com
    webserver.google.com
[usa@control project]$
```

>> To list all the hosts in one host-group available in the inventory file

```
[usa@control project]$ ansible group1 --list-hosts
  hosts (2):
    machine1.example.com
    machine2.example.com
[usa@control project]$
[usa@control project]$ ansible group3 --list-hosts
  hosts (2):
    database.google.com
    webserver.google.com
[usa@control project]$
```

>> To list all the hosts in different host-group in the same command

```
[usa@control project]$ ansible group1,group2 --list-hosts
  hosts (5):
    machine1.example.com
    machine2.example.com
    192.168.0.10
    192.168.0.20
    192.168.0.30
[usa@control project]$
```

>> To list the children group from the parent group

```
[usa@control project]$ ansible mgroup --list-hosts
  hosts (5):
    192.168.0.10
    192.168.0.20
    192.168.0.30
    database.google.com
    webserver.google.com
[usa@control project]$
```

Remote_user

When the connection-related parameters have been read, Ansible proceeds with making connections to the managed host. By default, connections to managed hosts are initiated using the SSH protocol. The SSH protocol requires the connection be established using an account on the managed host. This account is referred to by Ansible as the remote user and is defined using the remote_user setting under the [defaults] section of the Ansible configuration file.

The remote_user parameter is commented out by default in /etc/ansible/ansible.cfg. With this parameter undefined, ad hoc commands default to connecting to managed hosts using the same remote user account as that of the user account on the control node which executes the ad hoc command.

When an SSH connection has been made to a managed host, Ansible proceeds with using the specified module to perform the ad hoc operation. Once the ad hoc command is completed on a managed host, Ansible displays on the control node any standard output which was produced by the remotely executed operation.

As with all operations, the remote operation will be executed with the permissions of the user which initiated it. Because the operation is initiated with the remote user, it will be restricted by the limits of that user's permissions.

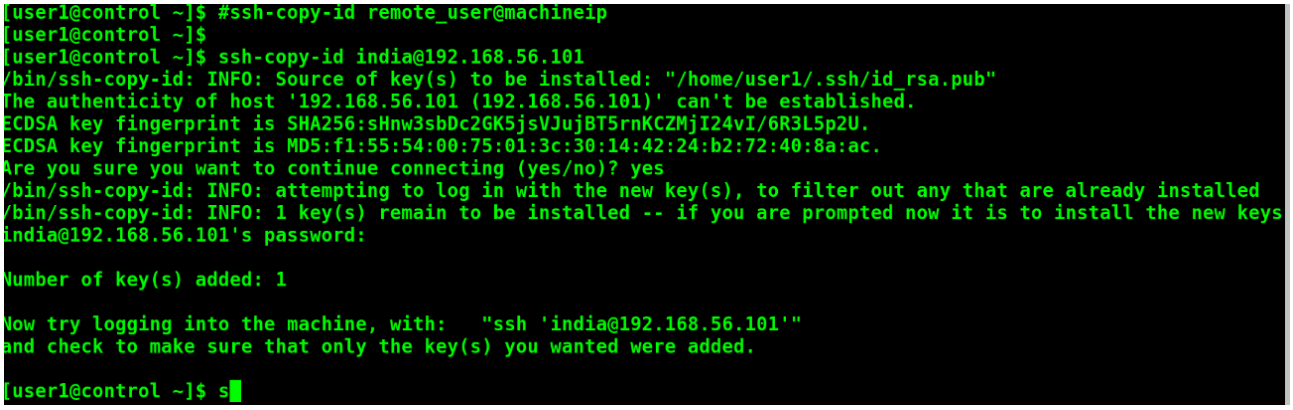
Users can authenticate ssh logins without a password by using public key authentication. Ssh allows users to authenticate using a private-public key scheme. This means that two keys are generated, a private key and a public key.

The private key file is used as the authentication credential and, like a password, must be kept secret and secure. The public key is copied to systems the user wants to log in to and is used to verify the private key. The public key does not need to be secret.

Key generation is performed using the **ssh-keygen** command. This generates the private key, ~/.ssh/id_rsa, and the public key, ~/.ssh/id_rsa.pub.

```
[user1@control ~]$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user1/.ssh/id_rsa):
Created directory '/home/user1/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/user1/.ssh/id_rsa.
Your public key has been saved in /home/user1/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:yMxuGBYd0cTJTnQztYfW3CXJ+kRBBUYRldw2mG4jsQY user1@control.example.com
The key's randomart image is:
+---[RSA 2048]----+
|      oBo.+..+&O*|
|     . .*E + O+*+|
|    . .o  . Bo+.o|
|     = ..  =.=.  |
|    o = S . oo.  |
|   . +       .   |
|    . o          |
|     .           |
|                 |
+----[SHA256]-----+
[user1@control ~]$
```

Before key-based authentication can be used, the public key needs to be copied to the destination system. This can be done with ssh-copy-id.



Privilege_escalation

After successfully connecting to a managed host as a remote user, Ansible can switch to another user on the host before executing an operation. This is done using Ansible's privilege escalation feature. For example, using the sudo command, an Ansible ad hoc command can be executed on a managed host with root privileges even if the SSH connection to the managed host was authenticated by a non-privileged remote user.

Configuration settings to enable privilege escalation are located under the [privilege_escalation] section of the ansible.cfg configuration file.

When privilege escalation is enabled, the become_method, become_user, and become_ask_pass parameters come into play. This is true even if these parameters are commented out in /etc/ansible/ansible.cfg since they are predefined internally within Ansible.

- ***become=true (Config file) or --become or -b (CLI)*** option is to activate or deactive the privilege escalation for operations on managed hosts
- ***become_user or --become-user*** option defines the user with desired privileges – the user will be come, not the user you login as i.e. remote user
- ***become_method or --become-method*** option defines the method to escalate the privilege escalation using different become plugins like sudo, su , enable, runas etc
  you can check the become plugins option on "https://docs.ansible.com/ansible/latest/plugins/become.html#become-plugins"
- ***become_ask_pass or –ask-become-pass or -K*** option defines whether the privilege escalation on the managed host should prompt for a password or not.

Ad-hoc Commands

Before you start running the ad-hoc commands, couple of things needs to be taken care of. These settings are already discussed in the previous section.

1) Remote User should be planned.
2) SSH-Key based authentication (for password less auth)
3) Either Privileges has to be given to the user or not.
4) If yes, privilege escalation will be passwordless ?

Ansible allows administrators to execute on-demand tasks on managed hosts. These ad hoc commands are the most basic operations that can be performed with Ansible.

Ad hoc commands are performed by running ansible on the control node, specifying as part of the command which operation should be performed on the managed hosts. Each ad hoc command is capable of performing a single operation. If multiple operations are required on the managed hosts, then an administrator needs to execute a series of ad hoc commands on the control node.

The ad hoc command is an easy way for administrators to get started with using Ansible so they can get a better idea of its use and application in a real world setting. It also serves as an introduction to more advanced Ansible features such as modules, tasks, plays, and playbooks.

Sample syntax
 # ansible host-pattern -m module_name -a arguments -i inventory

In the example below, we are first lisiting the hosts available for us inside our inventory file and then using one basic module “PING” to check the connectivity between the two boxes.

```
[usa@control project]$ ansible all --list-hosts
  hosts (1):
    192.168.56.101
[usa@control project]$ ansible all -m ping
192.168.56.101 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
[usa@control project]$
```

The module specified by the -m option defines which Ansible module to use in order to perform the remote operation. Modules will be discussed in a later section. For now, think of a module as a tool which is designed to accomplish a specific task.

Arguments are passed to the specified modules using the -a option. Some modules are capable of accepting no arguments; other modules can accept multiple arguments. When no argument is needed, omit the -a option from the ad hoc command. If multiple arguments need to be specified, supply them as a quoted space-separated list.

Using the command module to run linux shell command on the managed host. The command module allows administrators to quickly execute remote commands on managed hosts. These commands are not processed by the shell on the managed hosts. As such, they cannot access shell environment variables or perform shell operations such as redirection and piping

>> Using command module to execute some commands on the managed host

```
[usa@control project]$ ansible all -m command -a 'whoami'
192.168.56.101 | CHANGED | rc=0 >>
india
[usa@control project]$ ansible all -m command -a 'hostname'
192.168.56.101 | CHANGED | rc=0 >>
host1.example.com
[usa@control project]$ ansible all -m command -a 'ls -l /tmp'
192.168.56.101 | CHANGED | rc=0 >>
total 8
-rw-r--r--. 1 root  root   31 Apr  1 16:03 abc
drwx------. 2 india india  41 Apr  2 02:00 ansible_command_payload__Goz5J
-rwx------. 1 root  root  836 Apr  1 11:16 ks-script-5gauq0
drwx------. 3 root  root   17 Apr  1 11:48 systemd-private-5deb9a2f71b140afa0196eab60fca4fc-chronyd.service-oQIG
tg
drwx------. 3 root  root   17 Apr  1 16:15 systemd-private-5deb9a2f71b140afa0196eab60fca4fc-httpd.service-KvQe2P
-rw-------. 1 root  root    0 Apr  1 11:10 yum.log
[usa@control project]$
```

>> Using command module to see the privilege escalation in action with –become option. As you can see in the snippet below, ansible gives us an error we are trying to get the output of ls -l /root command on the managed host without using "***--become or -b***" option. Once you use that option, it automatically reads the confirguation files and escalate the privilege according to the configured setttings in ansible.cfg file.

```
[usa@control project]$ ansible all -m command -a 'ls -l /root'
192.168.56.101 | FAILED | rc=2 >>
ls: cannot open directory /root: Permission deniednon-zero return code
[usa@control project]$
[usa@control project]$ ansible all -m command -a 'ls -l /root' -b
192.168.56.101 | CHANGED | rc=0 >>
total 4
-rw-------. 1 root root 1261 Apr  1 11:16 anaconda-ks.cfg
[usa@control project]$
[usa@control project]$ ansible all -m command -a 'whoami' -b
192.168.56.101 | CHANGED | rc=0 >>
root
[usa@control project]$
```

Chapter 3: Writing your first playbook

Playbooks are the files where Ansible code is written. Playbooks are written in YAML format. YAML stands for Yet Another Markup Language. Playbooks are one of the core features of Ansible and tell Ansible what to execute. They are like a to-do list for Ansible that contains a list of tasks.

Playbooks contain the steps which the user wants to execute on a particular machine. Playbooks are run sequentially. Playbooks are the building blocks for all the use cases of Ansible.

YAML was designed primarily for the representation of data structures such as lists and associative arrays in an easy to write, human-readable format. This design objective is accomplished primarily by abandoning traditional enclosure syntax, such as brackets, braces, or opening and closing tags, commonly used by other languages to denote the structure of a data hierarchy. Instead, in YAML, data hierarchy structures are maintained using outline indentation.

Indentation can only be performed using the space character. Indentation is very critical to the proper interpretation of YAML. Since tabs are treated differently by various editors and tools, YAML forbids the use of tabs for indentation.

Playbook Structure

Each playbook is an aggregation of one or more plays in it. Playbooks are structured using Plays. There can be more than one play inside a playbook.

The function of a play is to map a set of instructions defined against a particular host.

YAML is a strict typed language; so, extra care needs to be taken while writing the YAML files. There are different YAML editors but we will prefer to use a simple editor like vim which some settings which will make our work easy.

Before you begin, Create a .vimrc file with following settings and then source the file

> set ai (auto indentation)
> set ts=2 (setting up the tap to move 2 spaces rather then using default 8 spaces)
> set cursorcolumn (to run the vertical line highlight column in your vim file)

```
[usa@control project]$ cat /home/usa/.vimrc
set ai
set ts=2
set cursorcolumn
[usa@control project]$ source /home/usa/.vimrc
[usa@control project]$
```

For Ansible, nearly every YAML file starts with a list. Each item in the list is a list of key/value pairs, commonly called a “hash” or a “dictionary”. So, we need to know how to write lists and dictionaries in YAML.

There’s another small quirk to YAML. All YAML files (regardless of their association with Ansible or not) can optionally begin with `---` and end with `...`. This is part of the YAML format and indicates the start and end of a document.

- **List**
  In YAML, lists are like arrays in other programming languagesAll members of a list are lines beginning at the same indentation level starting with a `"- "` (a dash and a space):

- **Dictionaries**
  dictionary is represented in a simple `key: value` form (the colon must be followed by a space):

- More complicated data structures are possible, such as lists of dictionaries, dictionaries whose values are lists or a mix of both:

Understanding modules

Modules are programs that Ansible uses to perform operations on managed hosts. They are ready-to-use tools designed to perform specific operations. Modules can be executed from the ansible command line or used in playbooks to execute tasks. When run, modules are copied to the managed host and executed there.

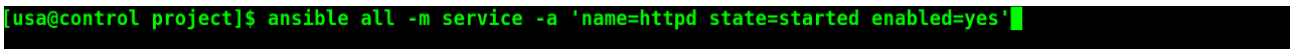
Ansible documentation consists of entire module index available.
https://docs.ansible.com/ansible/latest/modules/modules_by_category.html

Command to see the modules already available in your system is
ansible-doc -l
ansible-doc -l | grep module_name
ansible-doc module_name

Modules can be invoked as part of an ad hoc command, using the ansible command. The -m option allows administrators to specify the name of the module to use.

Let's try to use the ansible module in the adhoc command with some argurments
Following example shows us to use the 'YUM' module to install the httpd package in the managed host. We are using "-m" to define the module name to be used in the ad hoc command and "-a" to pass the arguments required for this module to operate. We are passing two argument 'name and state' with the module. To get the complete list of arguments and their uses, please check "ansible-doc modulename".

Another example of using "SERVICE" module to start the httpd daemon on the managed host



Modules can also be called in playbooks, as part of a task. The following snippet shows how the yum module can be invoked with the name of a package and its desired state as arguments.

Creation of Playbook

Let us start by writing a sample YAML file. We will walk through each section written in a yaml file.

```
[usa@control project]$ cat first.yml
- name:  my first playbook
  hosts: all
  become: yes
  vars:
    var1: vishal
  tasks:
    - name: install the package
      yum:
        name: httpd
        state: present

    - name: start the service
      service:
        name: httpd
        state: started
        enabled: yes

    - name: install another software
      yum:
        name: mysql
        state: present

[usa@control project]$
```

Let us now go through the different YAML tags. The different tags are described below −

- **name**
  This tag specifies the name of the Ansible playbook. As in what this playbook will be doing. Any logical name can be given to the playbook.

- **hosts**
  This tag specifies the lists of hosts or host group against which we want to run the task. The hosts field/tag is mandatory. It tells Ansible on which hosts to run the listed tasks. The tasks can be run on the same machine or on a remote machine. One can run the tasks on multiple machines and hence hosts tag can have a group of hosts' entry as well.

- **vars**
  Vars tag lets you define the variables which you can use in your playbook. Usage is similar to variables in any programming language.

- **tasks**
  All playbooks should contain tasks or a list of tasks to be executed. Tasks are a list of actions one needs to perform. A tasks field contains the name of the task. This works as the help text for the user. It is not mandatory but proves useful in debugging the playbook. Each task internally links to a piece of code called a module. A module that should be executed, and arguments that are required for the module you want to execute.

Ansible playbooks should be written so that tasks do not make unnecessary changes if the managed hosts are already in the correct state. In other words, if a playbook is run once to put the hosts in the correct state, the playbook should be written so it is safe to run it a second time and it should make no further changes to the configured systems. A playbook with this property is **idempotent**. Most Ansible modules are idempotent, so it is relatively easy to ensure this is true.

Execution of Playbooks

Playbooks are executed using the ansible-playbook command. The command is executed on the control node and the name of the playbook to be executed is passed as an argument.

When the playbook is executed, output is generated to show the play and tasks being executed. The output also reports the results of each task executed.

The following example shows the execution of a simple playbook. It shows the tasks are executed on the managed host but there was no changes done as the managed host was already on the desired state.

```
[usa@control project]$ ansible-playbook first.yml

PLAY [my first playbook] *******************************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [192.168.56.101]

TASK [install the package] *****************************************************************************
ok: [192.168.56.101]

TASK [start the service] *******************************************************************************
ok: [192.168.56.101]

TASK [install another software] ************************************************************************
ok: [192.168.56.101]

PLAY RECAP *********************************************************************************************
192.168.56.101             : ok=4    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored
=0

[usa@control project]$
```

To verify the syntax of the playbook, you can use the following command

```
[usa@control project]$ ansible-playbook --syntax-check first.yml

playbook: first.yml
[usa@control project]$
```

To do a dry run on the managed host, which will report what changes would have occurred if the playbook were executed, but does not make any actual changes to managed hosts.

```
[usa@control project]$ ansible-playbook -C first.yml

PLAY [my first playbook] *******************************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [192.168.56.101]

TASK [install the package] *****************************************************************************
ok: [192.168.56.101]

TASK [start the service] *******************************************************************************
ok: [192.168.56.101]

TASK [install another software] ************************************************************************
ok: [192.168.56.101]

PLAY RECAP *********************************************************************************************
192.168.56.101             : ok=4    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored
=0

[usa@control project]$
```

When developing new playbooks, it may be helpful to execute the playbook interactively. The ansible-playbook command offers the --step option for this purpose.

Playbook exercise

In this exercise, you will write and use the ansible playbook to perform some administration tasks on a managed host

Create a playbook named chap3.yml in your control node which should do the following

> Should install the httpd and firewalld package
> Should start the httpd and firewalld service
> Should allow the http traffic from the firewall
> Create the test index.html page inside *"/var/www/html"* directory.

Execute the playbook and test the scenario by accessing the http server on the managed host from the control node using curl or elinks command.

Chapter 4: Ansible Variables

Variable in playbooks are very similar to using variables in any programming language. It helps you to use and assign a value to a variable and use that anywhere in the playbook. One can put conditions around the value of the variables and accordingly use them in the playbook. This can help simplify creation and maintenance of a project and reduce the incidence of errors.

Variables provide a convenient way to manage dynamic values for a given environment in your Ansible project. Some examples of values that variables might contain include

> Package to be install
> User to create
> Files to remove or create
> Services to be managed etc.

There are standard's to be followed while declaring the variable inside the playbook like the naming convention.

- The variable name should not contain any space.
- It cann't contain any special character apart from "_".
- The variable name cann't start with an integer.

Variable Scope's
All many programming language, the variables as well can be declared or defined in many places and may have different scope level. We have three standard scopes to deal with.

Global scope: Variables set from the command line or Ansible configuration

Play scope: Variables set in the play and related structures

Host scope: Variables set on host groups and individual hosts by the inventory, fact gathering, or registered tasks

If the same variable name is defined at more than one level, the higher wins. So variables defined by the inventory are overridden by variables defined by the playbook, which are overridden by variables defined on the command line.

Scenario #1

Let's try declaring a variable in the host inventory file and call it inside your playbook using the debug module

```
[usa@control project]$ cat datafile
192.168.56.101  var1=program
[usa@control project]$ cat datavar.yml
- name: using vars in playbook
  hosts: all
  tasks:
    - name: calling variable
      debug:
        msg: hi there {{ var1 }}     # the variables are called with {{ }} braces

    - name: calling variable 2
      debug:
        msg: "{{ var1 }}"    # Remember when a variable is used as first element,quotes are mandatory"
[usa@control project]$
```

Now to execute this playbook, use the ansible-playbook command and see it's output.

```
[usa@control project]$ ansible-playbook datavar.yml

PLAY [using vars in playbook] **************************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [192.168.56.101]

TASK [calling variable] ********************************************************************************
ok: [192.168.56.101] => {
    "msg": "hi there program"
}

TASK [calling variable 2] ******************************************************************************
ok: [192.168.56.101] => {
    "msg": "program"
}

PLAY RECAP *********************************************************************************************
192.168.56.101             : ok=3    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored
=0

[usa@control project]$
```

Scenario #2

Let's see how to declare the variable in the playbook and use it. In the below yaml you are able to see that one variable named var1 is declared and it's value is been called upon by the debug module which is used to print it's value. Create the playbook and execute it

```
[usa@control project]$ cat var.yml
- name: using vars in playbook
  hosts: all
  vars:
    var1: data    # declaration of variable in the playbook
  tasks:
    - name: calling variable
      debug:
        msg: hi there {{ var1 }}     # the variables are called with {{ }} braces

    - name: calling variable 2
      debug:
        msg: "{{ var1 }}"    # Remember when a variable is used as first element,quotes are mandatory"
[usa@control project]$
```

```
[usa@control project]$ ansible-playbook var.yml

PLAY [using vars in playbook] ***************************************************************************

TASK [Gathering Facts] **********************************************************************************
ok: [192.168.56.101]

TASK [calling variable] *********************************************************************************
ok: [192.168.56.101] => {
    "msg": "hi there data"
}

TASK [calling variable 2] *******************************************************************************
ok: [192.168.56.101] => {
    "msg": "data"
}

PLAY RECAP **********************************************************************************************
192.168.56.101             : ok=3    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored
=0

[usa@control project]$
```

As you can see from the output, the value of the variable defined inside inventory has been overridden by the value in the playbook.

Using Register Variable

Administrators can capture the output of a command by using the register statement. The output is saved into a variable that could be used later for either debugging purposes or in order to achieve something else, such as a particular configuration based on a command's output.

Demostration

The following playbook demonstrates how to capture the output of a command for debugging purposes:

```
[usa@control project]$ cat reg.yaml
- name: using register
  hosts: all
  become: yes
  tasks:
    - name: installing software
      yum:
        name: httpd
        state: present
      register: myreg

    - name: printing the debug
      debug:
        var: myreg
[usa@control project]$



[usa@control project]$ ansible-playbook reg.yaml

PLAY [using register] ***********************************************************************************

TASK [Gathering Facts] **********************************************************************************
ok: [192.168.56.101]

TASK [installing software] ******************************************************************************
ok: [192.168.56.101]

TASK [printing the debug] *******************************************************************************
ok: [192.168.56.101] => {
    "myreg": {
        "changed": false,
        "failed": false,
        "msg": "",
        "rc": 0,
        "results": [
            "httpd-2.4.6-90.el7.centos.x86_64 providing httpd is already installed"
        ]
    }
}

PLAY RECAP **********************************************************************************************
192.168.56.101             : ok=3    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored
=0

[usa@control project]$
```

Facts

Ansible facts are variables that are automatically discovered by Ansible from a managed host. Facts are pulled by the setup module and contain useful information stored into variables that administrators can reuse. Ansible facts can be part of playbooks, in conditionals, loops, or any other dynamic statement that depends on a value for a managed host; for example:

- A server can be restarted depending on the current kernel version.
- The MySQL configuration file can be customized depending on the available memory.
- Users can be created depending on the host name.

There are virtually no limits to how Ansible facts can be used, since they can be part of blocks, loops, conditionals, and so on. Ansible facts are a convenient way to retrieve the state of a managed node and decide which action to take based on its state. Facts provide information about

- The host name
- The kernel version
- The network interfaces
- The IP addresses
- The version of the operating system
- Various environment variables
- The number of CPUs
- The available or free memory
- The available disk space

You can use the “SETUP” module to see facts ansible gathered from a managed node.

```
[usa@control project]$ ansible all -m setup
192.168.56.101 | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "10.0.2.15",
            "192.168.56.101"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::db2d:a5b1:5ddf:a943",
            "fe80::1474:220a:1d4f:68f5"
        ],
        "ansible_apparmor": {
            "status": "disabled"
        },
        "ansible_architecture": "x86_64",
        "ansible_bios_date": "12/01/2006",
        "ansible_bios_version": "VirtualBox",
        "ansible_cmdline": {
            "BOOT_IMAGE": "/vmlinuz-3.10.0-862.el7.x86_64",
            "LANG": "en_US.UTF-8",
            "crashkernel": "auto",
            "quiet": true,
            "rd.lvm.lv": "centos/swap",
```

To filter the output in the setup module, we can use ansible filters in order to limit the results returned when gathering facts from managed node.

```
[usa@control project]$ ansible all -m setup -a 'filter=ansible_hostname'
192.168.56.101 | SUCCESS => {
    "ansible_facts": {
        "ansible_hostname": "host1",
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false
}
[usa@control project]$ ansible all -m setup -a 'filter=ansible_eth0'
192.168.56.101 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false
}
[usa@control project]$ ansible all -m setup -a 'filter=ansible_enp0s3'
192.168.56.101 | SUCCESS => {
    "ansible_facts": {
        "ansible_enp0s3": {
            "active": true,
            "device": "enp0s3",
            "features": {
                "busy_poll": "off [fixed]",
                "fcoe_mtu": "off [fixed]",
                "generic_receive_offload": "on",
                "generic_segmentation_offload": "on",
                "highdma": "off [fixed]",
                "hw-tc-offload": "off [fixed]"
```

Using the facts in the playbook and running it

```
[usa@control project]$ cat facts.yml
- name: calling facts in playbook
  hosts: all
  tasks:
    - name: print the hostname
      debug:
        msg: the machine hostname is {{ ansible_hostname }}
[usa@control project]$
[usa@control project]$ ansible-playbook facts.yml

PLAY [calling facts in playbook] ***********************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [192.168.56.101]

TASK [print the hostname] ******************************************************************************
ok: [192.168.56.101] => {
    "msg": "the machine hostname is host1"
}

PLAY RECAP *********************************************************************************************
192.168.56.101             : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored
=0

[usa@control project]$
```

Inclusions

When working with complex or long playbooks, administrators can use separate files to divide tasks and lists of variables into smaller pieces for easier management. There are multiple ways to include task files and variables in a playbook.

Tasks can be included in a playbook from an external file using include directive:

```
tasks:
- name: Include file from another task
  include: file.yml
~
~
```

```
[usa@control project]$ cat test.yml
- name: print the value for variable
  debug:
    msg: "{{ pkg1 }}"

[usa@control project]$  cat call.yml
- name: using include task
  hosts: all
  tasks:
    - name: calling the task file
      include: test.yml
      vars:
        pkg1: new-package
[usa@control project]$
```

Let’s see the execution of the playbook

```
[usa@control project]$ ansible-playbook call.yml

PLAY [using include task] ******************************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [192.168.56.101]

TASK [print the value for variable] ********************************************************************
ok: [192.168.56.101] => {
    "msg": "new-package"
}

PLAY RECAP *********************************************************************************************
192.168.56.101             : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored
=0
```

The include_vars module can include variables defined in either JSON or YAML files, overriding host variables and playbook variables already defined.

```
tasks:
- name: Include the variables from a YAML or JSON file
  include_vars: vars/variables.yml
```

```
pkg1: new-package
~
~
~
~
~
~
~
~
~
~
~
```

```
- name: calling the variables
  hosts: all
  tasks:
    - name: include the va
      include_vars: var.yml

    - name: printing it
      debug:
        msg: "{{ pkg1 }}"
~
~
~
```

Understanding Task control in Ansible

Ansible can use conditionals to execute tasks or plays when certain conditions are met. For example, a conditional can be used to determine the available memory on a managed host before Ansible installs or configures a service.

Conditionals allow administrators to differentiate between managed hosts and assign them functional roles based on the conditions that they meet. Playbook variables, registered variables, and Ansible facts can all be tested with conditionals. Operators such as string comparison, mathematical operators, and Booleans are available. Task control feature allow us to run or execute the task inside the playbook with some conditional check. There are many methods to perform the task control in ansible which includes:

> Using the when Condition

> Using Notify and Handler

> Using Tags

> Using Block and Rescue for error handling

When Syntax
The when statement is used to run a task conditionally. It takes as a value the condition to test. If the condition is met, the task runs. If the condition is not met, the task is skipped.

One of the simplest conditions which can be tested is whether a Boolean variable is true or false. The when statement in the following example causes the task to run only if v is true.
Booleans can be used as a simple switch to enable or disable tasks.

```
[usa@control project]$ cat abc.yml
- name: asd
  hosts: all
  vars:
    v: true
  tasks:
    - debug:
        msg: hello
      when: v
[usa@control project]$ ansible-playbook abc.yml

PLAY [asd] *********************************************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [192.168.56.101]

TASK [debug] *******************************************************************************************
ok: [192.168.56.101] => {
    "msg": "hello"
}

PLAY RECAP *********************************************************************************************
192.168.56.101             : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored
=0
```

You can try changing the value of v from true to false and then run the playbook you will see the task will be skipped automatically by ansible.

The next example is a bit more sophisticated, and tests whether the var1 variable has a value. If it does, the value of var1 is used as the name of the package to install. If the var1 variable is not defined, then the task is skipped without an error.

```
[usa@control project]$ cat when1.yml
- name: when2
  hosts: all
  become: yes
  vars:
    var1: httpd
  tasks:
    - name: install the package
      yum:
        name: "{{ var1 }}"
      when: var1 is defined


[usa@control project]$
```

If you run this playbook right now, it will execute the task and the httpd package will be installed on the managed node. But if you remove the var1 from the playbook, let's see the output

```
[usa@control project]$ cat when1.yml
- name: when2
  hosts: all
  become: yes
  tasks:
    - name: install the package
      yum:
        name: "{{ var1 }}"
      when: var1 is defined


[usa@control project]$ ansible-playbook when1.yml

PLAY [when2] ***************************************************************************************************

TASK [Gathering Facts] *****************************************************************************************
ok: [192.168.56.101]

TASK [install the package] *************************************************************************************
skipping: [192.168.56.101]

PLAY RECAP *****************************************************************************************************
192.168.56.101             : ok=1    changed=0    unreachable=0    failed=0    skipped=1    rescued=0    ignored
=0

[usa@control project]$
```

Testing Multiple Conditions

One when statement can be used to evaluate multiple values. To do so, conditionals can be combined with the and and or keywords or grouped with parentheses.

The following snippets show some examples of how to express multiple conditions.

```
when: ansible_kernel == 3.10.0-327.el7.x86_64 and inventory_hostname in groups['staging']

when: ansible_distribution == "RedHat" or ansible_distribution == "Fedora"

when: (ansible_distribution == "RedHat" and ansible_distribution_major_version == 7) or
(ansible_distribution == "Fedora" and ansible_distribution_major_version == 23)
~
```

Notify and Handlers

Ansible modules are designed to be idempotent. This means that in a properly written playbook, the playbook and its tasks can be run multiple times without changing the managed host, unless they need to make a change in order to get the managed host to the desired state.

However, sometimes when a task does make a change to the system, a further task may need to be run. For example, a change to a service's configuration file may then require that the service be reloaded so that the changed configuration takes effect.

Handlers are tasks that respond to a notification triggered by other tasks. Each handler has a globally-unique name, and is triggered at the end of a block of tasks in a playbook. If no task notifies the handler by name, it will not run. If one or more tasks notify the handler, it will run exactly once after all other tasks in the play have completed. Because handlers are tasks, administrators can use the same modules in handlers that they would for any other task. Normally, handlers are used to reboot hosts and restart services.

Handlers can be seen as inactive tasks that only get triggered when explicitly invoked using a notify statement.

Sample playbook with handler and notify statement

```
[usa@control project]$ cat handle.yml
- name: using handler
  hosts: all
  become: yes
  tasks:
    - name: install pkg
      yum:
        name: httpd
        state: present
      notify:
        - hand1

  handlers:
    - name: hand1
      debug:
        msg: pkg is installed
[usa@control project]$
```

Let’s see the execution of the playbook, As you will see the task notifying the handler doesn’t change the state of the managed host and it remained OK, so the handler doesn’t execute. Handler only execute when the state of the host changed.

```
[usa@control project]$ ansible-playbook handle.yml

PLAY [using handler] ***********************************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [192.168.56.101]

TASK [install pkg] *************************************************************************************
ok: [192.168.56.101]

PLAY RECAP *********************************************************************************************
192.168.56.101             : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored
=0

[usa@control project]$
```

Tag's In Ansible

For long playbooks, it is useful to be able to run subsets of the tasks in the playbook. To do this, tags can be set on specific resources as a text label. Tagging a resource only requires that the tags keyword be used, followed by a list of tags to apply. When plays are tagged, the –tags option can be used with ansible-playbook to filter the playbook to only execute specific tagged plays.

```
[usa@control project]$ cat tag.yml
- name: using tags
  hosts: all
  tasks:
    - name: task1
      debug:
        msg: task with tags
      tags:
       - tag1

    - name: task2
      debug:
        msg: task without tags

[usa@control project]$
```

You can execute the tasks with –tags flag to see only the task which has the tag got executed and the other task was not even considered.

```
[usa@control project]$ ansible-playbook --tag 'tag1' tag.yml

PLAY [using tags] **************************************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [192.168.56.101]

TASK [task1] *******************************************************************************************
ok: [192.168.56.101] => {
    "msg": "task with tags"
}

PLAY RECAP *********************************************************************************************
192.168.56.101             : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored
=0
```

Use the –skip-tag flag to run the tasks which doesn't have the mentioned tag.

```
[usa@control project]$ ansible-playbook --skip-tag 'tag1' tag.yml

PLAY [using tags] **************************************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [192.168.56.101]

TASK [task2] *******************************************************************************************
ok: [192.168.56.101] => {
    "msg": "task without tags"
}

PLAY RECAP *********************************************************************************************
192.168.56.101             : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored
=0

[usa@control project]$
```

Handling Error's

Ansible evaluates the return code of each task to determine whether the task succeeded or failed. Normally, when a task fails Ansible immediately aborts the rest of the play on that host, skipping all subsequent tasks.

However, sometimes you may want to have play execution continue even if a task fails. For example, you might expect that a particular task could fail, and a you might want to recover by running some other task conditionally. There are a number of Ansible features that can be used to manage task errors.

Ignoring Task Failure

By default, if a task fails, the play is aborted. However, this behavior can be overridden by ignoring failed tasks. To do so, the ignore_errors keyword needs to be used in a task.

The following snippet shows how to use ignore_errors on a task to continue playbook execution on the host even if the task fails. For example, if the notapkg package does not exist the yum module will fail, but having ignore_errors set to yes will allow execution to continue.

```
- yum:
    name: notapkg
    state: latest
  ignore_errors: yes
```

Ansible Blocks and Error Handling

In playbooks, blocks are clauses that logically group tasks, and can be used to control how tasks are executed.

Blocks also allow for error handling in combination with the rescue and always statements. If any task in a block fails, tasks in its rescue block are executed in order to recover. After the tasks in the block and possibly the rescue run, then tasks in its always block run. To summarize:

• block: Defines the main tasks to run.

• rescue: Defines the tasks that will be run if the tasks defined in the block clause fails.

• always: Defines the tasks that will always run independently of the success or failure of tasks defined in the block and rescue clauses.

The following example shows how to implement a block in a playbook. Even if tasks defined in the block clause fail, tasks defined in the rescue and always clauses will be executed.

```
[usa@control project]$ cat blk.yml
- name: using blockresuce
  hosts: all
  become: yes
  tasks:
    - block:
        - name: installing the pkg
          yum:
            name: new-pkg
            state: installed
      rescue:
        - name: installing another service
          yum:
            name: httpd
            state: installed

      always:
        - name: print
          debug:
            msg: this will always print the info
[usa@control project]$
```

As you can see, the block section contains unknown package name, so when it fails the rescue task will automatically get initiated and always will run irrespective to the success or failure of BLOCK and RESCUE tasks.

```
[usa@control project]$ ansible-playbook blk.yml

PLAY [using blockresuce] *******************************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [192.168.56.101]

TASK [installing the pkg] ******************************************************************************
fatal: [192.168.56.101]: FAILED! => {"changed": false, "msg": "No package matching 'new-pkg' found available, in
stalled or updated", "rc": 126, "results": ["No package matching 'new-pkg' found available, installed or updated
"]}

TASK [installing another service] **********************************************************************
ok: [192.168.56.101]

TASK [print] *******************************************************************************************
ok: [192.168.56.101] => {
    "msg": "this will always print the info"
}

PLAY RECAP *********************************************************************************************
192.168.56.101             : ok=3    changed=0    unreachable=0    failed=0    skipped=0    rescued=1    ignored
=0

[usa@control project]$
```

Chapter 5: File Handling and Jinja2

The Files modules library includes modules allowing you to accomplish most tasks related to Linux file management, such as creating, copying, editing, and modifying permissions and other attributes of files.

Some of the common file modules which are frequently used are

blockinfile
This module will insert/update/remove a block of multi-line text surrounded by customizable marker lines.

Copy
Copy a file from the local or remote machine to a location on a managed host. Similar to the file module, the copy module can also set file attributes, including SELinux context.

Fetch
This module works like the copy module, but in reverse. This module is used for fetching files from remote machines to the control node and storing them in a file tree, organized by host name.

File
Set attributes such as permissions, ownership, SELinux contexts, and time stamps of regular files, symlinks, hard links, and directories. This module can also create or remove regular files, symlinks, hard links, and directories.

Lineinfile
Ensure that a particular line is in a file, or replace an existing line using a back-reference regular expression. This module is primarily useful when you want to change a single line in a file.

Jinja2 Template
Ansible uses the Jinja2 templating system for template files. Ansible also uses Jinja2 syntax to reference variables in playbooks, so you already know a little bit about how to use it.

Chapter 6: Ansible Roles

Roles provide a framework for fully independent, or interdependent collections of variables, tasks, files, templates, and modules.

In Ansible, the role is the primary mechanism for breaking a playbook into multiple files. This simplifies writing **complex playbooks**, and it makes them easier to reuse. The breaking of playbook allows you to logically break the playbook into reusable components.

Each role is basically limited to a particular functionality or desired output, with all the necessary steps to provide that result either within that role itself or in other roles listed as dependencies.

Roles are not playbooks. Roles are small functionality which can be independently used but have to be used within playbooks. There is no way to directly execute a role. Roles have no explicit setting for which host the role will apply to.

Top-level playbooks are the bridge holding the hosts from your inventory file to roles that should be applied to those hosts.

Roles provide Ansible with a way to load tasks, handlers, and variables from external files. Static files and templates can also be associated and referenced by a role. The files that define a role have specific names and are organized in a rigid directory structure, which will be discussed later.

Roles can be written so they are general purpose and can be reused.

Use of Ansible roles has the following benefits:

• Roles group content, allowing easy sharing of code with others

• Roles can be written that define the essential elements of a system type: web server, database

server, git repository, or other purpose

• Roles make larger projects more manageable

• Roles can be developed in parallel by different administrators

Creating a New Role

The directory structure for roles is essential to create a new role.

Role Structure

Roles have a structured layout on the file system. The default structure can be changed but for now let us stick to defaults.

Each role is a directory tree in itself. The role name is the directory name within the /roles directory

Now to initilize the role directory

```
[usa@control project]$ ansible-galaxy init role1
- Role role1 was created successfully
[usa@control project]$ ls role1
defaults  files  handlers  meta  README.md  tasks  templates  tests  vars
[usa@control project]$ tree role1
role1
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml

8 directories, 8 files
[usa@control project]$
```

The decription of the directories are as per the picture below:

Subdirectory	Function
defaults	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.

Defining the role content

After the directory structure is created, the content of the Ansible role must be defined. A good place to start would be the ROLENAME/tasks/main.yml file. This file defines which modules to call on the managed hosts that this role is applied.

The following tasks/main.yml file manages the /etc/motd file on managed hosts. It uses the template module to copy the template named motd.j2 to the managed host. The template is retrieved from the templates subdirectory of the role. The template itself contains some variable defined under the vars subdirectory of the role.

```
[usa@control project]$
[usa@control project]$ cat role1/tasks/main.yml
- name: copy the motd file
  template:
    src:  templates/motd.j2
    dest:  /etc/motd
    owner: root
    group: root
    mode: 0444
[usa@control project]$
[usa@control project]$
[usa@control project]$ cat role1/templates/motd.j2
Welcome to {{ ansible_hostname }}
If you need access please connect to {{ vname }} on his id {{ email }}
[usa@control project]$
[usa@control project]$
[usa@control project]$ cat role1/vars/main.yml q
vname: admin
email: admin@localhost
```

Using role in playbook

To access a role, reference it in the roles: section of a playbook. The following playbook refers to the role1 role. Because no variables are specified, the role will be applied with its default variable values.

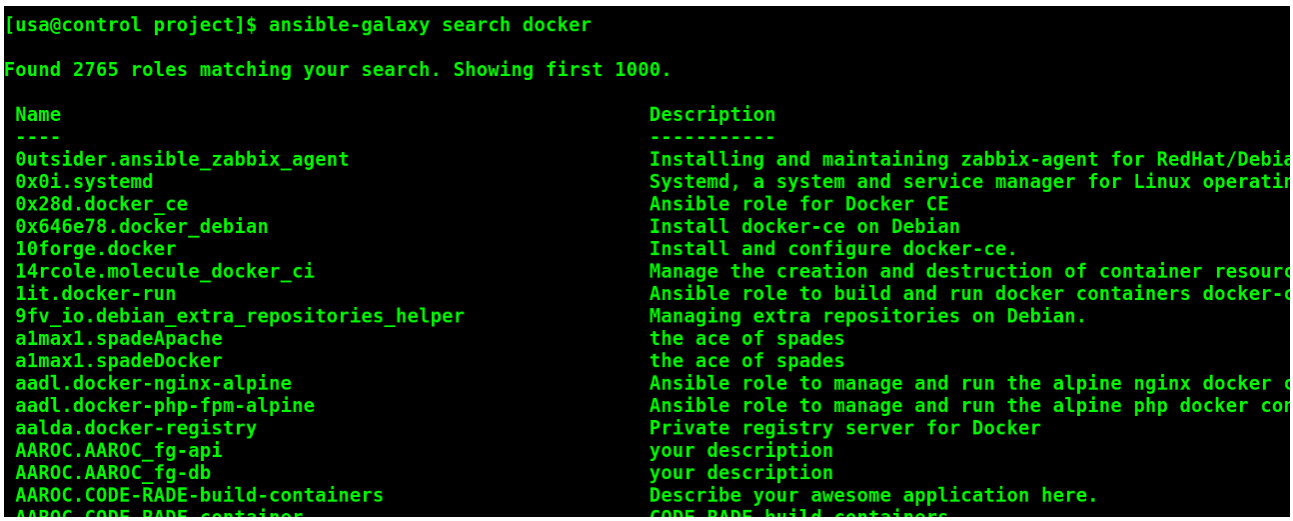
```
[usa@control project]$ vim using_role.yml
[usa@control project]$ cat using_role.yml
- name: using role
  hosts: all
  roles:
    - role1

[usa@control project]$
```

Ansible Galaxy

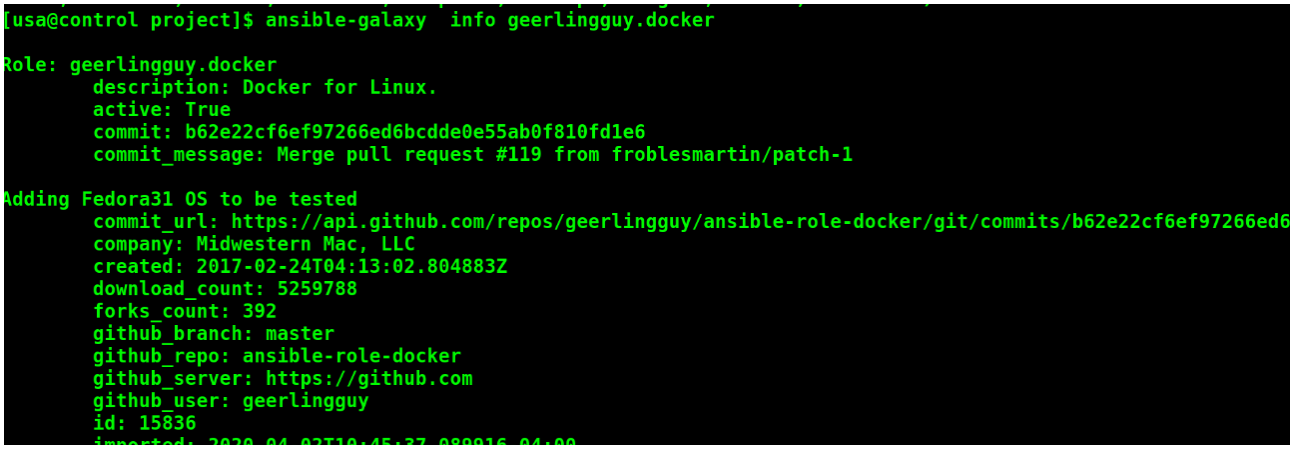
Ansible Galaxy [https://galaxy.ansible.com] is a public library of Ansible roles written by a variety of Ansible administrators and users. It is an archive that contains thousands of Ansible roles and it has a searchable database that helps Ansible users identify roles that might help them accomplish an administrative task. Ansible Galaxy includes links to documentation and videos for new Ansible users and role developers

The ansible-galaxy command line tool can be used to search for, display information about, install, list, remove, or initialize roles.

The ansible-galaxy search subcommand searches Ansible Galaxy for the string specified as an argument. The --author, --platforms, and --galaxy-tags options can be used to narrow the search results.



The ansible-galaxy info subcommand displays more detailed information about a role. The following command displays information about the geerlingguy.docker role, available from Ansible Galaxy. Because the information requires more than one screen to display, ansible-galaxy uses less to display the role's information



The ansible-galaxy install subcommand downloads a role from Ansible Galaxy, then installs it locally on the control node. The default installation location for roles is /etc/ ansible/roles. This location can be overridden by either the value of the role_path configuration variable, or a -p DIRECTORY option on the command-line.

```
[usa@control project]$ ansible-galaxy  install geerlingguy.docker -p roles
- downloading role 'docker', owned by geerlingguy
- downloading role from https://github.com/geerlingguy/ansible-role-docker/archive/2.7.0.tar.gz
- extracting geerlingguy.docker to /home/usa/project/roles/geerlingguy.docker
- geerlingguy.docker (2.7.0) was installed successfully
[usa@control project]$
```

The ansible-galaxy command can manage local roles. The roles are found in the roles directory of the current project, or they can be found in one of the directories listed in the roles_path variable. The ansible-galaxy list subcommand lists the roles that are found locally.

```
[usa@control project]$ ansible-galaxy list
# /home/usa/project/roles
- geerlingguy.docker, 2.7.0
[usa@control project]$
```

A role can be removed locally with the ansible-galaxy remove subcommand

Chapter 7: Ansible Vault

Ansible may need access to sensitive data such as passwords or API keys in order to configure remote servers. Normally, this information might be stored as plain text in inventory variables or other Ansible files. But in that case, any user with access to the Ansible files or a version control system which stores the Ansible files would have access to this sensitive data. This poses an obvious security risk.

There are two primary ways to store this data more securely:

• Use Ansible Vault, which is included with Ansible and can encrypt and decrypt any structured data file used by Ansible.

• Use a third-party key management service to store the data in the cloud, such as Vault by HashiCorp, Amazon's AWS Key Management Service, or Microsoft Azure Key Vault.

To use Ansible Vault, a command line tool called ansible-vault is used to create, edit, encrypt, decrypt, and view files. Ansible Vault can encrypt any structured data file used by Ansible. This might include inventory variables, included variable files in a playbook, variable files passed as an argument when executing the playbook, or variables defined in Ansible roles

To create a new encrypted file use the command ansible-vault create filename. The command will prompt for the new vault password and open a file using the default editor.

```
[usa@control project]$
[usa@control project]$ ansible-vault create abc.yml
```

Instead of entering the vault password through standard input, a vault password file can be used to store the vault password. This file will need to be carefully protected through file permissions and other means.

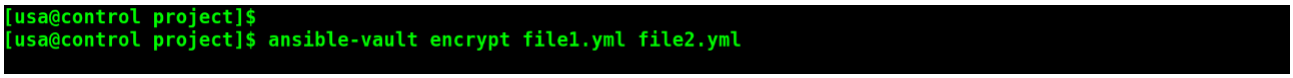
```
[usa@control project]$
[usa@control project]$ ansible-vault create  --vault-password-file=pass abc.yml
```

To edit an existing encrypted file, Ansible Vault provides the command ansible-vault edit filename. This command will decrypt the file to a temporary file and allows you to edit the file. When saved, it copies the content and removes the temporary file.

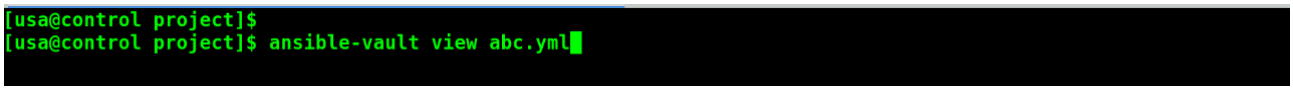
```
[usa@control project]$
[usa@control project]$ ansible-vault edit abc.yml
```

The vault password can be changed using the command ansible-vault rekey filename. This command can rekey multiple data files at once. It will ask for the original password and the new password

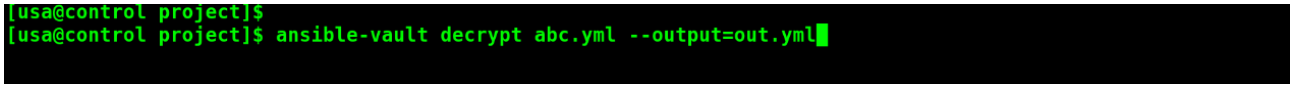
To encrypt a file that already exists, use the command ansible-vault encrypt filename. This command can take the names of multiple files to be encrypted as arguments.



Ansible Vault allows you to view the encrypted file using the command ansible-vault view filename, without opening it for editing.



An already existing encrypted file can be permanently decrypted by using the command ansible-vault decrypt filename. The --output option can be used when decrypting a single file to save the decrypted file under a different name.



Chapter 8: Ansible Tower / Ansible AWX

AWX Project or AWX is a web based console for Ansible. **AWX** is designed to make **Ansible** more usable for DevOps members of different technical proficiencies and skill sets. **AWX** is a hub for automation tasks. **Ansible Tower** is a commercial product supported by **Red Hat, Inc.** but derived from **AWX** upstream project, which is open source since September 2017. This makes **AWX** a good alternative of **Ansible Tower**.

Currently, stable version of **AWX 7.0** has been released that requires a containerized environment to install and run. According to AWX Project Documentation, we can deploy **AWX** on following containerization platforms.

1. OpenShift
2. Kubernetes
3. Docker-Compose

We are using **Docker-Compose** here, because it is relatively easy to setup and requires less hardware resources as compare to OpenShift and Kubernetes.

We have already installed Ansible control node on CentOS 7. Now, we are installing **AWX** with **Docker-Compose** on the same **Ansible** control node.