IT Infrastructure Automation using Ansible

Guidelines to Automate the Network, Windows, Linux, and Cloud Administration

WAQAS IRTAZA

bpb

IT Infrastructure Automation Using Ansible

Guidelines to Automate the Network, Windows, Linux, and Cloud Administration

Waqas Irtaza

www.bpbonline.com

FIRST EDITION 2022

Copyright © BPB Publications, India

ISBN: 978-93-91030-032

Dedicated to

NAZMA AND ZIMAL

My wife and daughter, for letting me
sacrificing family time in peruse of my passion

About the Author

Waqas Irtaza is an experienced IT professional, who for the past 11 years working cross infrastructure domains. He started his career with system administration, then move to network, wireless and cloud administration. Later in his career he start using his diverse experience in Infrastructure automation. He is certified from almost all technology leaders Cisco, Microsoft, AWS, PMI, EC-Council and Linux Foundation.

He has started his automation journey with small python and shell scripts. He managed to write complex script later in his career to automate the day to day repetitive work for cross IT infrastructures.

Ansible was a pleasant surprise for him, since complex IT task can be managed with simple Ansible playbooks. This simplicity and diversity of Ansible encourage him to write this book.

Outside work, Waqas like cycling and horse riding in addition to help, coach, and mentor young people in taking up their careers in technology.

About the Reviewer

Sumit Jaiswal has 10 years of industry experience in the development domain with core development technologies like Python, C++, and C#. He's currently working as a Senior software engineer under the Ansible automation team and working on developing content for the networking and security domain. He's an enthusiast of open-source projects.

Acknowledgement

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank my wife and daughter, Nazma and Zimal for putting up with me while I was spending many weekends and evenings on writing—I could have never completed this book without their support.

This book wouldn't have happened if I hadn't had the support from my line Manager Mr. John Daniel and Manager Technology Command Centre Mr. Manoj Panicker. My gratitude goes to my team Technology Command Centre (TCC) at The Emirates Group, especially Mr. Ahmed Humayun Zaheer, for providing valuable insights and expert opinions.

Finally, I would like to thank BPB Publications for giving me this opportunity to write my first book for them.

Preface

Ansible is a powerful open source automation language. Uniquely, it's also a deployment and orchestration tool. While Ansible provides more productive drop-in replacements for many core capabilities in other automation solutions, it also seeks to solve other major unsolved IT challenges.

Ansible is an open-source automation tool, or platform, used for IT tasks such as configuration management, application deployment, service orchestration, and infrastructure provisioning. Automation is crucial these days, with IT environments that are complex and often need to scale quickly for system, network and cloud administrators and developers to keep up if they had to do everything manually. Automation simplifies complex infrastructure tasks, not just making developers' jobs more manageable but allowing them to focus attention on other tasks that add value to an organization. In other words, it frees up time and increases efficiency. And Ansible, as noted above, is rapidly rising to the top in the world of automation tools.

When I start learning Ansible and gone through couple of books I have noticed that I have to toggle here and there to clear Ansible concepts. Therefore, I thought to come up with a book which covers all aspects of infrastructure administration. This book begin with Ansible basics so anyone with no automation experience can start from there. Once basics are cleared each chapter is specifically designed for respective technology. Over the course of 7 chapters in this book, you will learn the following:

Chapter 1 Lab setup and introduction to ansible and core concepts

Chapter 2 Understand Ansible Ad-Hoc and playbook with examples. This chapter also covers the core concepts and practical implementations.

Chapter 3 This chapter covers the Ansible advance concepts which will help reader writing professional playbooks.

Chapter 4 is a key chapter for network administration. This chapter will discuss one liner Ad-Hoc commands which are handy for network administrators. Later this chapter discuss advance Ansible concepts specifically for network administration

Chapter 5 is also a key chapter for system administration which covers both Linux and Windows Infrastructures for automation.

Chapter 6 is a key chapter for DevOps and cloud Administrators. It covers Ansible automation for the public and private clouds in addition to Docker and Kubernates.

Chapter 7 is the final chapter that discusses recommendation and best practices for Ansible.

Table of Contents

CHAPTER 1

Up and Running with Ansible

Ansible is a simple open-source engine which automates the application deployment, infrastructure service orchestration, cloud provisioning, and many other IT tools. Ansible is completely agentless, which means that it works by connecting your nodes through SSH. Most of the current technologies use SSH as a default administration tool, whether it is a public or a private cloud platform, networking equipment, or a locally hosted ESX, Windows, or Linux servers. Henceforth, Ansible has a diverse scope.

Moreover, if you want the other methods for a remote connection like those options are also available. So, let's begin our journey and start exploring what to expect from this chapter.

Structure

In this chapter, we will cover the following topics:

Introduction to Ansible

Lab setup

Important concepts about Ansible

Basic understanding of YAML

Objective

After studying this chapter, you should be able to do the following:

Understand why Ansible is popular.

Install Ansible and set up your environment.

Understand the Ansible components.

Understand the YAML syntax.

Introduction to Ansible

If you are a Systems Administrator, Network Administrator, or just anybody working in the IT, you were probably involved in doing a lot of repetitive tasks in your environment, whether it was to configure thousands of routers and switches, create new virtual machines, apply configurations, patch bulk of servers, migrations, deploy applications, or even perform security and compliance audits.

All of these repetitive tasks involve execution of similar commands on thousands of different devices, while maintaining the right sequence of events. Smart people develop scripts to automate these tasks, but it requires coding skills; regular IT administrators don't have the professional coding skills, and an IT developer with good coding skills don't have the strong infrastructure knowledge. Hence, there is a gap which Ansible leverages.

Ansible is a powerful IT automation tool that you can quickly learn. It's simple enough for everyone in IT, yet powerful enough to automate even the most complex deployments.

The following are some products related to Ansible:

Ansible An open-source automation platform.

Ansible A website with a large catalogue of community created roles.

Ansible A web application and REST API solution for Ansible.

The main focus of the book is the Ansible core. The following are some of the Ansible use cases:

Configuration management

Deployment

Orchestration

Provisioning

Let's explore the use cases with examples. Imagine you have a number of servers in your environment that you would like to patch and restart in a particular order. Some of them are application servers and the others are database servers. So, first you would patch the servers, and power down the application servers, followed by the database servers, then power up the database servers, and once the database is online, then power up the application servers. You could write an Ansible playbook to get this done in a matter of minutes and simply invoke the Ansible playbook every time you wish to perform this patching activity.

Similarly, if you have to enable the port security on all the switches in the network, which is a repetitive work across the network, and if this needs to be performed on thousands of switches, a simple Ansible playbook can achieve this task.

Let's take a look at a complex example. Let’s suppose, we are setting up a complex infrastructure that spans across public and private cloud and hundreds of VMs. With Ansible, you could provision the hosts on public clouds like Azure or AWS and provision private cloud like OpenStack or in VMware based infrastructure and move onto configuring the applications on those systems and set up communications between them. There are a lot of built-in modules available in Ansible that supports these kinds of operations and many more. We will be exploring them in the following chapters.

The following block diagram will clarify the Ansible key components:

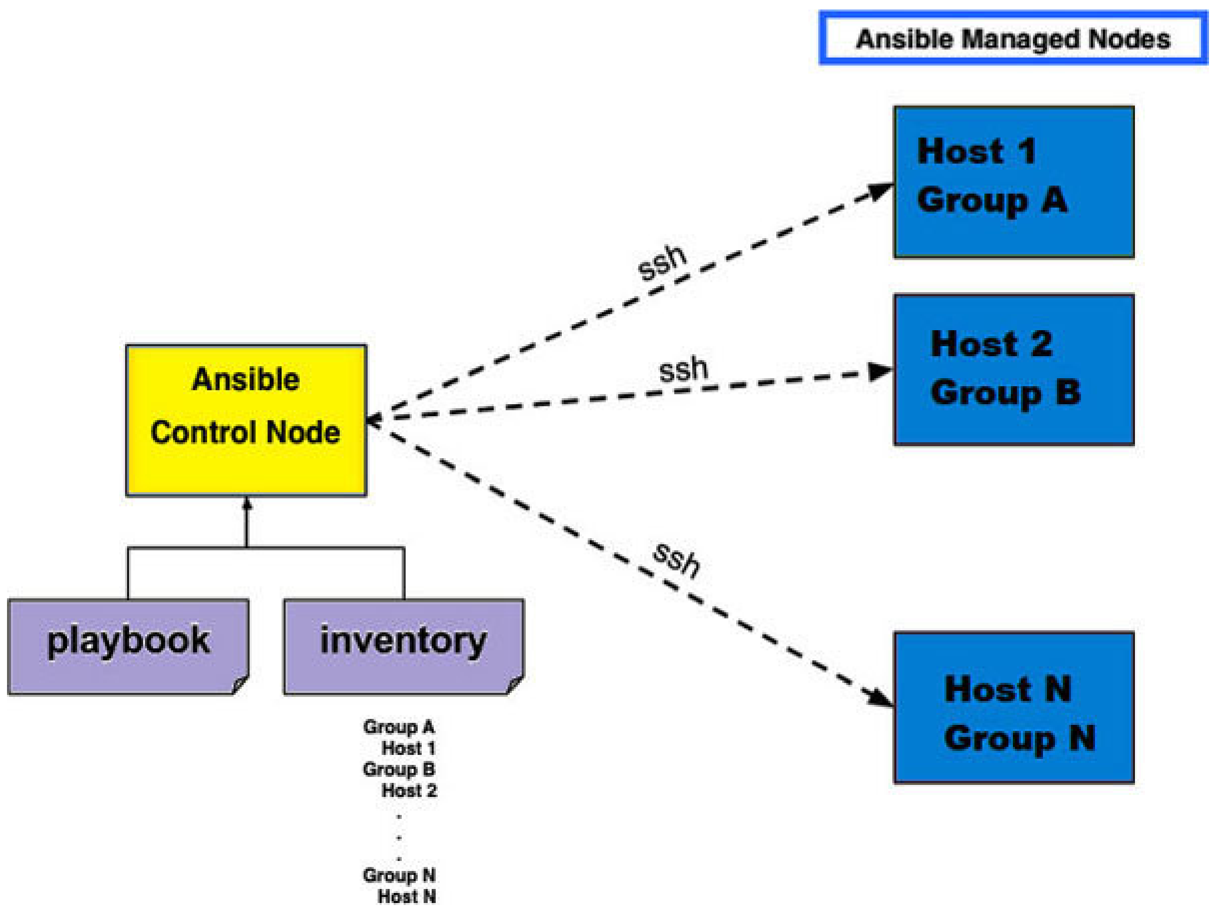


Figure 1.1: *Ansible key components*

To understand better, the following is the definition of each Ansible component:

Ansible Control Ansible Control Node is any machine which has Ansible installed. It must be a Linux machine; it can't be any Windows machine. This is the one machine where the installation is required.

Ansible Managed Ansible Managed Nodes are servers or any IT devices, which we want to manage through Ansible. Ansible Managed Nodes are also called Ansible installation is not required on hosts.

Inventory is a list of hosts. We can organize the hosts by nesting, and creating groups for scaling. Static and dynamic inventories are available, which we will discuss later.

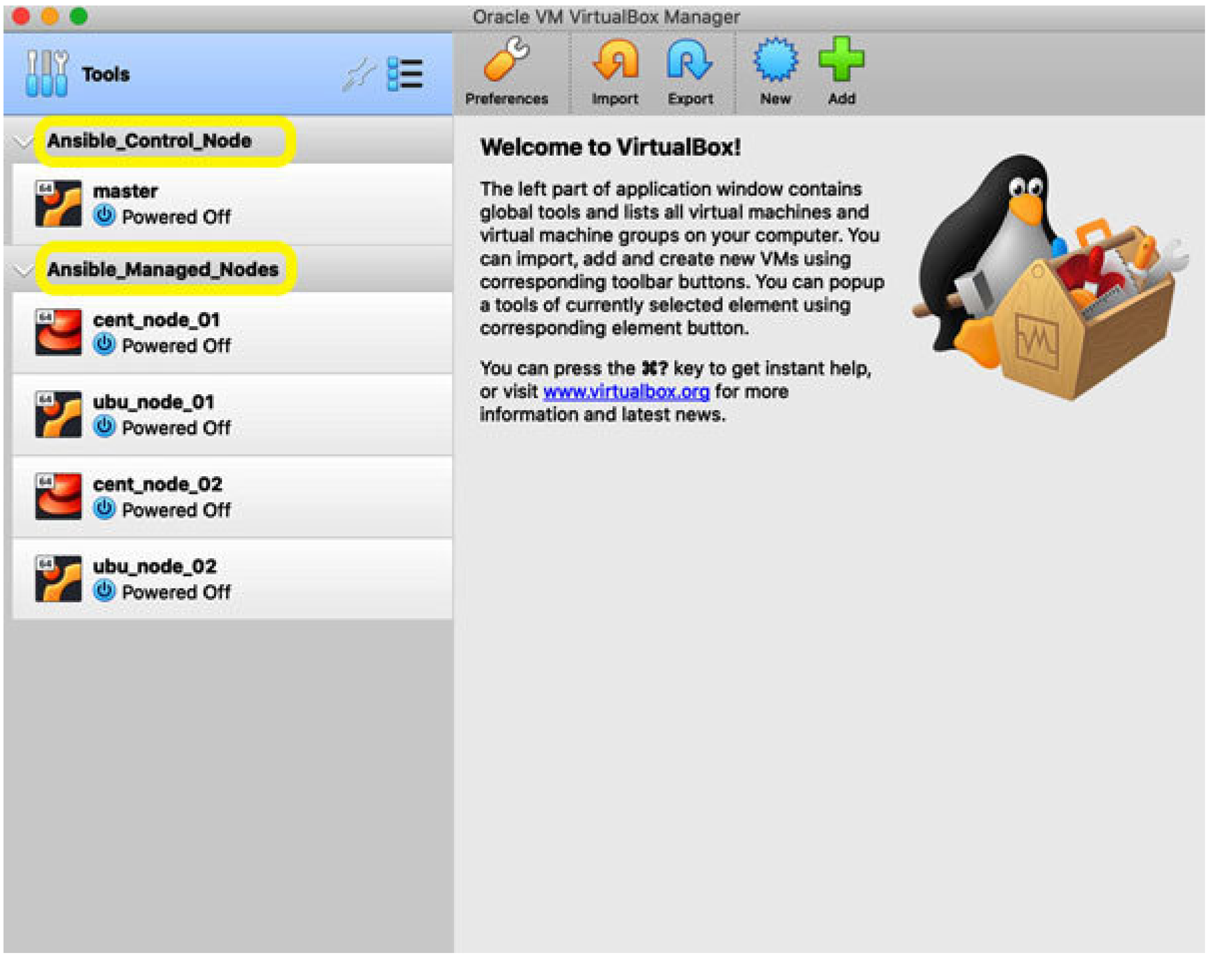
A Playbook is an ordered list of saved tasks, which can run repeatedly. Playbooks gives a programmable flavor to Ansible by allowing variable, loops, and conditional statements. Playbooks are written in YAML format.

Lab setup

Ansible installation is extremely simple; just a couple of commands and you have a fully functional Ansible control node. However, our goal is not just to install Ansible, but to understand the infrastructure requirement and have a fully functional lab to follow along the practical examples. Let's prepare the lab by following these processes.

Infrastructure preparation for Ansible

We will be setting up an entire lab environment on our laptop from scratch. For this, we will use a virtualization tool called **Oracle** since it is open source and works with all the flavors of operating systems like Windows, Linux, and OSX. We will first install the Oracle VirtualBox on our laptop and then install the CentOS or Ubuntu operating systems; I have created one Ansible control node on Ubuntu and four managed nodes, two on CentOS and two on Ubuntu. The following screenshot shows the VMs on VirtualBox:



Ansible_Control_Node
master
Powered Off
Ansible_Managed_Nodes
cent_node_01
Powered Off
ubu_node_01
Powered Off
cent_node_02
Powered Off
ubu_node_02
Powered Off

Welcome to VirtualBox!

The left part of application window contains global tools and lists all virtual machines and virtual machine groups on your computer. You can import, add and create new VMs using corresponding toolbar buttons. You can popup a tools of currently selected element using corresponding element button.

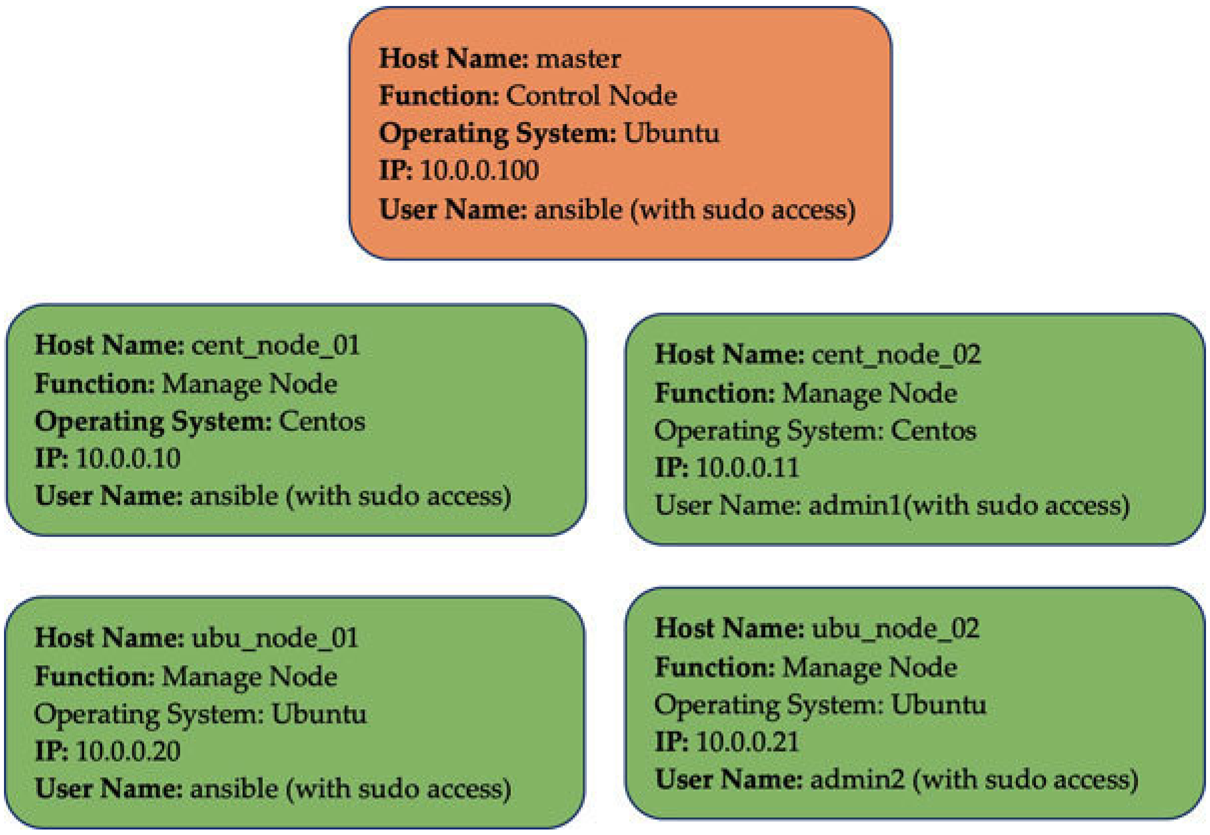
You can press the ⌘? key to get instant help, or visit www.virtualbox.org for more information and latest news.

Figure 1.2: *VMs on VirtualBox*

Having four managed nodes is not required if you have a low specification laptop, a single Linux host can also be used as both the control node and the managed nodes.

We will not cover the CentOS/Ubuntu installation, since it is out of our scope and a lot of YouTube videos are available to achieve it. We will cover the Ansible installation after the operating system installation.

The following is the basic configuration on all the hosts before starting the Ansible installation:



Host Name: master
Function: Control Node
Operating System: Ubuntu
IP: 10.0.0.100
User Name: ansible (with sudo access)

Host Name: cent_node_01
Function: Manage Node
Operating System: Centos
IP: 10.0.0.10
User Name: ansible (with sudo access)

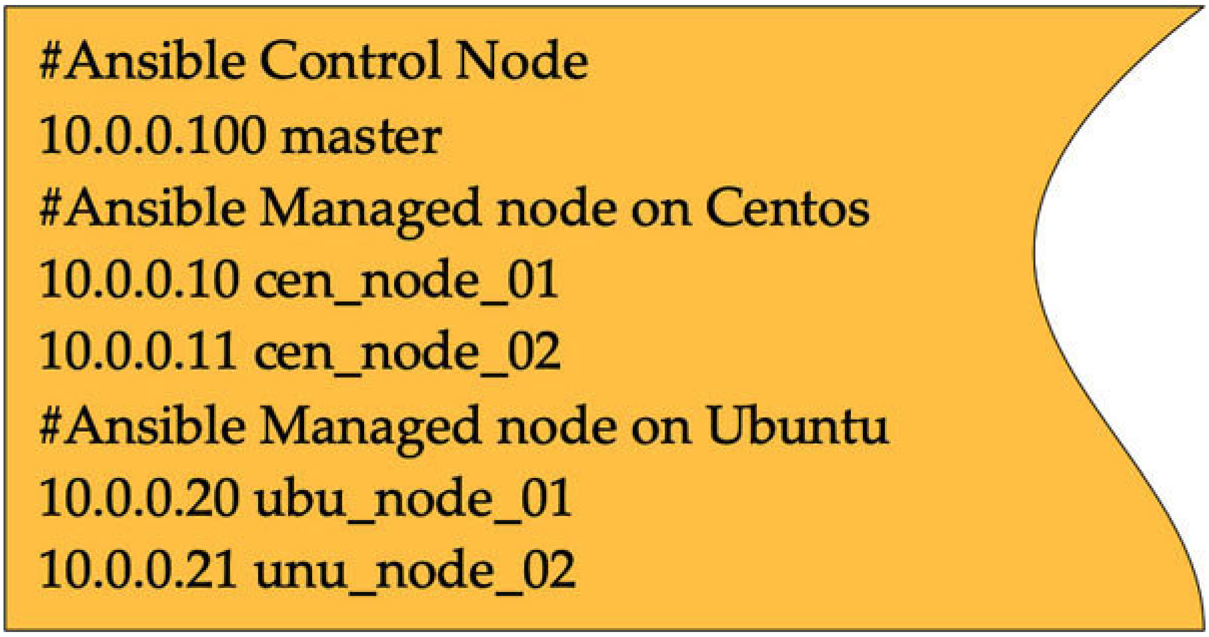
Host Name: cent_node_02
Function: Manage Node
Operating System: Centos
IP: 10.0.0.11
User Name: admin1(with sudo access)

Host Name: ubu_node_01
Function: Manage Node
Operating System: Ubuntu
IP: 10.0.0.20
User Name: ansible (with sudo access)

Host Name: ubu_node_02
Function: Manage Node
Operating System: Ubuntu
IP: 10.0.0.21
User Name: admin2 (with sudo access)

__Figure 1.3:__ Basic configuration

Ansible works fine with IP address but it doesn't look professional. Therefore, we will resolve the IP address to host name, which is not compulsory. We can copy the following content in the **/etc/hosts** file in all the Linux systems to resolve the host names to their respective IP addresses:



```
#Ansible Control Node
10.0.0.100 master
#Ansible Managed node on Centos
10.0.0.10 cen_node_01
10.0.0.11 cen_node_02
#Ansible Managed node on Ubuntu
10.0.0.20 ubu_node_01
10.0.0.21 unu_node_02
```

__Figure 1.4:__ Host file for name resolution

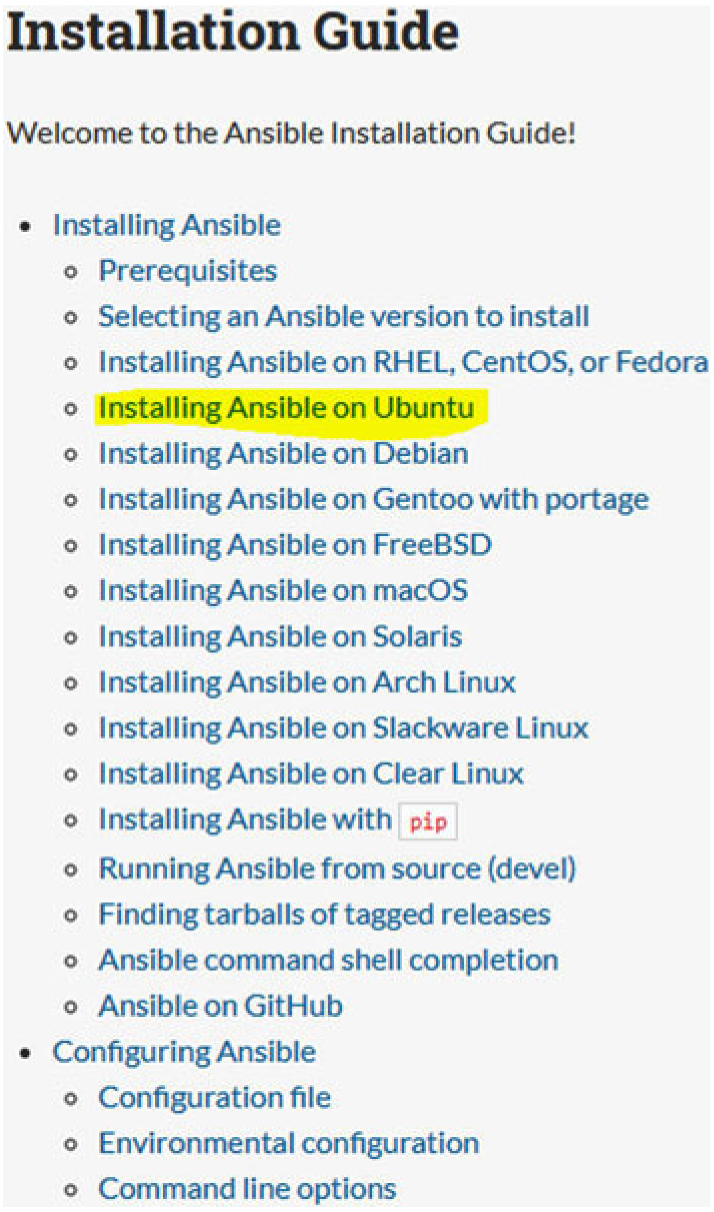
Once copied, we will be able to ping all the servers by their respective host names. At this point, our infrastructure is ready to install the Ansible.

Ansible installation

In this section, we will cover the Ansible installation on the Ansible control node and also see all possible options that are available for the Ansible managed nodes. Let's begin.

Control node setup

The Ansible installation is only required on control node. Ansible has a very simple installation process; just a few commands and you have a fully functional Ansible control node. All you need to do is go to the Ansible documentation and complete the provided steps. The following is a screenshot taken from the Ansible official documentation, which has a diverse range of installation options against each operating system:



Installation Guide

Welcome to the Ansible Installation Guide!

- Installing Ansible
  - Prerequisites
  - Selecting an Ansible version to install
  - Installing Ansible on RHEL, CentOS, or Fedora
  - Installing Ansible on Ubuntu
  - Installing Ansible on Debian
  - Installing Ansible on Gentoo with portage
  - Installing Ansible on FreeBSD
  - Installing Ansible on macOS
  - Installing Ansible on Solaris
  - Installing Ansible on Arch Linux
  - Installing Ansible on Slackware Linux
  - Installing Ansible on Clear Linux
  - Installing Ansible with `pip`
  - Running Ansible from source (devel)
  - Finding tarballs of tagged releases
  - Ansible command shell completion
  - Ansible on GitHub
- Configuring Ansible
  - Configuration file
  - Environmental configuration
  - Command line options

Figure 1.5: *Installation options for Ansible*

In the preceding screenshot, we can see that we have several installation options based on the operating system. We have the Ubuntu operating system as the Ansible control node, so the following commands will install the Ansible:

```
$ sudo apt update
$ sudo apt install software-properties-common
$ sudo apt-add-repository --yes --update ppa:ansible/ansible
$ sudo apt install ansible
```

Verify the Ansible installation

The following command on the Ansible control node will confirm if Ansible is properly installed:

```
ansible@master:~$ ansible --version
ansible 2.9.6
config file = /etc/ansible/ansible.cfg
configured module search path =
['/home/ansible/.ansible/plugins/modules',
'/usr/share/ansible/plugins/modules']
ansible python module location = /usr/lib/python3/dist-
packages/ansible
executable location = /usr/bin/ansible
python version = 3.8.2 (default, Jul 16 2020, 14:00:26) [GCC 9.3.0]
ansible@master:~$
```

Managed node setup

Ansible has a range of options when it comes to manage nodes; we can leverage Ansible even without configuring the manage nodes. All we need is an SSH connection to the remote machine and a privileged user. This is a distinctive option in Ansible in comparison to the other automation tools, which require client software installation on all manage nodes.

For example, you have an infrastructure consisting of thousands of managed servers; installing the client software itself becomes a huge task. Moreover, if you are managing the networking devices, it will not let you install any client software.

The Ansible manage node setup has several options which makes Ansible a unique automation tool, which can automate almost any IT equipment with SSH access.

The following sections talk about the available options for managing node setup.

PASSWORD LESS LOGIN

This is a very useful option available in the Linux operating systems. It allows the administrators to login on the remote servers without typing the passwords. Suppose if we are administrating thousands of servers, each time when you login, you have to type the password, which is time consuming and hectic as well. In order to setup a password less login, we have to complete the following steps:

Create the SSH key on the Ansible control node with the following command, just press *Enter* on each input, which means, select the default options and the SSH key will be created:

```
$ ssh-keygen
```

Copy this SSH key on all the manage nodes with the following commands:

```
$ ssh-copy-id ansible@cen_node_01
$ ssh-copy-id admin1@cen_node_02
$ ssh-copy-id ansible@ubu_node_01
```

When you execute the preceding command, the system will require the password one last time. If the preceding command executes without any error, a password less login is established, which can be verified with the following commands:

```
$ ssh cen_node_01 hostname
$ ssh cen_node_02 hostname
$ ssh ubu_node_01 hostname
$ ssh ubu_node_02 hostname
```

We purposely excluded the ubu_node_02 to see the error. We will be fixing it later when we discuss the inventory file.

MANAGE HOSTS WITH SSH CREDENTIAL

This is the easy-to-use option since we don't have to do any configuration on the manage nodes. This is particularly useful when we are writing playbooks for the networking devices like routers and switches.

If we add the following section in the inventory file, Ansible will use these credentials to login to the manage nodes:

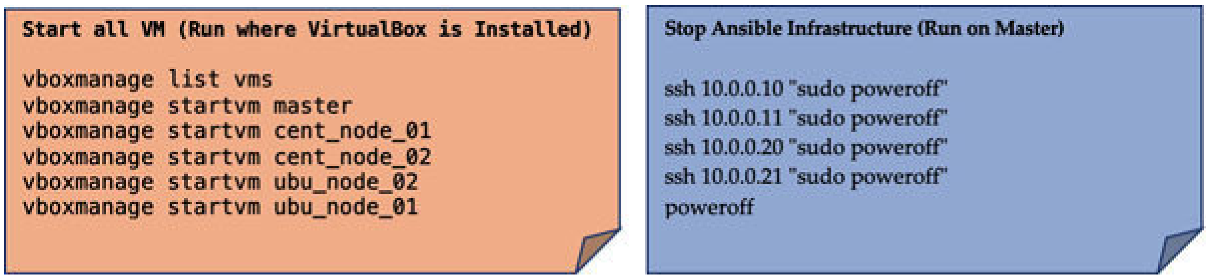
```
[all:vars]
ansible_connection=ssh
ansible_port=22
ansible_user=admin
ansible_pass=P@$$word123
```

ANSIBLE VAULT

Manage host with SSH is a good option but it saves the password in clear text, which is a huge security risk. Therefore, Ansible comes with the Ansible VAULT, which encrypts the credential and only decrypts it before login. We will discuss the Ansible VAULT in the later chapters.

If you are following through the lab, then you can copy the following scripts in a file, give it an executive permission, and run. You will be able to automate the stop and start of all VMs.

This script will work on any Linux box or OSX. For Windows, you might have to adjust a little bit.



```
Start all VM (Run where VirtualBox is Installed)

vboxmanage list vms
vboxmanage startvm master
vboxmanage startvm cent_node_01
vboxmanage startvm cent_node_02
vboxmanage startvm ubu_node_02
vboxmanage startvm ubu_node_01
```

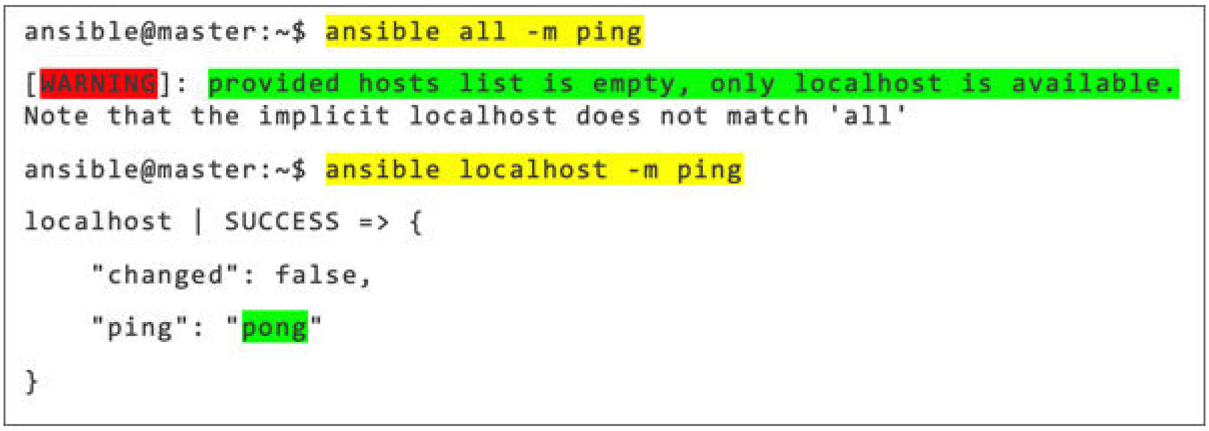
```
Stop Ansible Infrastructure (Run on Master)

ssh 10.0.0.10 "sudo poweroff"
ssh 10.0.0.11 "sudo poweroff"
ssh 10.0.0.20 "sudo poweroff"
ssh 10.0.0.21 "sudo poweroff"
poweroff
```

Figure 1.6: *Command to automate the virtual lab setup*

Important concepts about Ansible

At this point you should have a fully functional Ansible lab setup, which includes a control node named as **master** and four **managed** nodes. Ping command is used in any network to check the connectivity. Ansible has a ping module which does exactly same, If we use the **ping** module to test the lab setup, we will get the following reply from active nodes. Don’t worry we will discuss each and every module and required parameters in details. For the time being, just run the following command:



```
ansible@master:~$ ansible all -m ping
[WARNING]: provided hosts list is empty, only localhost is available.
Note that the implicit localhost does not match 'all'
ansible@master:~$ ansible localhost -m ping
localhost | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Figure 1.7: *Output of ping module*

The following are the key Ansible concepts which are essential to understand Ansible:

Ansible Inventory

Ansible Configuration File

Ansible Modules

Ansible Inventory

Ansible works against multiple managed hosts in your infrastructure, at the same time, using a list or group of lists known as Once your inventory is defined, you can use the patterns to select the hosts or groups that you want Ansible to run against.

The default inventory location is as follows:

When running the scripts, we can provide our own inventory location. So, if we have multiple scripts on different nodes, we can differentiate them with their own inventory files.

We can put the host IP address, hostname, or create groups including the hostname or IP address in the inventory file:

all is a special default group which includes all the hosts from the inventory. Duplicating hosts in different groups is allowed. Nested grouping is allowed. Adding a range of hosts is also allowed, like the following:

```
[db]
db[01:50].example.com
```

The preceding inventory means the group **db** has fifty hosts starting from **db01.example.com** to The same is valid for alphabet, like the following:

```
[db]
db-[a:f].example.com
```

The preceding inventory means the group **db** has six hosts starting from **db-a.example.com** to

VARIABLES IN INVENTORIES

To change the default HTTP port for the web server, we can use the following variables:

```
[atlanta]
host1 http_port=80 maxRequestsPerChild=808
host2 http_port=303 maxRequestsPerChild=909
```

If we have to change the SSH specific configuration, the following variables will be used:

```
Host3.example.com:5309
#Change SSH user for specific host
host1.example.com ansible_connection=ssh ansible_user=myuser
ansible_pass="PWD"
```

```
host2.example.com ansible_connection=ssh
ansible_user=myotheruser
```

For the lab setup, let's take the following sample inventories.

SAMPLE INVENTORY FILE

Let's take the sample files for the following two scenarios:

One VM as control and four VMs as managed nodes.

Single VM as control and managed node.

One VM as control and four VMs as managed nodes

The following is the inventory file for our lab setup in which we have four managed nodes:

```
[centos]
cen_node_01
cen_node_02
[ubuntu]
ubu_node_01
ubu_node_02 ansible_user=admin1 ansible_pass=Pass@123
```

We are using ansible_pass for our learning purpose. In production, never save your password as clear text. We will fix it in the later chapters. If you remember, we left out ubu_node_02 node without

password-less login, we are fixing that here in inventory by providing SSH credentials.

SINGLE VM AS CONTROL AND MANAGED NODE

In case you have a low specs laptop and you can't have multiple VMs, a single Linux VM can be used as the Ansible control and managed node. If you are using a single VM, then the following is the inventory file that you need:

Localhost ansible_connection=localhost

Variable explanation

The following variable names are self-explanatory which will help us in writing complicated scripts in the later chapters:

The name of the host to connect to, if it's different from the alias you wish to give to it.

The connection port number, if not the default for SSH).

The user's name to use when connecting to the host.

The password to use to authenticate to the host.

Equivalent to **ansible_sudo** or allows to force privilege escalation.

After using the preceding sample inventory in the current directory, we will get the following output:

```
ansible@master:~/ansible$ ansible -i inventory  centos -m ping
cen_node_01 | SUCCESS => {
"ansible_facts": {

"discovered_interpreter_python": "/usr/bin/python"
},
"changed": false,
"ping": "pong"
}
cen_node_02 | SUCCESS => {
"ansible_facts": {
"discovered_interpreter_python": "/usr/bin/python"
},
"changed": false,
"ping": "pong"
}
ansible@master:~/ansible$
```

Ansible configuration file

When the Ansible command or playbook is executed, it looks for the **Ansible.cfg** file, which has the Ansible related parameters configured.

Ansible looks for the configuration file in the following order:

Priority variable if set).

Priority the current directory).

Priority the home directory).

Priority /etc/ansible/ansible.cfg (default location)

Ansible will process the preceding list and use the first file found, and all the others will be ignored. Ansible creates a default configuration file at the location when Ansible is installed.

The recommendation is to keep a separate configuration file for each script or a group of scripts, otherwise Ansible will use the default configuration file. Later, if someone modified the parameter of the default configuration file, all your scripts will be affected and it will become a huge troubleshooting task.

We will discuss some basic parameters in the following section; the other parameters will be covered in the later chapters:

Inventory= We can specify the inventory file directly from the configuration file, so we don't have to provide it during the script execution.

Forks = This represents the number of parallel connections that needs to be established. By default, its value is If we increase this value, the script processing time will be enhanced. However, it will generate more load on the Ansible server and network.

sudo_user = Here we can change the **sudo** user to some other user with the admin rights, since the root user is not normally used for the day-to-day tasks.

gathering = This option is used to collect more information about the host machine. This is a very useful feature; we will discuss it further in the later chapters.

timeout = This is for the SSH timeout in seconds; for lab setup **10** seconds is OK, but for production, you must change this to **30** seconds or even **60** seconds. Otherwise, you might get the timeout messages on some hosts, which have high utilization.

Ansible modules

In this section, we will look at some ansible modules in a bit more detail. This is required, so that we can practice developing some more meaningful playbooks. Ansible modules are categorized into various groups based on their functionality. As I said earlier, there are hundreds of Ansible modules available. They are categorized into the following list:

Cloud modules

Clustering modules

Command modules

Crypto modules

Database modules

File modules

Identity modules

Inventory modules

Messaging modules

Monitoring modules

Net tools modules

Network modules

Notification modules

Packaging modules

Remote management modules

Source control modules

Storage modules

System modules

Utilities modules

Web infrastructure modules

Windows modules

Let's discuss some important Ansible categorization of modules, which are as follows:

System System modules are actions to be performed at a system level, such as modifying the users and groups on the system, modifying IP tables, and firewall configurations, working with logical volume groups, mounting operations, or working with services. For example, starting, stopping, or restarting services in a system.

Command Command modules are used to execute the commands or scripts on a host. These could be simple commands using the command module or an interactive execution. You could also run a script on the host using the script module.

File File modules help in working with files. For example, using the ACL module to set and retrieve the ACL information on files, use the archive and unarchive modules to compress and unpack files, and use find to search and replace the modules to modify the contents of an existing file.

Database Database modules help in working with the databases, such as MongoDB, MSSQL, MySQL, or PostgreSQL, to add or remove the databases or modify database configurations, and so on.

Cloud Cloud modules have a vast collection of modules for the various cloud providers like Amazon, Azure, Docker, Google, OpenStack, and VMware being just a few of them. There are a number of modules available for each of these that allow you to perform various tasks such as creating and destroying instances, performing configuration changes, networking and security and managing containers, data centers, clusters, social networking, VSAN, and a lot more.

Windows Windows modules will help you use Ansible in a Windows environment. Some of them are **Win_copy** to copy files, and **Win_command** to execute the command on the Windows machine. And there are a bunch of other modules available to work with the files on Windows. Create an IIS website, install a software using the MSI installer, making changes to the registry using Regedit, and manage services and users in Windows. These are just a few modules in a few categories.

There are a lot more and a comprehensive list can be found at **docs.ansible.com** along with detailed instructions on each of them.

Let's look at a few of these modules in detail to understand how you can use them.

Command

This is default module, if you don't specifically mention any module, this module will be used. It will work only when you have Python installed on manage nodes. The command module takes the command name followed by a list of space-delimited arguments. These arguments are commands which will be executed on all the nodes. The command(s) will not be processed through the shell, so variables like **$HOME** and operations like and **&** will not work.

Shell

This will work only when you have Python installed on manage nodes. It is almost exactly like the command module but runs the command through a shell on the remote node. Therefore, variables like **$HOME** and operations like and **&** will work.

RAW

This module is doing the same thing as Shell and command modules, but the key point is, it doesn’t require Python on the remote machine. This is useful and should only be done in a few cases. A common case is installing Python on a system without Python installed by default. Another is managing any devices such as routers and switches that do not have any Python installed. Arguments given to RAW are run directly through the configured remote shell.

PING

This is a trivial test module; this module always returns ping on successful contact. It does not make sense in playbooks, but it is useful from to verify the ability to login and that a usable Python is configured. This is **NOT ICMP** ping, this is just a trivial test Ansible module that requires Python on the remote-node.

Copy

The copy module copies a file from the local or remote machine to a location on the remote machine.

File

This sets attributes of files, symlinks, or directories. Alternatively, it is used to remove files, symlinks, or directories.

YUM/APT

This installs, upgrade, downgrades, removes, and lists packages, and groups with the respective Linux package manager.

This is just the tip of the iceberg; we have thousands of such modules available for specific purpose like cloud, system, storage, monitoring, database, and network administration. We will cover them in the later chapters.

Basic understanding of YAML

This is the last theoretical concept that we have in Ansible. I am planning to cover all the theory in this chapter. In the following chapters, it will be all hands-on. Understating YAML is essential in writing the Ansible playbooks. Ansible playbooks are the text/configuration files that are written in a particular format called YAML. Therefore, we will spend some time to understand the YAML syntax before jumping to the Ansible playbooks.

All the YAML files can optionally begin with **---** and end with This is part of the YAML format and indicates the start and end of a document.

The following are the YAML components:

The following is the YAML sample list format:

```
---
# A list of fruits
- Apple
- Orange
- Banana
- Mango
...
```

The following is the YAML sample dictionary format:

```
# An employee record
martin:
name: Waqas Irtaza
job: Engineer
skill: Automation
```

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

Proper spacing is essential, otherwise we will get a syntax error depicted as follows:

```
Correct Syntax              Incorrect Syntax
# An employee record          # An employee record
User:                         User:
name: Waqas Irtaza            name: Waqas Irtaza
job: Engineer               job: Engineer
skill: Automation          skill: Automation
```

A dictionary is unordered and a list is ordered; see the following example to understand the main difference between lists and directories.

Dictionary The following example shows the YAML dictionary comparison:

```
# An employee record            # An employee record
user:                          user:
```

```
name: Waqas Irtaza                    skill: Automation
job: Engineer        =               name: Waqas Irtaza
skill: Automation                    job: Engineer
```

List The following example shows the YAML list comparison:/p>

```
---                          ---
# A list of fruits           # A list of fruits
- Apple                         - Apple
- Orange          =             - Banana
- Banana                        - Orange
- Mango                          - Mango
...                          ...
```

More complicated data structures are possible, such as lists of dictionaries, dictionaries whose values are lists or a mix of both. The following is an example to understand the complicated data structures:

```
# Employee records
- Waqas:
name: Waqas Irtaza
job: Engineer
skills:
- python
- Ansible
- John:
name: John Smith
job: Developer
skills:
```

```
- lisp
- fortran
```

Dictionaries and lists can also be represented in an abbreviated form:

```
#Directory

user: {name: Waqas Irtaza, job: Engineer, skill: Ansible}

#List/Array

['Apple', 'Orange', 'banana', 'Mango']
```

For strings in YAML, we don’t have to put them in double quotes, unless they have special characters.

That’s all you really need to know about YAML to start writing the Ansible playbooks. We will be using this YAML format in the following chapters.

Conclusion

In this chapter, we covered the theory of Ansible. Ansible is the simplest way to automate the IT infrastructure. It uses the SSH port for management, which is a default industry standard for the IT infrastructure administration. Zero configuration on managed nodes is a unique feature in Ansible.

We discussed the Ansible components and how they come together. We also discussed the option to install Ansible, configure the managed nodes, and test the installation. Ansible provides a diverse range of options to configure the managed nodes and we discussed all of them in detail. An Ansible inventory is a file, which stores the host or managed nodes in details. The Ansible modules are the code, written for specific tasks and infrastructure. Ansible already has thousands of modules available and ready to be used. The Ansible modules are written for specific networking devices, cloud platforms, and almost all the major aspects of IT. The Ansible scripts are written in YAML, which is a human-readable data-serialization language. It is commonly used for the configuration files. The important components of YAML are list and dictionary. Dictionary is unordered and list is ordered.

In the next chapter, we will understand and practice the Ansible ad-hoc mode and write useful automation scripts for the day-to-day activities. We will cover the basics of the Ansible playbooks and see the different programming options like the conditional statements, variables, loops, and much more.

Multiple choice questions

What is Ansible?

An automation platform.

A programming language.

A GUI interface for the programming language interpreter.

A fictional device.

In which language is the Ansible engine written?

Ruby.

DSL.

Python.

YAML.

What are the advantages of using Ansible?

Agentless.

Very low overhead.

Good performance.

All of the above.

What are the Ansible use cases?

Security and compliance policy integration.

Automated workflow for continuous delivery.

Simplified orchestration.

App deployment.

Configuration management.

Streamlined provisioning.

All of the above.

Which default protocol does Ansible use for host management?

SMTP.

HTTP.

Telnet.

SSH.

What is the name of the Ansible configuration file?

host.

inventory.

ansible.cfg.

ansible_config.cfg.

What is the priority sequence for the Ansible configuration file?

ANSIBLE_CONFIG environmental variable.

ansible.cfg in current directory.

ansible.cfg user home directory.

/etc/ansible/ansible.cfg.

Options are:

1234

3124

1324

4321

None of the above

Multiple choice questions answers

A

C

D

G

D

C

C

Points to remember

Ansible itself is written in Python, however, the Ansible playbooks are written in YAML.

By default, Ansible uses SSH for managing the hosts' machines.

Location and priority of the Ansible configuration file is extremely important.

Command, shell, and raw are all the modules used to execute commands on the remote hosts; however, command doesn't support the environmental variables, and raw is used when the remote machine doesn't have Python installed.

Key terms

The following are the key terms that we have used and defined in this chapter:

Ansible control A server, where Ansible is installed. It must be a Linux machine.

Ansible manage Servers/devices managed through Ansible.

Ansible List of servers/devices.

Ansible configuration Modifying Ansible default parameters.

Ansible Pre-build scripts for specific tasks.

CHAPTER 2

Ansible Basics

Before we start working with Ansible, let's recap what we learned in the previous chapter. We discussed what makes Ansible a popular automation tool. We also discussed the Ansible infrastructure requirements and components. Moreover, Ansible installation and YAML syntax was discussed the in detail.

In this chapter, we will introduce the real-world usage of Ansible. We will discuss the Ansible nodes and when to use them. We will take the real-world tasks and automate them with Ansible. After going through this chapter, you will be able to automate the basic repetitive tasks with Ansible.

Structure

In this chapter, we will cover the following topics:

Ansible ad-hoc mode

Examples for ad-hoc mode

Ansible playbook

Ansible variables

Conditional statements in Ansible

Loops in Ansible

Ansible handler

Ansible Vault

Objective

After studying this chapter, you should be able to do the following:

Use Ansible ad-hoc mode for automation.

Write Ansible playbooks for automation.

Understand Ansible user-defined and built-in variables and conditional statements.

Understand and use different types of loops in Ansible.

Understand and use Ansible handler.

Understand and use Ansible Vault.

Ansible ad-hoc mode

The Ansible ad-hoc mode is a command-line tool to automate single tasks. The Ansible ad-hoc mode is quick and easy, but it is not reusable, since each time you have to provide the desired parameter. We can automate thousands of servers with just a single ad-hoc command, which demonstrates the power of Ansible. The practical use of Ansible ad-hoc mode includes managing users, file copying, rebooting servers, installing or removing packages, and many more. All the Ansible modules can be used in the ad-hoc mode, however the default module is command.

The following is a sample ad-hoc command:

```
$ ansible [pattern] -i inventory-file -m [module] -a "[module
options]"
```

Let us discuss all the inputs for a better understanding.

Pattern

Pattern helps us in selecting the specific hosts or groups from the inventory file. The following table lists the common patterns for targeting the inventory hosts and groups:

groups:
groups: groups: groups: groups: groups: groups: groups: groups: groups:
groups: groups: groups: groups: groups: groups: groups: groups:
groups: groups: groups: groups: groups: groups: groups: groups: groups:

Table 2.1: *Ansible Patterns*

Once you know the basic patterns, you can combine them in any order. The following is an example to demonstrate it:

webservers:dbservers:&staging:!phoenix

We use the wildcard patterns to locate the host from the inventory file. All the hosts that match the wildcard will be selected. The Wildcard pattern can be used for IP or FQDNs. The following example demonstrates the power of using wildcards:

```
192.168.\*
\*.example.com
\*.com
```

We can also mix the wildcard patterns and groups at the same time. The following example will demonstrate it:

```
webservers:dbservers:&staging:!phoenix
```

At this point, one thing should be clear, and that is, if the pattern is not matching any host in the inventory file, the following error will be displayed:

```
[WARNING]: No inventory was parsed, only implicit localhost is
available
[WARNING]: Could not match supplied host pattern, ignoring:
*.not_in_inventory.com
```

Inventory

We have spent a decent amount of time in understanding all about the Ansible inventory in the previous chapter. The **-i** option is used to assign an inventory file for the Ansible ad-hoc mode. If we have defined the inventory file path in the Ansible configuration file, then we don’t have to mention it here.

Modules

We have already discussed about modules in the previous chapter. For the Ansible ad-hoc mode, the **-m** option is used to provide the module name and **–a** is used to provide the module arguments. We will be following a lot of examples which will clarify this point.

Examples for ad-hoc mode

In this section, we will take several useful practical examples to understand how useful the ad-hoc mode is. Most of the examples in this chapter can be used in any production environment. Let's take a look at these examples.

CHECK HOST STATUS

We have already used the **ping** module in the previous chapter. When this module is executed and host is up, we get the **PONG** message in the output. The following is the output for the **ping** module:

```
ansible@master:~/ansible/Chapter-02$ ansible cen_node_01 -i
inventory -m ping
cen_node_01 | SUCCESS => {
"ansible_facts": {
"discovered_interpreter_python": "/usr/bin/python"
},
"changed": false,
"ping": "pong"
}
ansible@master:~/ansible/Chapter-02$
```

This is a quick check to see if the host is up before executing the other commands.

REBOOTING SERVERS

As we discussed earlier, the **-m** option is for selecting the module and **-a** is for the module parameter. However, if we don't specifically mention any module, Ansible will use the **COMMAND** module. At this point, we have the Ansible setup ready and the inventory file has managed host details. To reboot all the servers in a specific group, for example we will be using the following example:

```
$ ansible centos -i inventory -a "/sbin/reboot"
```

As per the Ansible configuration file, by default, Ansible has five parallel connections. Which means, if we reboot ten servers, Ansible will reboot five servers and then the other five servers. This is a configurable parameter, we can either change it in the configuration file or provide it via the command-line. For example, if we want to reboot **20** servers in parallel, the following command will be used:

```
$ ansible centos -i inventory -a "/sbin/reboot" -f 20
```

Ansible will by default to run from current user account. To connect from a different user. For example, current user doesn't have **sudo** access, we can use the following command:

```
$ ansible centos -i inventory -a "/sbin/reboot" -f 10 -u username
```

As we know, only the administrator has the rights to reboot any particular server, therefore, we will be requiring privilege escalation to reboot the servers. For privilege escalation, the following commands will be used:

```
$ ansible centos -i inventory -a "/sbin/reboot" -f 10 -u username --
become [--ask-become-pass]
```

If you add **--ask-become-pass** or Ansible prompts you for the password to use for privilege escalation.

Working with host environmental variable

So far, all our examples have used the default command module which doesn’t support the environment variable. We already discussed this in *Chapter 1: Up and Running with* Here we have an example to elaborate it better. To use a different module, pass **-m** for the module name. For example, to use the shell module, the following command will be used:

```
$ ansible centos  -i inventory -m shell -a 'echo $HOME'
cen_node_01 | CHANGED | rc=0 >>
/home/ansible
cen_node_02 | CHANGED | rc=0 >>
/home/admin1
ansible@master:~/ansible/Chapter-02$
```

Managing files

Managing files is also a regular process, which most system administrators have to go through on a daily basis. Ansible does it beautifully, for example, if you have a file and you want to copy it on several servers, a single Ansible ad-hoc command will do the job. The following is an example which copies a file to all the **[centos]** machines:

```
$ ansible centos -i inventory -m copy -a "src=/etc/hosts
dest=/tmp/hosts"
```

Can you feel the power of Ansible? Suppose we have to copy some files on thousands of servers, a single command can do it in Ansible.

The file module allows changing the ownership and permissions on files. Not only can we copy the file, we can also change the permission and the owner details on the file. This is very important. Suppose you are automating the file copy for some application, and the application users don't have the administrative access, if we just copy the file, file owner will be the automation user, and the application will not be able to use the file. However, we can change the file permission and automate the complete process. The following commands will be used to change permission and owner details:

```
$ ansible centos -i inventory -m copy -a "src=/etc/hosts
dest=/tmp/hosts mode=622"
$ ansible centos -i inventory -m copy -a "src=/etc/hosts
dest=/tmp/hosts mode=622 owner=app_user group=app_user”
```

File module is not yet over; a file module can also create directories, similar to

```
$ ansible centos -i inventory -m copy -a "src=/var/application
dest=/tmp/application mode=755 owner=app_user group=mapp_user
state=directory "
```

We can also recursively delete the files from the Ansible managed nodes by using the following command:

```
$ ansible centos -i inventory -m copy -a "src=/tmp/application
state=absent"
```

Managing packages

Installing or uninstalling packages can also be done with the Ansible ad-hoc. Ansible has several package management modules which can be used for managing packages. For the Debian and Red Hat operating systems, **yum** and **apt** are the most famous modules. The following is the syntax to install using both the modules:

```
$ ansible centos -m yum -a "name=httpd state=present"
$ ansible ubuntu -m apt -a "name=apache2 state=present"
```

The inputs are the same for both the **yum** and the **apt** modules. Therefore, we will only use one of them in the following examples.

We can also force Ansible to only install a specific version by using the following command:

```
$ ansible centos -m yum -a "name= httpd-2.2.26 state=present"
```

To ensure that a latest version of package is installed, we can use the following command:

```
$ ansible centos -m yum -a "name=httpd state=latest"
```

We can also uninstall a package or ensure that a specific package is not installed on the Ansible managed nodes. The following command will do it:

```
$ ansible centos -m yum -a "name=httpd state=absent"
```

Ansible has modules for almost all the package managers. If you are using some legacy or outdated system whose Ansible module is not available, then the Ansible command module can be used to install the packages. Moreover, if you are good with Python, you can write your own package manager.

Managing users and groups

The Ansible ad-hoc mode is also useful to create, remove, and manage the user accounts. The following ad-hoc command will be used to create the user:

```
$ ansible centos -i inventory -m user -a "name=alex
password=password here>"
```

Similarly, we can remove a user by using the following command:

```
$ ansible centos -i inventory -m user -a "name=alex state=absent"
```

Managing services

We can use the service module to start and stop the services on the Ansible managed nodes. The following examples will start, stop, and restart the HTTP service on the CentOS machines respectively:

```
$ ansible centos -i inventory -m user -a "name=httpd
state=started"
$ ansible centos -i inventory -m user -a "name=httpd
state=stopped"
$ ansible centos -i inventory -m user -a "name=httpd
state=restarted"
```

Gathering facts

Fact gathering is another useful feature of Ansible. It collects the managed nodes' information and stores them in variables. This variable can be used to write flexible and complex automation task. The following simple command can be used to get all the variables on managed nodes:

```
$ ansible centos -i inventory -m setup
```

The preceding command will provide a lot of information, but this information will be on Ansible console, which is not the right place to go through such a large amount of data. So, we can use the following command to store all the facts in a file:

```
$ ansible centos -i inventory -m setup --tree /tmp/facts
```

The beauty of the command is that it will create different files for each managed host, name it with the hostname, and save it on the provided directory.

The preceding command is helpful to understand the overall variables, but when you are looking for specific variables, for example to display only facts regarding memory, we can use the following command:

```
$ ansible centos -i inventory -m setup -a 'filter=ansible_*_mb'
```

To display only the facts about certain interfaces, use the following command:

```
$ ansible centos -i inventory -m setup -a 'filter=ansible_eth[0-2]'
```

We will be discussing the facts later in this chapter and use them in writing playbooks. At this point, you should get an idea about the Ansible ad-hoc mode.

Ansible playbooks

The Ansible ad-hoc mode is great for doing individual tasks; however, people mostly prefer the playbook because it allows them to execute multiples task, use variables, conditional statements, and much more. Normally, when we automate some activity, it's not a single task but a series of tasks. The Ansible playbook allows us to organise all those tasks in an easy-to-understand language called YAML. We have already seen the basics of Ansible in the previous chapter. Let's take a very simple task and try to understand how playbooks are written. Let's execute two Linux commands **-a** and in the following example.

The following is the example for the Ansible playbook:

```
---

- hosts: centos
tasks:
- name: Execute uname
shell: uname -a>/tmp/ansible_output.txt

- name: Execute Whoami
shell: whoami  >>/tmp/ansible_output.txt
```

Explanations

Let's add the line number and understand how playbooks are written. Let me repeat again, the number of spaces are extremely important in the YAML format:

```
1 ---
2
3 -_hosts:_centos
4 __tasks:
5 ____-_name:_Execute a uname command on Linux

6 ______shell:_uname -a>/tmp/ansible_output.txt
7
8     - name: Execute Whoami Command on Linux
9         shell: whoami   >>/tmp/ansible_output.txt
```

Important points

The following are the important points when writing the Ansible playbooks; go through them a couple of times until you know them all:

Playbook starts with three hyphens

Line 3 has one space after hyphen and one space after colon

Line 4 has two spaces before tasks.

Line 5 has three spaces before hyphen one space after hyphen and one space after colon

Line 6 has five spaces before the module name and one space after colon

In *Line* we can see as many empty lines as we want to organize our playbooks.

Lines 8 and *9* are a repetition of *Lines 5* and

Understanding the spacing in YAML is extremely important. Don’t worry, after writing two or three playbooks, you will get used to it.

Now let’s understand the following output:

```
ansible@master:~/ansible/Chapter-02$ ansible-playbook my_first_pb
-i inventory

PLAY [centos]
***********************************************************
***********************************************************
**********



TASK [Gathering Facts]
***********************************************************
***********************************************************
***********
```

```
ok: [cen_node_02]
ok: [cen_node_01]

TASK [Execute uname]
**************************************************************
*******************************************************


changed: [cen_node_01]
changed: [cen_node_02]

TASK [Execute Whoami]
**************************************************************
********************************************************


changed: [cen_node_01]
changed: [cen_node_02]

PLAY RECAP
**************************************************************
**************************************************************
************************


cen_node_01                    :
ok=3     changed=2     unreachable=0     failed=0     skipped=0
 rescued=0     ignored=0
cen_node_02                    :
ok=3     changed=2     unreachable=0     failed=0     skipped=0
 rescued=0     ignored=0
```

```
ansible@master:~/ansible/Chapter-02$ ssh cen_node_01
Last login: Tue Sep 15 17:53:01 2020 from master
[ansible@cent_node_01 ~]$ cat /tmp/ansible_output.txt
Linux cent_node_01 3.10.0-1127.8.2.el7.x86_64 #1 SMP Tue May 12
16:57:42 UTC 2020 x86_64 GNU/Linux
ansible
[ansible@cent_node_01 ~]$ exit
```

Important points

The following are the important points to note in the output:

Playbooks are executed with the command **ansible-playbook my_first_pb -i**

Gathering facts is an important point. We will discuss it later in detail.

Whatever name we mention, the playbook will show up in front of

In the final commands, we verified that the commands are executed properly.

Ansible variables

The variable concept for Ansible is the same as in any other programming language, which is a symbolic name of a memory location where the data is stored. In Ansible, we have two kinds of variables. Let’s understand both of the following variables:

User defined variables

Built-in variables

User defined variables

Let's understand the user defined variables with an example. We will create two variables and print them with the echo command. In Ansible, variables are called by double curly brackets

Playbook

The following is an example playbook:

```
---
- hosts: centos
vars:
- name: Waqas Irtaza
- address: Dubai UAE
tasks:
- name: Print variables
shell: echo "My name is {{name}} and I am located in
{{address}}">/tmp/ansible-output.txt
```

Output

The following is the output after running the playbook:

```
ansible@master:~/ansible/Chapter-02$ ssh cen_node_01
Last login: Tue Sep 15 19:12:29 2020 from master
```

```
[ansible@cent_node_01 ~]$ cat /tmp/ansible-output.txt
My name is Waqas Irtaza and I am located in Dubai UAE
[ansible@cent_node_01 ~]$
```

```
ansible@master:~/ansible/Chapter-02$ ssh cen_node_02
Last login: Tue Sep 15 16:51:53 2020 from master
[ansible@cent_node_02 ~]$ cat /tmp/ansible-output.txt
My name is Waqas Irtaza and I am located in Dubai UAE
[ansible@cent_node_02 ~]$
```

This is just a basic example to walk you through variables. We will be using them quite frequently.

Built-in variables

If you remember, we skipped *Gathering Facts* from the first Ansible playbook output. Let's discuss it here. Before Ansible executes a command on hosts, it gathers a lot of useful information about the host and saves it in the built-in variables. We can use these variables in our playbook and make it more diverse.

To get all the built-in variables list, we can use the following ad-hoc command, which we have already discussed in the ad-hoc mode section:

```
ansible -i inventory cen_node_01 -m setup
```

The output is huge, so I am not pasting it here. We can use the filters to get specific information. For example, if we want to know the host operating system, we will use the following command:

```
ansible -i inventory cen_node_01 -m setup -a "filter=*family*"
```

Moreover, if we need the IP details, we will be using the following filter:

```
ansible -i inventory cen_node_01 -m setup -a "filter=*ip*"
```

I hope you got the point. Similarly, we can get a lot of important information about the host, like the following:

CPU type

Operating system

RAM

IP address

CPU cores

Interfaces

We can call these variables in our playbook and make intelligent decision. For example, if we are using the **yum** module in our playbook, we know that the **yum** command doesn't work with the Ubuntu machines, so we can write a conditional statement, which will check the built-in variable for the operating system, and then smartly execute the commands on the centos machines only. We will see this example in the conditional statements.

The built-in variables are used in the playbook in the same manner as the user defined variables, that is, I recommend that you use the preceding setup command and play with the different variables.

Ansible output

Ansible has a module specifically for the output. The module name is and it helps us register the output in a variable. Once the output is in a variable, we can display the output simply or modify it, if required. The output is formatted in JSON, which is slightly different from YAML.

The following is the playbook:

```
---
- hosts: centos
tasks:
- name: Execute whoami Command
shell: whoami
register: whoami_output

- name: Display the output
debug: var=whoami_output.stdout_lines
```

The following is the output:

```
TASK [Display the output]
**************************************************************
**************************************************************
********
```

```
ok: [cen_node_02] => {
"whoami_output.stdout_lines": [
"admin1"
]
}
ok: [cen_node_01] => {
"whoami_output.stdout_lines": [
"ansible"
]
}
```

The output variable **whoami_output** has the JSON output, but if you look closely, we are taking **stdout_lines** which has the desired output. We can also take **stderr_lines** if we need the error message.

The debug module prints the statements during execution and can be useful for debugging the variables or expressions without necessarily halting the playbook. It has the following three parameters:

The customized message that is printed. Moreover, variable required

When using the option, a variable name is **written** without

A number that controls when the debug is run; if you set to it will only run debug when **-vvv** or above.

For example, if we have to write the preceding output with the following playbook can achieve it:

```
---
- hosts: centos
tasks:
- name: Execute whoami Command
shell: whoami
register: whoami_output

- name: Display the output
debug: msg={{whoami_output.stdout_lines}}
```

Conditional statements in Ansible

Conditional statements in Ansible looks quite different. However, functionality wise, it is the same as in any other programming language. The first part of conditional statements is just a normal task, the second part is a condition; the task will only execute when the condition is true. Moreover, we use the special word **When** to check the condition. The following is a sample of the conditional statement:

```
Task:
- Name: Any description (Optional)
Module: module parameters
When: Condition
```

Let's understand it with a practical example.

Playbook

The following is the Ansible playbook:

```
---
- hosts: all
become: yes
tasks:
```

```
#Below script will install apache2 on Debian and ignore centos
hosts

- name: Install apache2 on Ubuntu machines
apt: name=apache2 state=latest
when: ansible_os_family == "Debian"
register: apt_output

- name: Display the console output
debug: var=apt_output

#Below script will install httpd on centos and ignore ubuntu hosts



- name: Install httpd on centos machines
yum: name=httpd state=latest
when: ansible_os_family == "RedHat"
register: yum_output

- name: Display the console output
debug: var=yum_output
```

We used it will allow us to become the root user since the installation requires **sudo** access. Moreover, we can also check multiple conditions as stated in the following example:

```
tasks:
- name: "shut down CentOS 6 systems"
command: /sbin/shutdown -t now
when:
```

```
- ansible_facts['distribution'] == "CentOS"
- ansible_facts['distribution_major_version'] == "6"
```

Loops in Ansible

The concept of loops is also the same in Ansible as in any other programming language.

Loops are used to minimize the repetitive code and repeat it, based on the predefined conditions or logic. Any network or system administrative task, when executed for *N* number of servers, can be simplified with the Ansible loops. In Ansible, we have two options for creating loops: **loop** and

with_list

with_items

with_indexed_items

with_flattened

with_together

with_dict

with_sequence

with_subelements

with_nested/with_cartesian

with_random_choice

The loop function was added in Ansible 2.5 and the following versions, although the Ansible documentation recommends loops and there is a document available in the official site, which has examples of replacing **with_** with loops. However, I found **with_** is extremely simple and I will be focusing on it for looping. We will take some examples which will give you an idea on how these loops work.

Install listed software on Ubuntu machines

The following is the playbook which will install the list of software on the Ubuntu machines:

```
---
- hosts: ubuntu
become: yes
tasks:
- name: Install below listed packages on ubuntu machines
apt: name={{item}} state=latest
with_items:
- nano
- vim
- python
- wget
```

Display content of two files

The following playbook will display the content of two files:

```
---
- hosts: ubuntu
become: yes
tasks:
- name: Display content of files
debug: msg="{{item}}"
with_file:
- ip.txt
- host.txt
```

Print a sequence from 0 to 10

The following playbook will print the sequence:

```
---
- hosts: ubuntu
become: yes
tasks:
- name: Display sequence from 0 to 10
debug: msg="{{item}}"
with_sequence: start=0 end=10
```

Let's take the following example to compare **loop** and

with_sequence

```
---
- hosts: ubuntu
tasks:
- name: with sequence
debug: msg="{{item}}"
with_sequence: start=0 end=4
```

loop

```
---
- hosts: ubuntu
tasks:
- name: with_sequence -> loop
```

```
debug: msg= "{{item}}"
loop: "{{range(0, 5)|list}}"
```

Both the preceding loops do the same thing. An important point to note is that when we are using a range command in loops, the upper limit should be one more than the repetition and the special word list is used to convert the integers to list, which is required for loop.

Ansible handler

Handlers are just like the normal tasks in an Ansible playbook, but it will run when called by another task that contains a **notify** directive. A Handler is useful for the secondary actions that might be required after running a task, such as starting a service after the installation or configuration change. The following example will show how the Ansible handlers are used:

```
---

- hosts: ubu_node_01
become: true
tasks:
- name: Install Package
apt: name=apache2 state=present
notify:
- Start apache2

handlers:
- name: Start apache2
service: name=apache2 state=started
```

Let's see the following sample to understand the key features of handlers:

Tasks

- Task1

Handler_name

- Task2
- Task3

Handler_name

- Task4

Handler
- Handler_name

Suppose we have four tasks, out of them, two are calling a handler as depicted in the sample. However, handler will only run once. This is the key feature which makes it different from a standard task.

Moreover, a handler will only be executed if the task made a change on the node. Otherwise, the handler will not run. In the preceding example, the handler will only run if **Task1** or **Task3** has made a change on the nodes. It is a very useful feature; suppose we are installing **apache2** on a host and then restarting the service – if a host already has **apache2** installed, restarting the service might affect the current service.

Ansible error handling

Ansible normally has defaults settings which make sure to check the return codes of the commands and modules; if Ansible playbook fails, Ansible exits the playbook with an error. Ansible provides several ways to change this default behaviour; let's discuss them in the following section.

Ignoring failed commands

Generally, the Ansible behaviour is to stop executing any more steps or commands on a host that has a task failed, which makes sense. For example, we have written a playbook which installs Apache and then starts the service. If the installation fails, it does not make any sense to run a command, which will try to start the service. In this case, both the commands are dependent. However, if the commands are independent and we want to keep going, then we have to add the following highlighted command in our playbook, which will change the Ansible default behaviour and execute all the tasks for the particular host, even if some tasks get failed:

```
- name: If fails, this command will not be counted as a failure
command: /bin/false
ignore_errors: yes
```

Resetting unreachable hosts

The default Ansible behavior is to set the host unreachable; if the host did not respond on time, and the Ansible timeout is breached – which can happen because of several issues, for example network or server utilization was high and host could not respond on time – then Ansible will remove the host from the active host.

The Ansible tasks will be executed on the active host only; in case we suspect there will be a false positive for the unreachable hosts, and we want to change the Ansible behavior to auto recover the unreachable host, then we have to use **clear_host_errors** in playbook.

Controlling what defines failure

In Ansible, we have the options to set a criterion which defines what *failure* means in each playbook task using the keyword A list of multiple **failed_when** conditions are joined like a conditional statement, meaning the task only fails when all the conditions are met.

Ansible Vault

The Ansible Vault encrypts the variables and files, so we can protect the sensitive content, such as passwords or keys, rather than leaving it visible as plaintext in playbooks. To use the Ansible Vault, we need one or more passwords to encrypt and decrypt the content.

Since it is common to store the Ansible configurations in the version control, we need a way to store the secrets securely.

The Ansible Vault is the answer to this. Ansible Vault can encrypt anything inside a YAML file, using a password of your choice. A typical use of the Ansible Vault is to encrypt the variable files or playbooks. Vault can encrypt any YAML file. Let's see how to use the Ansible Vault.

To create an encrypted file, the following command will be used:

```
$ ansible-vault create learn_ansible.yaml
New Vault password:
Confirm New Vault password:
$
```

Once we enter the command, it will prompt us for a password. This password is very important, it will be required to create, modify, or run this playbook. Once we enter the password, it will open a VI editor file, in which we can write normal Ansible plays, as shown in the following playbook:

```
---
- hosts: all
become: yes
vars:
- ansible_sudo_pass: Pass321

tasks:

- name: Install apache2 on Ubuntu machines
apt: name=apache2 state=latest
register: apt_output

- name: Display the console output
debug: var=apt_output
```

Once the playbook is written as per your requirement, just exit out of the VI editor. Let’s open the newly created file and see the content, as shown in the following example:

```
$ cat learn_ansible.yaml
$ANSIBLE_VAULT;1.1;AES256
34623530316263656532366230343530663466396436386132386465626 33
36538653533633230323535343164343665663236346130353137333937636 53
9620a663763346132303537643564316431633063313064373565643764356
43938333534656233393466653138636331356332313038313538623265303 13
63566320a643163663239306534323530616231613333343534373166363939
333438666637373762396365633466343436326666646130623831623837 3
```

```
9393038333561316231616334393765366439663630376532343363633266 3
6653531336462616633626538333930363630313131666661623732356162 36
3161646133633661326530346261343739346338396538336131323638313531
3237663666333653563363036333762326534333231376435646365323764323
9356432353964626563666234616561653362343631313631313031336630 32
32643135373934313335653737383833

6362323839633433383764646365386131313864363934346161643164613 73
7376565336439666166633461373631663532323537303466653263653763 6
1656466626666623131383866353966666338323738373735356136396131 6
3633763623834643534653339356536376664343566656432653439346266
6366653431313553135666330343961663361616361623439623038393232323
663663334333862376662623933323735303631643834303235666630666 13
7656530613737363464393534326666373762616239363431333731653331 61
353134313763653436326634396463363866386532646432303132646339 37
3565383963616362613662383037383335643230383161613762323938333 56
6366137653165656133363533613730353235613135353463313861373665313 8
3439353763383162323762643865306166633761396463393038373963653 6
3665656234303561383733633336356 3
64393838396261393561306433656537633 1
$
```

Now the playbook is encrypted; the following are some useful commands that work with this encrypted file.

Modify the playbook

The following command will let us edit the encrypted playbook:

```
$ ansible-vault edit learn_ansible.yaml
Vault password:
$
```

Run the playbook

The following command will let us run the encrypted playbook:

```
$ ansible-playbook learn_ansible.yaml -i inventory --ask-vault-pass
Vault password:
$
```

There is a catch; we have to type the password each time we are executing the playbook, which is not ideal. There is another option to run the playbook without having to type the password, but you have to save the vault password in a clear text file, which is still a security risk, since anyone read the file, and also modify and run your playbook. We can give that file read-only permission to the automation user, so that no one else can read the password. I know it's not ideal but it can do the job easily. The following is the command to run the playbooks without requiring the password:

```
$ ansible-playbook learn_ansible.yaml -i inventory --vault-password-file password.txt
```

This is just the basics for Ansible Vault. You will see the benefits of it when we discuss the roles and the other advanced topics in the next chapter.

Conclusion

In this chapter, we discussed the Ansible ad-hoc and Ansible playbooks. The Ad-hoc commands can run individually to perform quick functions. These commands are not required to be used frequently; for example, if we have to reboot the list of servers, we will be using the ad-hoc commands, since in production, we rarely reboot servers. The biggest benefit of the ad-hoc command is its simplicity and power which it provides the user. We have discussed a couple of use cases for the ad-hoc mode.

The Ansible playbooks offer a repeatable, re-usable, simple configuration management, and multi-machine deployment system, one that is well-suited to deploying complex applications. Ansible plays are written in YAML. YAML stands for **Yet Another Markup** YAML allows Ansible to use the conditional statements, loops, and many other programmable features which simplify the automation process.

In the next chapter, we will focus on the advanced concepts of Ansible and the best practices for writing the Ansible playbooks.

Multiple choice questions

Can we run the Ansible ad-hoc command without providing an inventory file in the command-line?

No, we have to provide the inventory file.

Yes, if we provide the inventory file path in the **ansible.cfg** file, then we don't have to provide the inventory file.

Can I run an Ansible ad-hoc command without any module?

No, Ansible must need a module to execute the command.

Yes, if no module is mentioned, Ansible uses the default module, named

Are spaces important to consider when writing the YAML playbooks?

Yes.

No.

Are empty lines important to consider when writing YAML playbooks?

Yes.

No.

Which Ansible module supports the managed host environmental variables?

Command.

Shell.

Raw.

Ping.

Which Ansible module does not require Python to be installed on managed nodes?

Command.

Shell.

Raw.

Ping.

Multiple choice answers

B

B

A

B

B

C

Points to remember

The Ansible ad-hoc mode is used for simple non repetitive tasks, which aren't planned to be used in the future.

An Ansible playbook gives a programming flavor to Ansible.

An Ansible playbook is used for complex, repetitive tasks.

Ansible offers two keywords for creating loops – **loop** and

By default, the Ansible playbook will not provide any useful information about the actual task; we must use the output module to get the desired output.

Ansible playbooks will stop executing any more steps on a host that has a task failed; we must use the command **ignore_errors:** if we want to execute the remaining commands.

The Ansible Vault encrypts the variables and files, so that we can protect sensitive content such as passwords or keys or even full playbooks.

Key terms

Ansible ad-hoc The Ansible ad-hoc command is the one-liner Ansible command that performs one task on the target host.

Ansible Ansible playbooks are one of the core features of Ansible that tells Ansible what to execute. They are like a to-do list for Ansible that contains a list of tasks.

CHAPTER 3

Ansible Advance Concepts

Before we start working with Ansible, let's recap what was learnt in the previous chapter. We discussed how to use Ansible in the ad-hoc mode. We even wrote the basic playbooks covering loops, conditional statements, and variables. We then covered some day-to-day automation using Ansible.

In this chapter, we will discuss the advanced concepts associated with Ansible. These concepts will help you write a professional playbook.

Structure

In this chapter, we will cover the following topics:

Manage control tasks

Ansible file separation

Ansible include module

Ansible roles

Ansible optimization

Troubleshooting Ansible

Ansible file lookup

Ansible template

Ansible dynamic inventory

Ansible filters

Objective

After studying this chapter, you should be able to do the following:

Write playbooks as per the standard.

Troubleshoot Ansible for complex tasks.

Optimize the overall performance of Ansible.

Create static and dynamic inventories.

Managing task control

This module covers the different solutions to control a task, which includes the following topics:

with_items

Nested loops

When statement

Registering variables

Handlers

Tags

Errors

Ansible blocks

We discussed some of the options in the previous chapter at the introductory level. In this chapter, we will focus on the practical usage of these concepts.

With_items

In this section, we will explore the **with_items** in Ansible. It is used for looping in Ansible. Loops are used to repeat a task based on different items. Loops avoid writing multiple tasks. **with_items** is a key word which will define a list of items that needs to be processed. It is used as a label, listing the different items. To refer to the items, **{{item}}** is used. We also have a flexible option available to use the variables for a list of items. It makes the Ansible playbook more diverse. We have seen the examples for the straight-forward examples in the previous chapter, so I will not explain them here. However, the option for complex item is also available, so let's understand the following example:

```
-name: Manage Group Membership for users
user:
name: “{{item.name}}”
state: present
groups: “{{item.group}}”
with_items:
- {name: ’Rony’, group: ‘student’}
- {name: ’Ali’, group: ‘teacher’}
- {name: ‘John’, group: ‘teacher’}
```

So, in the preceding example, **{name: ’Rony’, groups: ‘students’}** is an item. However, in this example, an item is a dictionary, so we can refer to the dictionary keys. When we refer the the playbook with iterate on and Similarly, when we refer the playbook will iterate on the user group name, which, in our example, are and

Let’s take another example, in which we will see the practical usage of We want to uninstall two packages and delete the files associated with them. The following is the example playbook:

```
---
- name: Clean-up the FTP and Webserver content
hosts: centos
task:
- name: Uninstall Packages
yum:
name: "{{item}}"
state: absent
with_items:
- httpd
- vsftpd
- name: Delete FTP and HTTP files
file:
path: "{{item}}"
state: absent
with_items:
- /var/www/html/index.html
- /var/ftp/pub/README
- /tmp/www/http_backup.tar
- /tmp/ftp/ftp_backup.tar
```

In the preceding example, we used two loops, one is to provide the list of packages, which we need to delete, and the other will provide the file paths, which we want to delete as part of cleanup. These sorts of playbooks are very easy to expand and reuse. For example, if we have to add another package to be removed, all we need to do is to put the name in first

Nested loops (with_nested)

A nested loop is a loop within a loop. It is very common in programming languages, in which two lists are used, and the task will run on the items in the first list, and combine with the items in the second list. I know, theory wise it looks complex but actually it is not. So, let's see the following example to understand the concept:

```
task:
- name: Add Users in Groups
user:
name: "{{item[0]}}"
state: present
groups: "{{item[1]}}"

with_nested:
- ['rony','ali','john']
- ['student','teacher','teacher']
```

Nested loop is actually a better version of especially, when you have to run multiple loops. Variables can be used in **with_nested** same as

When statement

We have already discussed the **when** statement in *Chapter 2, Ansible Basics* when discussing the conditional statements. Conditionals allow to run the Ansible tasks only if the minimal conditions have been met.

Registering_variables

The idea of registering the variables is to store the output of a task in a variable. The result is multi-value, each value is stored in a key. To do something with it, we will need to refer to a specific value **variablename.value** format. Let’s take the following example to understand the concept:

```
---
- name: Understand Registering Variables
hosts: cen_node_01
tasks:
- name: Capture the output of who command in a variable
command: who
register: loggedin_users
- shell: echo "Ansible user is logged in"
when: loggedin_users.stdout.find('ansible') != -1
```

Output

The following is the playbook output:

```
$ ansible-playbook register_variable.yml -i inventory -v
Using /home/ansible/ansible/Chapter-03/ansible.cfg as config file

PLAY [Understand Registering Variables]
****************************************************************
```

```
*******************************************

TASK [Gathering Facts]
***********************************************************
***********************************************************
*
ok: [cen_node_01]

TASK [Capture the output of who command in a variable]
***********************************************************
************************************



changed: [cen_node_01] => {"changed": true, "cmd": ["who"],
"delta": "0:00:00.003309", "end": "2020-10-18 06:48:05.806548",
"rc": 0, "start": "2020-10-18 06:48:05.803239", "stderr": "",
"stderr_lines": [], "stdout": "ansible  pts/0        2020-10-18 06:48
(master)", "stdout_lines": ["ansible  pts/0        2020-10-18 06:48
(master)"]}

TASK [shell]
***********************************************************
***********************************************************
***********


changed: [cen_node_01] => {"changed": true, "cmd": "echo
\"Ansible user s logged in\"", "delta": "0:00:00.003027", "end":
"2020-10-18 06:48:06.106427", "rc": 0, "start": "2020-10-18
06:48:06.103400", "stderr": "", "stderr_lines": [], "stdout": "Ansible
user s logged in", "stdout_lines": ["Ansible user s logged in"]}
```

```
PLAY RECAP
***************************************************************
***************************************************************
**************
cen_node_01                    :
ok=3     changed=2     unreachable=0     failed=0     skipped=0
 rescued=0     ignored=0
$
```

The first thing to note here is that we have used the **-v** option when executing the playbook. It will provide a verbose output and we can see the **stdout** output variable has the Linux **who** command output. The increasing number of **v** in **–v option** increases the level of verbose output that the user will get for the play run, like and

If we explore the playbook, we have executed the Linux **who** command and saved the output in a variable named The actual output value is stored in Now, in the second task, we took this variable output and used the Ansible build-in function **find()** and searched for the username Based on the **find()** output, we can decide which action we want to perform next. In our playbook, we only print **Ansible users logged** but it can be any action.

Let's take another example which is more practical, and you may use it for the day-to-day activities. The following is the playbook:

```
---
```

```
- name: Backup Linux user home directories
hosts: all
tasks:
# TASK 1
- name: Create a directory for backups
file:
path: /var/bkpspool
state: directory
#TASK 2
- name: Retrieve the list of home directories
command: ls /home
register: home_dirs
#TASK 3
- name: add home directories in Backup folder
file:
path: /var/bkpspool/{{item}}
src: /home/{{item}}
state: link
with_items: "{{home_dirs.stdout_lines}}"
```

The goal of the preceding playbook is to create a backup folder, get the list of home directories, and back them up in a backup folder.

Let’s discuss each task separately as follows:

TASK 1

This task is very simple. All we did was create a folder on the following location:

TASK 2

In this task, we created a variable named **home_dirs** and saved the **ls /home** command output in it. We know the output of this command is the list of all directories in the **/home** folder.

TASK 3

This is the actual place where all the magic happens. We use the **home_dirs.stdout_lines** for For each item, we are creating a symbolic link in the backup folder.

An important point to node is stdout and both store the output. However, if we need the output line-by-line, then we use the stdout_lines variable. If we need the output as a whole, then we will use In the previous example, we needed the output in the line-by-line format, since we needed to use with_item to iterate.

Handler

Handler is also an important concept in task control. We have already discussed the handler in the previous chapter. However, I will just state its properties for a recap. A Handler is a conditional task that only runs after being called by another task. It has a globally unique name and is triggered after all the tasks in the playbook. A Handler can be triggered by one or more tasks in a playbook. The Handler properties are the same as any task. To trigger a handler, the playbook must have a notifying item, which will call the name of the handler. More than one handler can be called from a task. A Handler always runs in the order in which the handler section is written, not in the order in which they are called in the plays. A Handler cannot be included in the playbook; they must be part of the playbook. We haven't discussed, **include** so far, but once we discuss, the **roles** and **include** statement later in this chapter, this point will be cleared.

Tags

Tags are also an import part of the complex playbooks. Tags are used at a resource level to give a name to a specific resource. The **–tags** are used with the Ansible playbook to run only the resources with a specific tag. When a task file is included in a playbook, it can be tagged in the **include** statement. When tagging a role or inclusion, the tag applies to everything in the role or in the inclusion. To run a playbook with a specific tag, use the command **Ansible-playbook --tags** This command will only run the resources which matches the We can also use **- - skip-tags 'tagname'** to exclude a resource with a specific tag. We have a special tag named which can be used to make sure that resource is always executed, unless specifically excluded with **- -** When the **- - tag** option is used to run a playbook, it can take up to three specific tags as an argument. **all** is another the default tag, which will run all the tasks; it is the default behavior of Ansible. An alternative of tags is creating multiple playbooks. If you want to create a single big playbook and segregate it based on tags, then use tags, otherwise create multiple playbooks. The following example will explain how to use the tags:

```
---
- name: Understand Ansible Tags
hosts: centos
tasks:

- name: Install network Analysis Packages
```

```
package:
name: "{{item}}"
state: installed

with_items:
- nmap
- wireshark
tags:
- network_analysis

- name: Install lamp packages
package:
name: "{{item}}"
state: installed
with_items:
- httpd
- mariadb-server
tags:
- lamp
```

In this example, if we want only the network analysis packages to be installed, then we have to use the following command:

```
$ ansible-playbook ansible_tags.yml -i inventory --tags
network_analysis
```

Similarly, if we want to install only the **lamp** packages, the following command will be used:

```
$ ansible-playbook ansible_tags.yml -i inventory --tags lamp
```

Dealing with errors

The Ansible default behavior is to stop the execution when the task has failed. We have already discussed in the previous chapter that we can use **ignore_errors: yes** and it will ignore the error and the playbook with continue running. When we are dealing with handlers, if a play fails, no handler will be executed. To overcome this issue, we can use **force_handler: yes** in a playbook. There is another interesting option called **failed_when** to define when to consider a specific command failed. This is useful for the commands that are producing a specific output. Let's see the following sample for our understanding:

```
tasks:
-shell: /usr/local/bin/mkusr.sh
register: mkusr_result
failed_when: "'password missing' in mkuser_result.stdout"
```

In the preceding example, the task is considered as failed only if the password missing is found in the registered variable.

If a module thinks it has changed the state of the affected machine, it will report the status as If this is not the desired option and you want to tweak this parameter, then the **changed_when=false** option will do it.

Blocks

Blocks are used to logically group tasks. Blocks are useful for error handling and **when** statements. One statement can be applied to the block, so that it affects all the tasks in the block. The following example will explain how to use blocks in playbooks:

```
--
- name: Understand Blocks
hosts: all
tasks:
- name: Installing Apache
block:
- package:
name: "{{item}}"
state: installed
with_items:
- httpd
- elinks
- mod_ssl
- service:
name: httpd
state: started
enable: true
when: ansible_distribution == 'CentOS'
```

Output

The following is part of the playbook output:

```
skipping: [ubu_node_01] => (item=httpd)
skipping: [ubu_node_01] => (item=elinks)
skipping: [ubu_node_01] => (item=mod_ssl)
skipping: [ubu_node_02] => (item=httpd)

skipping: [ubu_node_02] => (item=elinks)
skipping: [ubu_node_02] => (item=mod_ssl)
changed: [cen_node_02] => (item=httpd)
changed: [cen_node_01] => (item=httpd)
changed: [cen_node_02] => (item=elinks)
changed: [cen_node_01] => (item=elinks)
changed: [cen_node_02] => (item=mod_ssl)
changed: [cen_node_01] => (item=mod_ssl)
```

In the above example, we used the **when** statement at the block level, which make sure that installation only happen on CentOS hosts and skips the Ubuntu machines.

Blocks allow error handling; if a task fails, the task in the rescue task can be executed for recovery. We also have an **always** task, which will run regardless of success or failure of the task defined in blocks or rescue. Let's take the following example to understand the **rescue** and **always** keywords in blocks:

```
---
- name: Error handling in Blocks
hosts: all
```

```
tasks:


- block:
- name: Upgrading Database
shell:
cmd: /opt/db_scripts/upgrade-database


rescue:
- name: Revert If Upgrade failure happens
shell:
cmd: /opt/db_scripts/revert-database


always:


- name: Restart the database
service:
name: mariadb
state: restarted
```

In this example, Ansible block option will upgrade the database. If upgrade script executed properly, the **rescue** block will be ignored and the **always** block will be executed.

In case upgrade script failed, the **rescue** block will be executed, which will revert all the database changes; once it's done, the **always** block will restart the database. So, either **block** or rescue will be executed at any time, but the **always** block will execute in any case.

Ansible file separation

We will now be talking about file separation. So far, we've been defining the variables in the same inventory file. However, this is not the best practice. It is better to define these in a separate variable file. To do this, first create a **host_vars** directory next to the playbook and then create a file with the same name as that of the server, which, in this case, is **cen_node_02.yml** and If you remember when we deployed the lab in *Chapter 1 Up and Running with* we can configure the credentials in the inventory file by using the Ansible variables. We will move the variables and their values from the inventory file into this new file. Remember to remove the = sign and change it to a colon followed by a space; this is because this new file is in a YAML format.

When the Ansible playbook is executed, Ansible automatically reads the values from this file, and associates them with the host. So, it's important to name the file with the same name as that of the server, and also, it's important to name the folder because that's the folder that Ansible looks set to find out whether there is a variable file for that particular server. For the host variables, the folder must be named **host_vars** and the group variables must be named and you could have a file for each host inside this folder. The following are the changes which we have made on our lab:

```
$ cat inventory
[centos]
cen_node_01
```

```
cen_node_02
[ubuntu]

ubu_node_01
ubu_node_02

$ tree

.
├── ansible.cfg
├── host_vars
│   ├── cen_node_02.yml
│   └── ubu_node_02.yml
└── inventory

1 directory, 4 files

$ cat host_vars/cen_node_02.yml
ansible_host: 10.0.0.11
ansible_user: admin1
ansible_password: Pass321

$ cat host_vars/ubu_node_02.yml
ansible_host: 10.0.0.21
ansible_user: admin1
ansible_password: Pass321
$
```

Ansible Output

```
$ ansible all -i inventory -a "whoami"
```

```
cen_node_01 | CHANGED | rc=0 >>
ansible

cen_node_02 | CHANGED | rc=0 >>
admin1

ubu_node_01 | CHANGED | rc=0 >>
ansible

ubu_node_02 | CHANGED | rc=0 >>
admin1

$
```

Ansible include statement

Let’s understand the **include** statement with an example; suppose we have a playbook with two sets of tasks – one to install and configure the database, and the other to install and configure the web server. *What if we need to reuse these tasks in different playbooks where the requirement may be to only install the database or to only install the web server?* So, we first create two files under a folder called naming them **deploy_db.yml** and and move the **db** tasks into the first one and the web tasks into the second one. Once we're done, we can add the **include** statements under the task in our playbook to include the task from these external files:

Ansible roles

The Ansible roles provide a uniform way to load variables, tasks, and handlers from the external files. The purpose of roles is to keep the size of the playbooks manageable. Roles use a specific directory structure with the location for default task, handlers, templates, and variables. A role typically corresponds to the type of service, for example, we have a role for docker, web service, or database service. For specific groups of servers, specific playbooks may be created to include one or more roles. To manage what should happen, the default variables are set in the roles, which can be overwritten at the playbook level. To make working with the role easier, community roles can be downloaded from the Ansible galaxy. Roles are defined in a role's directory, which is created in the project directory.

Directory structure for role

The following are the important folders for roles. Let’s see what these folders contain, as follows:

```
$ tree .
.
└── test-role-1
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
└── main.yml
9 directories, 8 files
$
```

defaults

This folder contains a **main.yml** file with default values for variables.

files

This folder has static files that are referenced by role's tasks.

handlers

This folder contains a **main.yml** with handler definitions.

meta

This folder contains **main.yml** with information about the roles, including author, license, platform, and dependencies.

tasks

This folder has a **main.yml** file with task definitions.

vars

This folder has a **main.yml** file with role variable definitions.

To create roles, we first create the role structure. We don't have to create structure manually. We can use the **ansible-galaxy** command which is, by default, installed with Ansible. The following command will create a role's directory structure:

```
$ ansible-galaxy init test-role-1
```

Once the preceding command is executed, a new folder with a name of **test-role-1** will be created. Let's see the content of the folder, as follows:

```
$ tree .
.
└── test-role-1
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml

├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
└── main.yml
9 directories, 8 files
$
```

Once the structured role's directory is created, we can define the role content, and then use these roles in the playbook. Each role has its own directory with specific subdirectories that exist in **~/roles** and not in specific project directories. So, if we are not using any specific subdirectory, we can keep it empty.

Role variables

The Role variables are defined in these variables have a high priority and cannot be overwritten by the inventory variables. The default variables can be defined in **defaults/main.yml** and have the lowest precedence. So, we can use the default variables only if we intend to have the variable overwritten somewhere else.

Defining role dependencies

We can expect some dependencies of roles like roles may include other roles. Dependencies are written in **meta/main.yml** within the role:

```
---
dependencies:
- {role: apache, port: 80}
- {role: mariadb, dbname: addresses, admin_user: Rony}
```

Order of execution in role

Normally, the tasks in a role is executed before the tasks of the playbook using them, but we have two solutions to override it, which are as follows:

pre_tasks are performed before the roles are applied.

post_tasks are performed after completing all roles.

The following is a sample playbook which gives an idea how to implement the preceding discussed concepts:

```
---
- hosts: cen_node_01
pre_tasks:
- debug:
msg: 'Starting'
roles:
- role1
- role2
- role3
tasks:
- debug:
msg: Execute After roles'
post_tasks:
- debug:
```

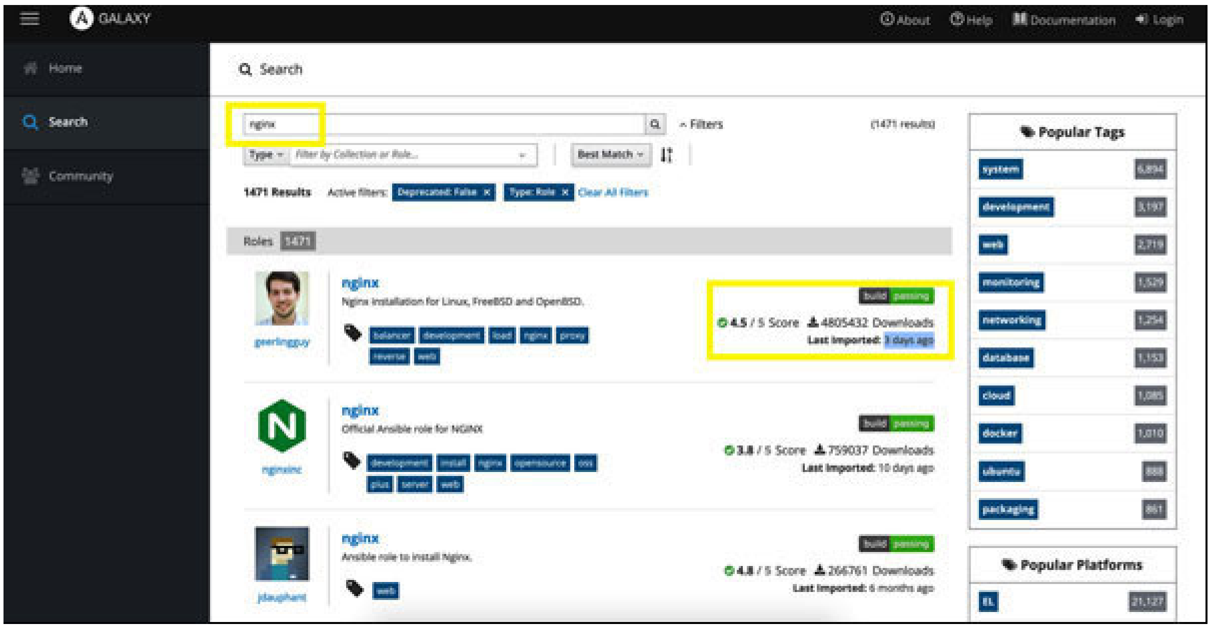
```
msg: 'Last Task'
```

Ansible Galaxy

Ansible Galaxy is the community resource for getting and publishing Ansible roles. The website for the Ansible Galaxy is as follows:

https://galaxy.ansible.com/

Since it is a community resource, many roles that are ready to use are available for download. The roles that are still in development can also be followed. Let's visit the site and see how we can utilize the Ansible Galaxy. The following is a screenshot taken from the official site:



nginx

(1471 results)

1471 Results Active filters: Deprecated: False × Type: Role × Clear All Filters

Roles 1471

nginx
Nginx installation for Linux, FreeBSD and OpenBSD.
balancer development load nginx proxy reverse web
geerlingguy
build passing
4.5 / 5 Score 4805432 Downloads
Last Imported: 3 days ago

nginx
Official Ansible role for NGINX
development install nginx opensource oss plus server web
nginxinc
build passing
3.8 / 5 Score 759037 Downloads
Last Imported: 10 days ago

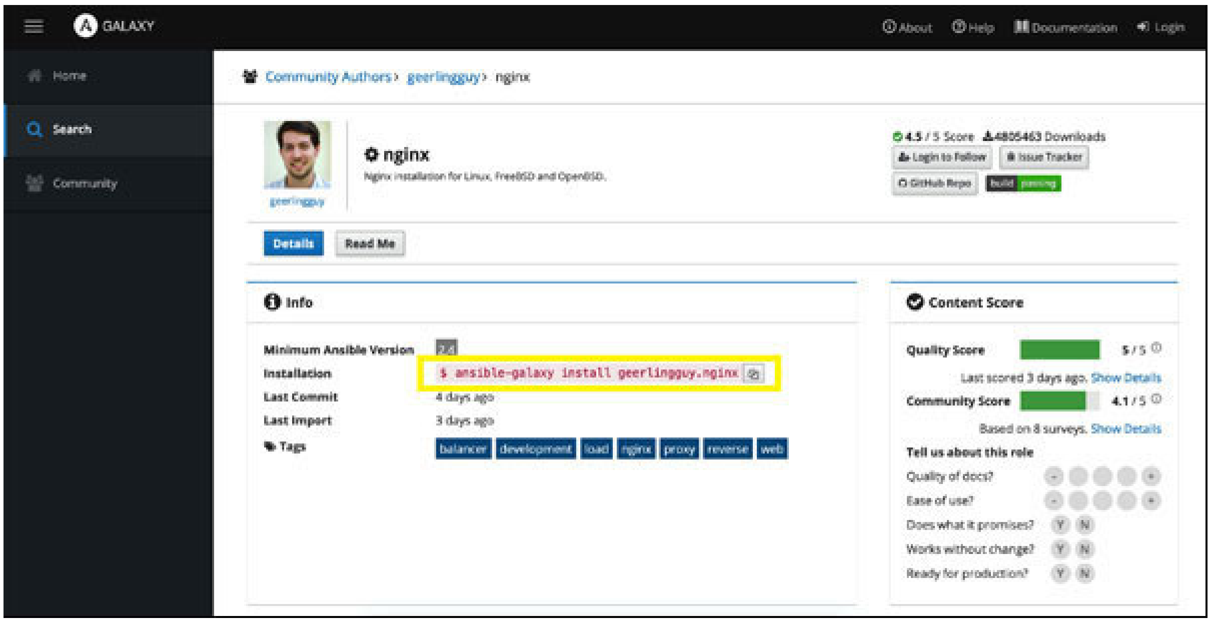
nginx
Ansible role to install Nginx.
web
jdauphant
build passing
4.8 / 5 Score 266761 Downloads
Last Imported: 6 months ago

system	6,894
development	3,197
web	2,719
monitoring	1,529
networking	1,254
database	1,153
cloud	1,085
docker	1,010
ubuntu	888
packaging	861

EL	21,127

Figure 3.1: *Ansible Galaxy*

We can go to search and look for any roles. For example, I have searched **nginx** and got a couple of results, but the top most result is from The score of the role, number of downloads, and last imported are the key factors in choosing a role. Let's suppose we are happy with the statistics and want this role. Just click on **nginx** and the following screen will be shown:



nginx
Nginx installation for Linux, FreeBSD and OpenBSD.

4.5 / 5 Score 4805463 Downloads

Info

Minimum Ansible Version	2.4
Installation	$ ansible-galaxy install geerlingguy.nginx
Last Commit	4 days ago
Last Import	3 days ago
Tags	balancer development load nginx proxy reverse web

Content Score

Quality Score 5 / 5
Last scored 3 days ago. Show Details
Community Score 4.1 / 5
Based on 8 surveys. Show Details
Tell us about this role
Quality of docs?
Ease of use?
Does what it promises? Y N
Works without change? Y N
Ready for production? Y N

__Figure 3.2:__ Ansible Galaxy geerlingguy Nginx role

All we need to do is just copy the installation command and run on the Ansible control node. The following is the output that we received after executing the command:

```
$ ansible-galaxy install geerlingguy.nginx
- downloading role 'nginx', owned by geerlingguy
- downloading role from https://github.com/geerlingguy/ansible-role-nginx/archive/2.8.0.tar.gz
- extracting geerlingguy.nginx to
/home/ansible/.ansible/roles/geerlingguy.nginx
- geerlingguy.nginx (2.8.0) was installed successfully
$
```

As per the output, the role has been created on the location so we can go to this directory and see the content of this role. The following is the output from the folder:

```
$ tree
.
├── defaults
│   └── main.yml

├── handlers
│   └── main.yml
├── LICENSE
├── meta
│   └── main.yml
├── molecule
│   └── default
│       ├── converge.yml
│       └── molecule.yml
├── README.md
├── tasks
│   ├── main.yml
│   ├── setup-Archlinux.yml
│   ├── setup-Debian.yml
│   ├── setup-FreeBSD.yml
│   ├── setup-OpenBSD.yml
│   ├── setup-RedHat.yml
│   ├── setup-Ubuntu.yml
│   └── vhosts.yml
├── templates
│   ├── nginx.conf.j2
│   ├── nginx.repo.j2
│   └── vhost.j2
└── vars
├── Archlinux.yml
├── Debian.yml
├── FreeBSD.yml
├── OpenBSD.yml
└── RedHat.yml


8 directories, 23 files
$
```

Now we can use this role in our playbook.

Galaxy CLI tool

ansible-galaxy search will search for roles. This argument provided with **ansible-galaxy search** will search in the role description. We can use options like **- - - -** and **- - galaxy-tags** to narrow down the search results. Let’s take the following example to search

```
$ ansible-galaxy search 'install nginx' --author 'geerlingguy'

Found 16 roles matching your search:

Name                                   Description
----                                   -----------
chaos_jetzt.freescout            Install and configure the freescout
helpdesk.
geerlingguy.certbot              Installs and configures Certbot (for
Let's Encrypt).
geerlingguy.collectd-signalfx SignalFx Collectd installation for Linux.
geerlingguy.drupal               Deploy or install Drupal on your
servers.
geerlingguy.fathom               Fathom web analytics
geerlingguy.gitlab               GitLab Git web interface
geerlingguy.munin                Munin monitoring server for
RedHat/CentOS or Debian/Ubuntu.
geerlingguy.nginx                Nginx installation for Linux, FreeBSD
and OpenBSD.
```

```
geerlingguy.php                    PHP for
RedHat/CentOS/Fedora/Debian/Ubuntu.
geerlingguy.pimpmylog            Pimp my Log installation for Linux
geerlingguy.varnish              Varnish for Linux.

leifmadsen.kibana-4               Kibana 4 for Linux
monsieurbiz.geerlingguy_php      PHP for
RedHat/CentOS/Fedora/Debian/Ubuntu.
Oefenweb.nginx                      Set up (the latest version of)
NGINX in Debian-like systems
sarathkgit.nginx_geerlingguy   Nginx installation for Linux, FreeBSD
and OpenBSD.
ubzyhd.nginx                        A ansible role to install NGINX +
apply basic security and hardening settings.
$
```

Once you have the name of the role, you can use the following commands to get the information and install the role:

```
$ ansible-galaxy info geerlingguy.nginx
$ ansible-galaxy install geerlingguy.nginx
```

We have some more galaxy commands which are very helpful. The following are the commands:

To remove a role from the Ansible control node, use the following command:

```
$ ansible-galaxy remove geerlingguy.nginx
```

Next, we will create the directory structure for your own role, but by default it will use the Ansible Galaxy API. We should specify the username and role name as the argument. However, if you want to work offline, use the following parameter

```
$ ansible-galaxy init myuser:My_role_name
$ ansible-galaxy init --offline myuser:My_role_name
```

Now, once role is either downloaded or created; we need to know how to use the role in a playbook. We will create the following playbook and add the role in it:

```
---
- name: How to use role
hosts: cen_node_01

roles:
- geerlingguy.nginx
```

Ansible optimization

In this module, we will cover the topics of optimizing the Ansible, which will enhance the processing speed and result in a better overall performance of Ansible. The following are some key Ansible optimizing parameters.

Host patterns

Host patterns are about what you are addressing in a playbook or the Ansible ad-hoc while running it. This simplest host pattern is just a name of the host, an IP address may also be used as long as they are available in the inventory. We can also use groups in our lab environment – CentOS and Ubuntu are the two groups. We also have the default groups available like we don't have to mention it in the inventory, it automatically knows all the host in the inventory file. We also have which refers to all the host that are not part of any group in the inventory. We can use * but don't forget to use single quote. A single quote is recommended in all cases to ensure that the shell doesn't interpret special characters. Refer to the following example:

```
Ansible '*.example.com' -i inventory –list-hosts
```

The preceding example means all the hosts in the inventory, which has the domain name

Moreover, a comma-separated list of hosts is also allowed. The following example clarifies one more important concept, which is, we can list different items like host, group, and IP address in our logic:

```
Ansible 'cen_node_01.example.com,10.0.0.21' -i inventory –list-hosts
```

Ampersand can be used as a logical operator, which means the host matches both the items. This makes sense when we talk about groups. For example, we have two groups, one is the **database** and the other is the We need the hosts that are common in both the groups. Then **&** will be used.

Exclamation point can be used as the not logic, which means, we can specifically exclude a host by putting this operator:

```
Ansible 'centos,!cen_node_01' -i inventory –list-hosts
```

So, in this example from the CentOS group, all the members will be included except

Configuring delegation

Sometimes, Ansible's scope goes beyond the managed hosts, so we have to configure the hosts that are not available in the inventory. Think of the monitoring environment, where the remote hosts need to be added in the environment or a DNS server that needs to be modified after adding a server to the configuration. We can run the tasks directly on the hosts involved, but that will be slow. Fact-gathering needs to happen for all the hosts involved. Delegation can happen to the different types of hosts like the local machines, hosts outside the play, or hosts within or out of the inventory. Let's take the following example to see how can we use delegations:

```
---
- name: Understand Delegation
hosts: cen_node_01
tasks:
- name: Get PS information from remote host
command: ps
register: remote_ps
changed_when: false

- name: Get PS information from local host
command: ps
delegate_to: localhost
register: local_ps
changed_when: false
```

```
- name: Print the output for remote PS
debug:
msg: "{{remote_ps.stdout}}"

- name: Print the output for local PS


debug:
msg: "{{local_ps.stdout}}"
```

Since localhost is not part of our inventory file, we will use the Ansible delegation to run a command on it. The delegate module will run on a host specified by in case you want to run the delegate command on the Ansible master. We did it in the preceding example, we can also use which is a shortcut for However, this shortcut is only applicable for the Ansible control node.

Addressing hosts that are in the inventory is straightforward, just use the IP address or the host name. To address hosts outside the inventory, use the **add_hosts** module. An important point to node is, don’t forget to configure the **delegate_to** hosts with a user account, **sudo** credentials, and SSH keys, because Ansible can only help when it is authenticated on the server.

Delegation host outside of inventory

In this case, we will see how to address a host which is not available in the inventory. However, as we discussed in the previous section, we will be requiring credentials. When accessing a host outside of the inventory, a temporary entry in the inventory must be created by using Ansible will use the same connection type and the details used for the managed host to connect to the delegating hosts. To understand better, I have removed the **ubu_node_01** from the inventory and applied the following playbook:

```
---
- name: Testing Add host
hosts: localhost
tasks:
#Task-1
- name: Add another host
add_host:
name: add_ubuntu
ansible_host: ubu_node_01
ansible_user: ansible
#Task-2
- name: Check where the command is running
command: hostname
delegate_to: add_ubuntu
register: output1
Task-3
```

```
- name: Check how facts are handled
command: echo "This is {{inventory_hostname}}"
delegate_to: add_ubuntu

register: output2

- name: Print the outputs
debug:
msg: "{{output1.stdout}}"
debug:
msg: "{{output2.stdout}}"
```

The following are the explanations for each task:

In this task, we will add a host which is not available in the inventory.

We will execute the commands of the newly added host and register them in a variable.

This is the trickiest part; we suspect that the Ansible built-in variables will provide the details of the newly added host, but actually it is not true. Facts gathering only work for the localhost host, not for the added host. So, we will get the details of localhost, when we print the output of the variable not the delegated host.

The default behaviour regarding facts, is that facts are gathered on the host where the playbook is running and not on the delegated hosts, which is super confusing. However, we can use **delegate_facts: true** to gather facts from the **delegate_to**

Parallelism in Ansible

Running the tasks in parallel will make Ansible faster. Ansible can run tasks in parallel on all the hosts. By default, tasks run on five hosts at once. This is because of the default configuration option, which is **forks=5** in **/etc/ansible/ansible.cfg** or wherever you put the Ansible configuration file. However, if we change the value of forks to any other number, that much Ansible hosts will run in parallel. If we want to override the Ansible configuration file, then we can use the **--forks** option with the playbook or ad-hoc mode.

We also have the **serial** keyword in a playbook to reduce the number of parallel tasks to a value that is lower than what is specified in the **forks** option.

Asynchronous tasks

Normally, Ansible waits for the completion of the tasks before starting the next task. However, we can use the **async** keyword in a task to run it in the background. For example, **async: 3600** tells Ansible to give the task an hour to complete. This is the maximum amount of time permitted for a task. We also have the **poll: 10** option which indicates that Ansible will poll every **10** seconds to see if the command has been completed. Using **async** allows the next task to be started as it will make the overall playbook more efficient. It is recommended for the backup job, updating packages, or downloading large files.

Ansible wait_for

It can be used in a task to check if a certain condition was met. It may be useful to verify if the server restart was successful. Use **poll: 0** in a task to tell Ansible not to wait for completion of this task, but to move on to the next task. Add **ignore_errors** as well, to prevent an error condition arising, and have this task fail.

Ansible async_status

For a start and forget type of task, we can use the module to see the current status of the task. This allows you to finish everything in the playbook, and close the run when you get positive results from

Troubleshooting Ansible

Troubleshooting Ansible is extremely important since we need to know what's happening if things don't go as per the plan. The following are the key topics when it comes to troubleshooting in Ansible.

Ansible logging

By default, Ansible doesn't log anything, because it has sufficient information written in We can see the error messages directly on the Ansible console. However, if we need the logs to be saved in a file, then we can specify the **log_path** in the default section of **ansible.cfg** to force the writing logs in files. We can also set the **$ANSIBLE_LOG_PATH** variable to enforce the logging in a file. An important point to note is that only the root user can log to so we should consider the log files in the local playbook directory, or provide the write permission on **/var/logs** for your automation user. A recommended option is to save the logs in the local directory, so that you don't have to modify the system default settings. Once we applied these logging parameters, the error messages which were written on STDOUT will be written to the log file. We also need to configure the **logrotate** on the Ansible logs. The following example will demonstrate the creation of log file. We have added the following highlighted line in the **ansible.cfg** file's default section:

```
[defaults]
log_path= errorlogs
```

We did not specify the full path, so Ansible took the relative path, which means it will create the log file to the place where the Ansible playbook is executed. Once it's done you can run any previously created playbook, and a new log file will be created.

Ansible common errors

Understanding the common errors will help troubleshooting the playbooks. For example, if there is a connectivity issue like the server is unreachable or wrong credentials, these are not actually a playbook error. A playbook error can be a syntax error, extra spaces, or things like that. We can use the **ansible-playbook --syntax-check playbook_name.yml** command to check if the playbook has any issue. If your playbook has many tasks, then **--syntax-check** will produce a high volume of messages. To avoid this, we can add **--step** to execute the tasks one by one. In this way, troubleshooting will become easy.

The Ansible playbook's output messages also give a good hint during troubleshooting. For example, the play header will show us which play is currently executed, and the task header will show which task is creating an error. So, automatically, when the error comes, we know which playbook lines have the error. Moreover, we can increase the log detail by adding **-v** or **-vv** or **-vvv** or **-vvvv** which add the following verbosity levels:

-v option mean verbosity in output data.

-vv mean verbosity in input and output data.

-vvv also includes information about connection to manage host.

-vvvv includes information about plugins, users, and scripts that have been executed.

By adding each we are getting more and more logs for troubleshooting.

Troubleshooting managed hosts

ansible-playbook - - check playbook.yml is used to perform the test without modifying anything on the targeted hosts. However, the Ansible modules that we are using, must offer support for the check mode, because if the module doesn't support the check mode, nothing will be shown. We can also use **check_mode** to specify if a specific task needs to be executed in the check mode or not. So, if the **check_mode:** task will to be executed during check mode and if the **check_mode:** task will not be executed during check mode.

Ansible modules for troubleshooting

Some Ansible modules provide additional information about the host status; check the following as examples:

The **uri** module can connect to a URL and check for specific content.

The **script** module supports execution of scripts on managed nodes.

The **stat** module can check if specific files are present on the managed nodes.

The **assert** module evaluate if given expressions are true with an optional custom message

Ansible ad-hoc commands for troubleshooting

We can use the Ansible ad-hoc commands to perform quick tests before running the playbooks on the managed nodes. The following are a few of the examples:

Check if specific package is installed:

```
$ ansible cen_node_01 -i inventory -m yum -a " name=httpd
state=present"
```

Check if manage host has the required disk size available:

```
$ ansible cen_node_01 -i inventory -a "df -h"
```

Check managed host RAM and swap utilization:

```
ansible cen_node_01 -i inventory -a "free -h"
```

Check if specific user is available:

```
$ ansible cen_node_01 -i inventory -m user -a 'name=ansible'
```

Check if the SSH connection is properly established with managed host:

Ansible file lookup

In this section, we will be talking about lookups. So far, we've been storing the credentials for our target servers inside the inventory file or the **var_host** folders. *What if there are too many servers or this information is already available elsewhere?* Let's say, we have the credentials of the servers stored in a CSV file in a host name and password format – the first column being the host name and the second column being the password. To read the contents of the file, while the Ansible playbook is running and to get the password associated to a host, we can use the **lookup** plugin. The following is how we use the **lookup** plugin:

```
{{lookup('csvfile', 'cen_node_03 file=tmp/my_servers.csv
delimiter=,'}}
```

For example, in this case, the first argument that we passed to the **lookup** plugin is the type of the file which happens to be **csv** file in this case. There are a number of other options available that we'll see in a bit. Then comes the value to lookup. For example, here we'd like to lookup the information about the server followed by the file we are looking at. For example, in this case, the file is stored under the **tmp** directory and is called and finally, the delimiter which happens to be comma in this case, because it's a CSV file. If you have a file which uses another separator maybe like a semicolon or something, then you could specify that here. So, this same lookup plugin for the CSV file supports multiple other character's separated file formats as well. In this case, this whole function or plugin is going to return the password from the CSV file.

There are a couple of lookup plugins available, such as the following:

CSV file lookup

INI file lookup

Credstash lookup

DNS lookup

MongoDB lookup

The information about this can be seen in the Ansible documentation page under the *Special topics and lockups* playbook.

Ansible template

Ansible has a very useful module template, it provides the ability to use the templates in which, we can make use of the Ansible variables, conditions to generate the specific data formats, and eventually provide a custom template for each manage host. Afterwards, this template will be transferred to manage hosts. We can also create a specific template for Windows by using the **win_template** module. The template concept is quite frequently used in the DevOps world. However, it is also applicable for the networking devices which have the configuration files in the HTTP or XML formats.

The template data is generated, based on the **Jinja2** templates. The Jinja2 template is a modern-day templating language for the Python developers; since Ansible itself is written in Python so we can leverage the Jinja2 templating benefits here. Jinja2 templates are used to create the HTML, XML, or other mark-up formats that returned to the user via an HTTP request.

In futuristic technologies, we have less requirement for static data and more for the dynamic data being requested from the end users. Therefore, Jinja2 templating is being used. The templates are processed by the Jinja2 templating engine. Also, the way these templates are formatted is an important aspect.

To use the template, we have to set at least the following two parameters:

This is the source path for the template file with relative or absolute path.

The destination path, where you want to copy the template on the managed hosts. Moreover, if the directory path is given, then a file with the same name as the template will be created.

We also have the optional parameters, let's discuss them for a better understanding:

This option is used to set the permission for the destination file.

This option is used to change the group of the file on remote hosts.

This option is used to change the owner of the file on remote hosts.

By default, this option is set to which means if there is an already existing file on the destination, then the target file will be overwritten. However, if we explicitly set this option to then it will not replace the existing file.

If the template file already exists and you want it to have a backup, then use this parameter and set to By doing this, every time a new file with the same name is pushed from the controller node, then the older file on the remote hosts will be renamed with the name with date and time.

We can also enforce the new-line sequence. To specify the newline sequence to use for templating files, the acceptable values are **\n** and

Let's take a couple of the following examples to understand how the template works in Ansible.

Example 1:

In this example, we will transfer a plain text template to managed nodes:

Playbook

The following is the playbook used for this example:

```
---
- name: Template Example 1
hosts: centos
tasks:
- name: understand Ansible Template
template:
src: template01.j2
dest: /tmp/template-01.txt
```

Ansible output

The following is the output from the Ansible playbook:

```
$ ansible-playbook template_example01.yml  -i inventory

PLAY [Template Example 1]
*************************************************************
***********************************************************


TASK [Gathering Facts]
*************************************************************
*************************************************************
*


ok: [cen_node_02]
ok: [cen_node_01]

TASK [understand  Ansible Template]
*************************************************************
*************************************************


ok: [cen_node_02]
ok: [cen_node_01]

PLAY RECAP
*************************************************************
```

```
****************************************************************************
*************

cen_node_01                    :
ok=2    changed=0    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
cen_node_02                    :
ok=2    changed=0    unreachable=0    failed=0    skipped=0

rescued=0    ignored=0
$
```

Verification

If the Ansible playbook is executed properly, the template will be transferred on the Ansible managed nodes. We are using the following commands to verify the template creation:

```
$ ssh cen_node_01 "cat /tmp/template-01.txt"
This is simple template file
$ ssh cen_node_02 "cat /tmp/template-01.txt"
This is simple template file
$
```

Explanation

So, we have a plain text template file which we want to transfer to the manage nodes. Since the template and the playbook, both are in the same folder, we have used the relative path, otherwise we have to use the full path like **/home/ansible/download/** The rest of the example is straight-forward.

Example 2:

Let's add some built-in variables or facts to understand how we can integrate them in the out templates.

Playbook

The following is the Ansible playbook used for this example:

```
---
- name: Template Example 2
hosts: centos
tasks:
- name: understand Ansible Template with FACTS
template:

src: template02.j2
dest: /tmp/template-02.txt
```

Jinja2 template file

The following is the Jinja2 template file for this example:

```
Server Details for template

Hostname:  {{ansible_hostname}}
```

```
IP Address:   {{ansible_all_ipv4_addresses}}
Operating system: {{ansible_distribution}}
```

Playbook verification

We can open the destination files to verify if the template worked as per requirements:

```
$ ssh cen_node_01 "cat /tmp/template-02.txt"
Server Details for template

Hostname :   cent_node_01
IP Address:  ['10.0.2.15', '10.0.0.10']
Operating system: CentOS

$ ssh cen_node_02 "cat /tmp/template-02.txt"
Server Details for template

Hostname :   cent_node_02
IP Address:  ['10.0.2.15', '10.0.0.11']
Operating system: CentOS
$
```

Explanation

This is an important example, which will explain how the dynamic data can be added in the template. So, here we use the Ansible facts and when the playbook is executed, Ansible intelligently replaces the facts with the respective managed host details. I hope the basic idea of the template module is clear so far. We can also use our own variables instead of facts, and Ansible will replace these variables with the values when the playbook is executed.

Ansible dynamic inventory

We have already discussed about the Ansible inventory file, which has the host details available; such an inventory is called the static inventory. However, if Ansible has to cover all the IT domains, then the static inventory is not realistic in the production environments. In the real world, infrastructure keeps changing rapidly. We also add new nodes and remove the unwanted nodes. This is very common, especially for the cloud and container environments, where the nodes are added and removed based on usage. Therefore, Ansible came up with a concept called **dynamic** which fetches the list of nodes from the infrastructure environments in real time.

In the modern IT environment, the sources of host nodes can be AWS, Azure, OpenStack, containers, and LDAP systems. These systems have their list of nodes and Ansible integrates with such external dynamic inventory.

In Ansible, we have the two following ways to connect to the external dynamic inventories:

Via This is the older way which ensures backward compatibility.

Ansible Ansible recommends this over the scripts for dynamic inventory, since the plugins are updated with the Ansible core code.

The Dynamic inventory scripts are written in a programming language like Python, PHP, and so on. When using the script, they get real time data from the target source environments, like AWS, OpenStack, GCP, and so on. The Ansible community have already developed such kind of scripts for the majority of platforms. If we have good programming skills, we may write such scripts as well, but using scripts is not very well-documented and not successful every time. Normally, we download the scripts for the source environment and set the **env** variable as per the guidelines available with script. These scripts are maintained by the Ansible community. Mostly, the scripts are written in Python and named as Along with these Python scripts, there are the INT files as well, which are needed to set the environment, but this is dependent on the script that we are using.

In Ansible, we have a set of available inventory plugins. Similarly, we can create our own plugins. Normally, the inventory returns the list of items in the JSON format. Ansible provides a bunch of inventory plugins, using which we can use to get the list of target hosts on which you will be running any task or play. For example, we can use the following commands to get the list of available Ansible plugins for our Ansible. These plugins will vary, depending on which Ansible version we are using:

```
$ ansible-doc -t inventory -l
advanced_host_list   Parses a 'host list' with ranges
auto                      Loads and executes an inventory plugin
specified in a      YAML config
aws_ec2                    EC2 inventory source
```

aws_rds	rds instance source
azure_rm	Azure Resource Manager inventory plugin
cloudscale	cloudscale.ch inventory source
constructed	Uses Jinja2 to construct vars and groups based on existing inventory
docker_machine	Docker Machine inventory source
docker_swarm	Ansible dynamic inventory plugin for Docker swarm nodes
foreman	foreman inventory source
gcp_compute	Google Cloud Compute Engine inventory source
generator	Uses Jinja2 to construct hosts and groups from patterns
gitlab_runners	Ansible dynamic inventory plugin for GitLab runners
hcloud	Ansible dynamic inventory plugin for the Hetzner Cloud
host_list	Parses a 'host list' string
ini	Uses an Ansible INI file as inventory source
k8s	Kubernetes (K8s) inventory source
kubevirt	KubeVirt inventory source
linode	Ansible dynamic inventory plugin for Linode
netbox	NetBox inventory source
nmap	Uses nmap to find hosts to target
online	Online inventory source
openshift	OpenShift inventory source
openstack	OpenStack inventory source
scaleway	Scaleway inventory source
script	Executes an inventory script that returns JSON

```
toml                    Uses a specific TOML file as an inventory
source

tower                   Ansible dynamic inventory plugin for Ansible
Tower
virtualbox              virtualbox inventory source
vmware_vm_inventory VMware Guest inventory source
vultr                   Vultr inventory source
yaml                    Uses a specific YAML file as an inventory
source
$
```

I have highlighted the most famous Ansible modules. Once we get the name of the module for our infrastructure, we can use the following command to get more details on the plugin:

```
$ ansible-doc -t inventory name>
```

To use the Ansible plugin, we have to enable it, in where we have the **enable_plugins** parameter for this purpose. The following are the default list of enabled plugins that ships with Ansible:

```
[inventory]
enable_plugins = host_list, script, auto, yaml, ini, toml
```

If the plugin is from a collection, then we need to use the fully qualified name:

```
[inventory]
enable_plugins = namespace.collection_name.inventory_plugin_name
```

For example, if we want to enable **aws_ec2** plugin, then we have to add the following line in

```
# demo.aws_ec2.yml
plugin: amazon.aws.aws_ec2
```

We will be using the dynamic inventory in the following chapters, when we discuss the cloud administration with Ansible.

Ansible filters

When writing playbooks in Ansible, we need data manipulation, processing, and formatting, since the output of one task may be used as the input for another task. For example, if we are writing a playbook for the network and the first task is to check the default route, then use the Ansible filters to get the service provider's IP address; once we have the IP address, we will use the **ping** module to check the reachability.

We have a set of filters, the Jinja2 template filters, and the custom filters. Filters in Ansible are from Jinja2, so the input data is transformed inside a template expression. Also, templating happens on the Ansible control node, not the managed nodes. So, the filters get executed on the Ansible control node. This is helpful, as in this way, the amount of data which needs to be transferred to the remote hosts is lesser.

Ansible has a rich set of filters powered by Jinja2 templating. We input some data into these templates and the Jinja2 template engine process that data and provide the formatted output. Filtering is very useful in debugging. The following are a list of built-in filters shipped by Jinja2:

Jinja2:
Jinja2:

Jinja2:
Jinja2:
Jinja2:
Jinja2:
Jinja2:

Jinja2:
Jinja2:
Jinja2:

Table 3.1: *Jinja Build-in Filters*

Let's discuss each filter to understand their functionality, as follows:

This filter will return the absolute value of the argument.

This filter will get an attribute of an object.

This filter will batch the items. It works pretty much like slicing.

This filter will capitalize a value. The first character will be uppercase, all others will be lowercase.

This filter will centre the value in a field of a given width.

This filter will provide the default value to a variable if it is undefined.

dictsort(): This filter will sort a dictionary and yield (key, value) pairs.

This filter will convert the characters and ” in strings to the HTML-safe sequences.

This filter will format the value like a human-readable file size (that is, 200 KB, 400 MB, and so on).

This filter will convert the value into a floating point number.

This filter will enforce the HTML escaping.

This filter will apply the given values to a **printf** format string like **string** % values.

This filter will group a sequence of objects by an attribute, using Python’s

This filter will return a copy of the string with each line indented by four spaces.

This filter will convert the value into an integer.

This filter will return a string which is the concatenation of the strings in the sequence. The separator between the elements is an empty string by default, but we can define it with the optional parameter.

This filter will return the last item of a sequence.

This filter will return the number of items in a container.

This filter will convert the value into a list. If it was a string, the returned list will be a list of characters.

This filter will convert a value to lowercase.

This filter will apply a filter on a sequence of objects.

This filter will return the largest item from the sequence.

This filter will return the smallest item from the sequence.

This filter will pretty print a variable. It is useful for debugging.

This filter will return a random item from the sequence.

This filter will filter a sequence of objects by applying a test to each object, and rejecting the objects that pass the test.

This filter will filter a sequence of objects by applying a test to the specified attribute of each object, and rejecting the objects that pass the test.

This filter will return a copy of the value with all the occurrences of a substring replaced with a new one provided in the parameters.

This filter will reverse the object.

This filter will round the number to a given precision.

This filter will mark the value as safe, which means that in an environment with automatic escaping enabled, this variable will not be escaped.

This filter will filter a sequence of objects by applying a test to each object, and only selecting the objects which pass the test.

This filter will filter a sequence of objects by applying a test to the specified attribute of each object, and only selecting the objects with the test succeeding.

This filter will slice an iterator and return a list of lists containing those items.

This filter will sort using Python's

This filter will make a string Unicode if it isn't already.

This filter will return a title cased version of the value.

This filter will dump a structure to JSON, so that it's safe to use in the