LEARN ANSIBLE QUICKLY

I dedicate this book to my late friend, Mostafa. You will never be forgotten.

BRIEF CONTENTS

CONTENTS IN DETAIL

Acknowledgments

I am very thankful to everyone who's supported me in the process of creating this book.

First and foremost, I would like to thank Abhishek Prakash who served as the main editor of the book. Without Abhishek, this book would be full of random errors!

I am also very grateful for the support I have from the Linux Foundation; I was the recipient of the 2016 & 2020 LiFT scholarship award, and I was provided with free training courses from the Linux Foundation that have benefited me tremendously in my career.

A big shoutout also goes to all readers on **Linux Handbook** and my 160,000+ students on **Udemy**. You guys are all awesome. You gave me the motivation to write this book and bring it to life.

Last but not least, thanks to Linus Torvalds, the creator of Linux. You have changed the lives of billions on this earth for the better. God bless you!

Introduction

Learn Ansible Quickly is a fully practical hands-on guide for learning Ansible. It will get you up and running with Ansible in no time.

About the Author

Ahmed Alkabary is a professional Linux/UNIX system administrator working at IBM Canada. He has over seven years of experience working with various flavors of Linux systems. He also works as an online technical trainer/instructor at Robertson College.

Ahmed holds two BSc degrees in Computer Science and Mathematics from the University of Regina. He also holds the following certifications:

- Red Hat Certified Engineer (RHCE).
- Red Hat Certified System Administrator (RHCSA).
- Linux Foundation Certified System Administrator (LFCS).
- AWS Certified DevOps Engineer - Professional.
- AWS Certified Solutions Architect - Associate.
- Azure DevOps Engineer Expert.
- Azure Solutions Architect Expert.
- Cisco Certified Network Associate Routing & Switching (CCNA).

Why this Book?

If you are tired of spending countless hours doing the same tedious task on Linux over and over again then this book is for you! *Learn Ansible Quickly* will teach you all the skills you need to automate borings tasks in Linux. You will be much more efficient working on Linux after reading this book, more importantly, you will get more sleep, I promise you! *Learn Ansible Quickly* does assume prior Linux knowledge (RHCSA Level) and that you have experience working on the Linux command line.

How to use this Book

I have intended this book to be used in two ways:

- ***As an educational text.*** You will read this book from the start chapter by chapter. You will also do the lab exercises at the end of each chapter to reinforce your learning experience.
- ***Earning RHCE certification.*** This book fully covers all the RHCE EX294 exam objectives; You can use *Learn Ansible Quickly* as your main preparation guide to help you become a Red Hat Certified Engineer.

To get the most out of this book, you should read it in a chronological order as every chapter prepares you for the next, and so I recommend that you start with the first chapter and then the second chapter and so on till you reach the finish line. Also, you need to write and run every playbook in the book by yourself, do not just read an Ansible playbook!

Contacting Me

There is nothing I like more in this life than talking about Ansible, DevOps, and Linux; you can reach me at my personal email *ahmed.alkabary@gmail.com*. I get a lot of emails every day, so please include *Learn Ansible Quickly* or *#LAQ* in the subject of your email.

Don't Forget

I would appreciate any reviews or feedback, so please don't forget to leave a review after reading the book. Cheers!

Chapter 1: Hello Ansible

In this chapter, you will get to understand what is Ansible all about and how is Ansible different from other automation and configuration management tools. Next, you will create the setup for our Ansible environment that we are going to use throughout the book to showcase all Ansible components and tools. Finally, you will learn how to install Ansible on RHEL (Red Hat Enterprise Linux), CentOS, and Ubuntu.

What is Ansible?

Ansible is an open-source configuration management, software provisioning, and application deployment tool that makes automating your application deployments and IT infrastructure operation very simple.

Ansible is very lightweight, easy to configure, and is not resource hungry because it doesn't need an agent to run (agentless) unlike other automation tools, for example, Puppet which is agent based and is a bit complex to configure.

This explains why Ansible is growing in popularity each day and becoming the goto automation tool for many enterprises.

> In science fiction, the word **Ansible** refers to a hypothetical device that enable uses to communicate instantaneously across great distances; that is, a faster-than-light communication device. Now you know where Ansible got its name inspiration.

Creating Your Playground

To make the most out of this book and follow along painlessly, I advise you to use the same setup that I am using.

I created one RHEL 8 (Red Hat Enterprise Linux 8) virtual machine that would serve as the control node. A control node is, as its name suggest is basically a server that is used to control other remote hosts (managed nodes).

I created three CentOS 8 virtual machine for managed nodes: node1, node2, and node3. I also created an Ubuntu 18.04 for the last managed node.

Figure 1 summarizes the whole setup:

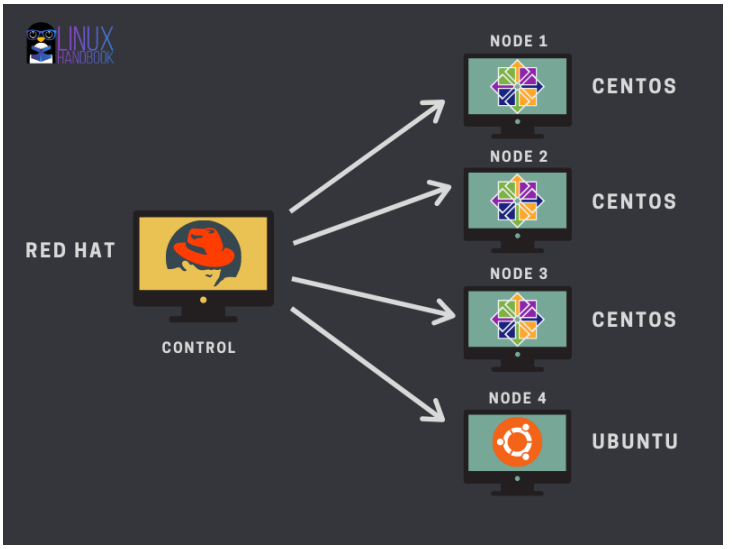

Figure 1: Our Ansible Setup

I don't have enough resources on my computer to create all these virtual machines without my computer crashing. So, I have used Microsoft Azure for all the virtual machines I have created as you can see in Figure 2:

5 items

Name ↑↓	Type ↑↓	Status	Resource group ↑↓
control	Virtual machine	Running	ansible
node1	Virtual machine	Running	ansible
node2	Virtual machine	Running	ansible
node3	Virtual machine	Running	ansible
node4	Virtual machine	Running	ansible

Figure 2: Virtual Machines created on Azure

Needless to say, that you are free any other cloud service providers like AWS, GCP (Google Cloud Platform), Linode, Digital Ocean, UpCloud etc. You are also free to create virtual machines locally on your computer using a virtualization software like VirtualBox, VMware Player, or Fusion (MacOS).

Installing Ansible

Ansible relies on SSH and Python to do all the automation magic and so you only need to install Ansible on the control node and make sure that OpenSSH and Python is installed on both the control and the managed nodes.

Long story short, you don't need to have Ansible installed on the managed nodes!

Now, I am going to show you how to install Ansible on a variety of systems.

Installing Ansible on RHEL 8

I will first start by showing you how to install Ansible on a RHEL 8 system as this book is primarily targeting all the RHCE exam objectives.

Login to your control node and switch to the root user:

```
[elliot@control ~]$ sudo su -
Last login: Tue Oct 20 01:05:00 UTC 2020 on pts/0
```

Checking Linux version information, you can see I am running RHEL 8.2 and I am going to use this as the control node:

```
[root@control ~]# cat /etc/redhat-release
Red Hat Enterprise Linux release 8.2 (Ootpa)
```

To get Ansible installed on a RHEL 8 system, you first have to register your system with the **subscription-manager** command:

```
[root@control ~]# subscription-manager register
Registering to: subscription.rhsm.redhat.com:443/subscription
Username:xxxxx
Password: xxxxx
The system has been registered with ID: 1d8ace59-c140-4f8c-b4bb-b4cd0f4fb811
The registered system name is: control
```

You will be prompted for a username and a password as you can see, if you don't have a Red Hat account, you can create an account and obtain a free trial at the following link: https://access.redhat.com/products/red-hat-enterprise-linux/evaluation

You would then need to attach the new subscription with the following command:

```
[root@control ~]# subscription-manager attach --auto
Installed Product Current Status:
Product Name: Red Hat Enterprise Linux for x86_64
Status: Subscribed
Product Name: Red Hat Enterprise Linux for x86_64 - Extended Update Support
Status: Subscribed
```

Notice that you could have registered and attached the subscription in a single command:

```
subscription-manager register --username=<USER_NAME> --password=<PASSWORD> --auto-attach
```

Now we have access to all the RHEL 8 repositories. You can list all the available Ansible repositories by running the following command:

```
[root@control ~]# yum repolist all | grep ansible
ansible-2-for-rhel-8-x86_64-debug-rpms                Red Hat Ansible E disabled
ansible-2-for-rhel-8-x86_64-rpms                      Red Hat Ansible E disabled
ansible-2-for-rhel-8-x86_64-source-rpms               Red Hat Ansible E disabled
ansible-2.8-for-rhel-8-x86_64-debug-rpms              Red Hat Ansible E disabled
ansible-2.8-for-rhel-8-x86_64-rpms                    Red Hat Ansible E disabled
ansible-2.8-for-rhel-8-x86_64-source-rpms             Red Hat Ansible E disabled
ansible-2.9-for-rhel-8-x86_64-debug-rpms              Red Hat Ansible E disabled
ansible-2.9-for-rhel-8-x86_64-rpms                    Red Hat Ansible E disabled
ansible-2.9-for-rhel-8-x86_64-source-rpms             Red Hat Ansible E disabled
```

Now find the newest Ansible version repository and enable it. At the time of writing this, **ansible-2.9** is the latest version and so I am going to enable the **ansible-2.9-for-rhel-8-x86__64-rpms** with the **yum__config__manager** command as follows:

```
[root@control ~]# yum-config-manager --enable ansible-2.9-for-rhel-8-x86_64-rpms
Updating Subscription Management repositories.
```

You can now verify that the Ansible repository is indeed enabled by listing all the enabled repositories on your system:

```
[root@control ~]# yum repolist enabled
Updating Subscription Management repositories.
repo id                                          repo name
ansible-2.9-for-rhel-8-x86_64-rpms               Red Hat Ansible Engine ...
microsoft-azure-rhel8-eus                        Microsoft Azure RPMs for RHEL8 ...
```

```
rhel-8-for-x86_64-appstream-eus-rhui-rpms       Red Hat Enterprise Linux 8 ...
rhel-8-for-x86_64-appstream-rpms                Red Hat Enterprise Linux 8 ...
rhel-8-for-x86_64-baseos-eus-rhui-rpms          Red Hat Enterprise Linux 8 ...
rhel-8-for-x86_64-baseos-rpms                   Red Hat Enterprise Linux 8 ...
```

All the preliminary work is done. You can now finally install Ansible:

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

After the installation is complete. You can verify that Ansible is indeed installed by running the command:

```
[root@control ~]# ansible --version
ansible 2.9.14
  config file = /etc/ansible/ansible.cfg
  configured module search path = ['/root/.ansible/plugins/modules',
  '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3.6/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 3.6.8 (default, Dec  5 2019, 15:45:45) [GCC 8.3.1 20191121]
```

Awesome! You have now successfully installed Ansible on RHEL 8. I am sure you may be thinking that was a lengthy process!

On the bright side, there will be no internet access on the exam, which means, your control system will come equipped with all the repositories that you will need, and so you won't have to worry about using the subscription manager.

But it's better to be prepared than to be surprised, always remember this!

Installing Ansible on CentOS

On CentOS, Ansible is provided by the EPEL (Extra Package for Enterprise Linux) repository.

You can install and enable the EPEL repo by installing the **epel-release** package as follows:

```
[root@node1 ~]# yum install -y epel-release
```

Now, you can install **ansible**:

```
[root@node1 ~]# yum install -y ansible
```

Keep in mind that we installed Ansible on one of the managed nodes here (node1) only for learning purposes; you only need to install Ansible on the control node.

Installing Ansible on Ubuntu

On Ubuntu, you need to make sure you have the desired Ansible version repository enabled on your system.

You can add and enable **ansible-2.9** ppa repository using the following command:

```
root@node4:~# apt-add-repository --yes --update ppa:ansible/ansible-2.9
```

Finally, you can install Ansible on Ubuntu:

```
root@node4:~# apt-get -y install ansible
```

This takes us to the end of our first chapter. In the next chapter, you are going to learn how to configure Ansible and get to run some really cool Ad-Hoc Ansible commands.

Knowledge Check

If you haven't already done so; Install the latest Ansible version on your RHEL 8 **control** node.

Hint: Enable the Ansible repository first.

Solution to the exercise is provided at the end of the book.

Chapter 2: Running Ad-Hoc Commands

In the first chapter, you got acquainted with Ansible and learned how to install it.

In this chapter, you will learn how to manage static inventory in Ansible. You will also understand various Ansible configuration settings.

Furthermore, you will explore few Ansible modules and you will get to run Ansible Ad-Hoc commands.

Creating an Ansible user

Even though you can use the root user in Ansible to run Ad-Hoc commands and playbooks, it's definitely not recommended and is not considered best practice due to the security risks that can arise by allowing root user ssh access.

For this reason, it's recommended that you create a dedicated Ansible user with sudo privileges (to all commands) on all hosts (control and managed hosts).

Remember, Ansible uses SSH and Python to do all the dirty work behind the scenes and so here are the four steps you would have to follow after installing Ansible:

1. Create a new user on all hosts.
2. Grant sudo privileges to the new user on all nodes.
3. Generate SSH keys for the new user on the control node.
4. Copy the SSH public key to the managed nodes.

So, without further ado, let's start with creating a new user named **elliot** on all hosts:

```
[root@control ~]# useradd -m elliot
[root@node1 ~]# useradd -m elliot
[root@node2 ~]# useradd -m elliot
[root@node3 ~]# useradd -m elliot
[root@node4 ~]# useradd -m elliot
```

After setting elliot's password on all hosts, you can move to step 2; you can grant **elliot** sudo privileges to all commands without password by adding the following line to the **/etc/sudoers** file:

```
[root@control ~]# echo "elliot  ALL=(ALL)  NOPASSWD: ALL" >> /etc/sudoers
[root@node1 ~]# echo "elliot  ALL=(ALL)  NOPASSWD: ALL" >> /etc/sudoers
[root@node2 ~]# echo "elliot  ALL=(ALL)  NOPASSWD: ALL" >> /etc/sudoers
[root@node3 ~]# echo "elliot  ALL=(ALL)  NOPASSWD: ALL" >> /etc/sudoers
[root@node4 ~]# echo "elliot  ALL=(ALL)  NOPASSWD: ALL" >> /etc/sudoers
```

Now, login as user **elliot** on your control node and generate a ssh-key pair:

```
[elliot@control ~]$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/elliot/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/elliot/.ssh/id_rsa.
Your public key has been saved in /home/elliot/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:Xf5bKx0kkBCsCQ/7rc6Kv6CxCRTH2XJajbNvpzel+Ik elliot@control
The key's randomart image is:
+---[RSA 3072]----+
|        .oo .    |
|  . ooo  . o     |
| . = *=.o   o    |
|  o =.o+ . o . . |
| . . .. S . . o  |
|.     .. . . . . |
|.. .   oo.o   o o|
|. = o oo++.  . +.|
| + ..++Eoo.   o. |
+----[SHA256]-----+
```

Finally, you can copy elliot's public ssh key to all managed hosts using the **ssh-copy-id** command as follows:

```
[elliot@control ~]$ ssh-copy-id node1
[elliot@control ~]$ ssh-copy-id node2
[elliot@control ~]$ ssh-copy-id node3
[elliot@control ~]$ ssh-copy-id node4
```

You should now be able to ssh into all managed nodes without being prompted for a password; you will only be asked to enter a ssh passphrase (if you didn't leave it empty, ha-ha).

Building your Ansible inventory

An Ansible inventory file is a basically a file that contains a list of servers, group of servers, or ip addresses that references that hosts that you want to be managed by Ansible (managed nodes).

The **/etc/ansible/hosts** is the default inventory file. I will now show you how you to create your own inventory files in Ansible.

Creating a project directory

You don't want to mess with the **/etc/ansible** directory; you should keep everything in **/etc/ansible** intact and basically just use it as a reference when you are creating inventory files, editing Ansible project configuration files, etc.

Now, let's make a new Ansible project directory named in **/home/elliot** named plays which you will use to store all your Ansible related things (playbooks, inventory files, roles, etc) that you will create from this point onwards:

```
[elliot@control ~]$ mkdir /home/elliot/plays
```

Notice that everything you will create from this point moving forward will be on the control node.

Creating an inventory file

Change to the **/home/elliot/plays** directory and create an inventory file named **myhosts** and add all your managed nodes hostnames so it will end up looking like this:

```
[elliot@control plays]$ cat myhosts
node1
node2
node3
node4
```

You can now run the following ansible command to list all your hosts in the myhosts inventory file:

```
[elliot@control plays]$ ansible all -i myhosts --list-hosts
  hosts (4):
    node1
    node2
    node3
    node4
```

The **-i** option was used to specify the myhosts inventory file. If you omit the **-i** option, Ansible will look for hosts in the **/etc/ansible/hosts** inventory file instead.

Keep in mind that I am using hostnames here and that all the nodes (vms) I have created on Azure are on the same subnet and I don't have to worry about DNS as it's handled by Azure.

If you don't have a working DNS server, you can add your nodes IP address/hostname entries in **/etc/hosts**, Figure 3 is an example:

```
  GNU nano 2.9.8                                          /etc/hosts

127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6

10.0.0.5    node1.linuxhandbook.local  node1
10.0.0.6    node2.linuxhandbook.local  node2
10.0.0.7    node3.linuxhandbook.local  node3
10.0.0.8    node4.linuxhandbook.local  node4
```

Figure 3: /etc/hosts

Creating host groups and subgroups

You can organize your managed hosts into groups and subgroups. For example, you can edit the **myhosts** file to create two groups **test** and **prod** as follows:

```
[elliot@control plays]$ cat myhosts
[test]
node1
node2

[prod]
node3
node4
```

You can list the hosts in the **prod** group by running the following command:

```
[elliot@control plays]$ ansible prod -i myhosts --list-hosts
  hosts (2):
    node3
    node4
```

There are two default groups in Ansible:

1. **all** → contains all the hosts in the inventory
2. **ungrouped** → contains all the hosts that are not a member of any group (aside from all).

Let's add an imaginary host **node5** to the **myhosts** inventory file to demonstrate the **ungrouped** group:

```
[elliot@control plays]$ cat myhosts
node5

[test]
node1
node2

[prod]
node3
node4
```

Notice that I added **node5** to the very beginning (and not the end), otherwise, it would be considered a member of the **prod** group.

Now you can run the following command to list all the **ungrouped** hosts:

```
[elliot@control plays]$ ansible ungrouped -i myhosts --list-hosts
  hosts (1):
    node5
```

You can also create a group (parent) that contains subgroups (children). Take a look at the following example:

```
[elliot@control plays]$ cat myhosts
[web_dev]
node1

[web_prod]
node2

[db_dev]
node3

[db_prod]
node4

[development:children]
```

```
web_dev
db_dev

[production:children]
web_prod
db_prod
```

The **development** group contains all the hosts that are in **web__dev** plus all the members that are in **db__dev**. Similarly, the **production** group contains all the hosts that are in **web__prod** plus all the members that are in **db__prod**.

```
[elliot@control plays]$ ansible development -i myhosts --list-hosts
  hosts (2):
    node1
    node3

[elliot@control plays]$ ansible production -i myhosts --list-hosts
  hosts (2):
    node2
    node4
```

Configuring Ansible

In this section, you will learn about the most important Ansible configuration settings. Throughout the entire book, I will show you other configuration settings as the need arises.

The **/etc/ansible/ansible.cfg** is the default configuration file. However, it's also recommended that you don't mess with **/etc/ansible/ansible.cfg** and just use it a reference. You should create your own Ansible configuration file in your Ansible project directory.

The **ansible --version** command will show you which configuration file you are currently using:

```
[elliot@control plays]$ ansible --version
ansible 2.9.14
  config file = /etc/ansible/ansible.cfg
  configured module search path = ['/home/elliot/.ansible/plugins/modules',
  '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3.6/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 3.6.8 (default, Dec  5 2019, 15:45:45)
```

As you can see from the output, **/etc/ansible/ansible.cfg** is currently in use as you haven't yet created your own **ansible.cfg** file in the project directory.

The **/etc/ansible/ansible.cfg** contains a whole of various Ansible configuration settings and sections:

```
[elliot@control plays]$ wc -l /etc/ansible/ansible.cfg
490 /etc/ansible/ansible.cfg
```

The two most important sections that you need to define in your Ansible configuration file are:

1. [**defaults**]
2. [**privilege_escalation**]

In the [**defaults**] section, here are the most important settings you need to be aware of:

- **inventory** → specifies the path of your inventory file.
- **remote_user** → specifies the user who will connect to the managed hosts and run the playbooks.

- **forks** → specifies the number of host that ansible can manage/process in parallel; default is 5.
- **host_key_checking** → specifies whether you want to turn on/off SSH key host checking; default is True.

In the [**privilege_escalation**] section, you can configure the following settings:

- **become** → specify where to allow/disallow privilege escalation; default is False.
- **become_method** → specify the privilege escalation method; default is sudo.
- **become_user** → specify the user you become through privilege escalation; default is root.
- **become_ask_pass** → specify whether to ask or not ask for privilege escalation password; default is False.

Keep in mind, you don't need to commit any of these settings to memory. They are all documented in **/etc/ansible/ansible.cfg**.

Now create your own **ansible.cfg** configuration file in your Ansible project directory **/home/elliot/plays** and set the following settings (as shown in Figure 4):

```
[defaults]
inventory = myhosts
remote_user = elliot
host_key_checking = false

[privilege_escalation]
become = true
become_method = sudo
become_user = root
become_ack_pass = false
```

Figure 4: /home/elliot/plays/ansible.cfg

Now run the **ansible --version** command one more time; you should see that your new configuration file is now in effect:

```
[elliot@control plays]$ ansible --version
ansible 2.9.14
  config file = /home/elliot/plays/ansible.cfg
  configured module search path = ['/home/elliot/.ansible/plugins/modules',
  '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3.6/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 3.6.8 (default, Dec  5 2019, 15:45:45)
```

Running Ad-Hoc Commands in Ansible

Until this point, you have really just been installing, setting up your environment, and configuring Ansible. *Now, the real fun begins!*

An Ansible ad-hoc command is a great tool that you can use to run a single task on one or more managed nodes. A typical Ansible ad-hoc command follows the general syntax:

ansible host_pattern -m module_name -a "module_options"

The easiest way to understand how Ansible ad-hoc commands work is simply running one! So, go ahead and run the following ad-hoc command:

```
[elliot@control plays]$ ansible node1 -m command -a "uptime"
Enter passphrase for key '/home/elliot/.ssh/id_rsa':
node1 | CHANGED | rc=0 >>
18:53:01 up 5 days, 18:03,  1 user,  load average: 0.00, 0.01, 0.00
```

I was prompted to enter my ssh key passphrase and then the server uptime of **node1** was displayed! Now, check Figure 5 to help you understand each element of the ad-hoc command you just ran:

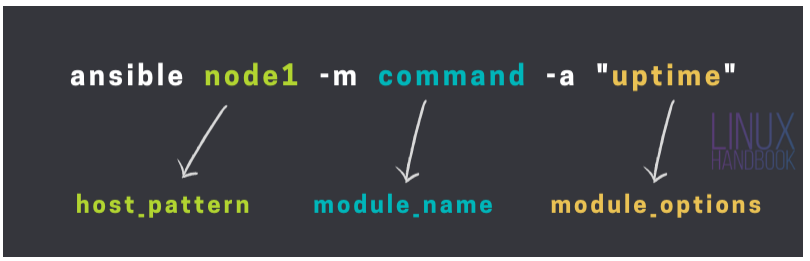

Figure 5: Understanding ad-hoc commands

You would probably have guessed it by now; **ansible modules** are reusable, standalone scripts that can be used by the **Ansible API**, or by the **ansible** or **ansible-playbook** commands.

The **command** module is one of the many modules that Ansible has to offer. You can run the **ansible-doc -l** command to see a list of all the available Ansible modules:

```
[elliot@control plays]$ ansible-doc -l | wc -l
3387
```

Currently, there are 3387 Ansible modules available, and they increase every day! You can pass any command way you wish to run as an option to the Ansible **command** module.

If you don't have an empty ssh key passphrase (just like me); then you would have to run **ssh-agent** to avoid the unnecessary headache of being prompted for a passphrase every single time Ansible try to access your managed nodes:

```
[elliot@control plays]$ eval `ssh-agent`
Agent pid 218750
[elliot@control plays]$ ssh-add
Enter passphrase for /home/elliot/.ssh/id_rsa:
Identity added: /home/elliot/.ssh/id_rsa (elliot@control)
```

Testing Connectivity

You may want to test if Ansible can connect to all your managed nodes before getting into the more serious tasks; for this, you can use the **ping** module and specify all your managed hosts as follows:

```
[elliot@control plays]$ ansible all -m ping
node4 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
node3 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
node1 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
node2 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
```

```
    "ping": "pong"
}
```

As you can see with all the SUCCESS messages in the output. Notice that the Ansible **ping** module doesn't need any options. Some Ansible modules require options and some does not, just like the case with Linux commands.

Ansible Modules Documentation

If someone asked me what you like the most about Ansible; I would quickly say it's the documentation. Ansible is so very well documented and it's all from the comfort of your own terminal.

If you want to how to use a specific Ansible module, then you can run **ansible-doc** followed by the module name.

For example, you can view the description of the **ping** module and how to use it by running:

```
[elliot@control plays]$ ansible-doc ping
```

This will open up the **ping** module documentation page as shown in Figure 6:

```
> PING    (/usr/lib/python3.6/site-packages/ansible/modules/system/ping.py)

        A trivial test module, this module always returns `pong' on
        successful contact. It does not make sense in playbooks, but
        it is useful from `/usr/bin/ansible' to verify the ability to
        login and that a usable Python is configured. This is NOT ICMP
        ping, this is just a trivial test module that requires Python
        on the remote-node. For Windows targets, use the [win_ping]
        module instead. For Network targets, use the [net_ping] module
        instead.

  * This module is maintained by The Ansible Core Team
OPTIONS (= is mandatory):

- data
        Data to return for the `ping' return value.
        If this parameter is set to `crash', the module will cause an
        exception.
        [Default: pong]
        type: str


SEE ALSO:
:
EXAMPLES:

# Test we can logon to 'webservers' and execute python with json lib.
# ansible webservers -m ping
```

Figure 6: ansible-doc **ping**

When reading modules documentation, pay especial attention to see if any option is prefixed by the equal sign (=). In this case, it's a mandatory option that you must include.

Also, if you scroll all the way down, you can see some examples of how to run the ad-hoc commands or Ansible playbooks (that we will discuss later).

Command vs. Shell vs. Raw Modules

There are three Ansible modules that people often confuse with one another; these are:

1. **command**
2. **shell**
3. **raw**

Those three modules achieve the same purpose; they run commands on the managed nodes. But there are key differences that separates the three modules.

You can't use piping or redirection with the **command** module. For example, the following ad-hoc command will result in an error:

```
[elliot@control plays]$ ansible node2 -m command -a "lscpu | head -n 5"
node2 | FAILED | rc=1 >>
lscpu: invalid option -- 'n'
Try 'lscpu --help' for more information.non-zero return code
```

That's because the **command** module doesn't support pipes or redirection. You can use the **shell** module instead if you want to use pipes or redirection. Run the same command again, but this time, use the **shell** module instead:

```
[elliot@control plays]$ ansible node2 -m shell -a "lscpu | head -n 5"
node2 | CHANGED | rc=0 >>
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              1
On-line CPU(s) list: 0
```

Works like a charm! It successfully displayed the first five lines of the **lscpu** command output on **node2**.

Ansible uses SSH and Python scripts behind the scenes to do all the magic. Now, the **raw** module just uses SSH and bypasses the Ansible module subsystem. This way, the **raw** module would successfully work on the managed node even if python is not installed (on the managed node).

I tampered with my python binaries on node4 (please don't do that yourself) so I can mimic a scenario of what will happen if you run the **shell** or **command** module on a node that doesn't have python installed:

```
root@node4:/usr/bin# mkdir hide
root@node4:/usr/bin# mv python* hide/
```

Now check what will happen if I run an Ansible ad-hoc with the **shell** or **command** module targeting **node4**:

```
[elliot@control plays]$ ansible node4 -m shell -a "whoami"
node4 | FAILED! => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "module_stderr": "Shared connection to node4 closed.\r\n",
    "module_stdout": "/bin/sh: 1: /usr/bin/python: not found\r\n",
}


[elliot@control plays]$ ansible node4 -m command -a "cat /etc/os-release"
node4 | FAILED! => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "module_stderr": "Shared connection to node4 closed.\r\n",
    "module_stdout": "/bin/sh: 1: /usr/bin/python: not found\r\n",
    "msg": "The module failed to execute correctly, you probably need
    to set the interpreter.\nSee stdout/stderr for the exact error",
    "rc": 127
}
```

I get errors! Now I will try to achieve the same task; but this time, I will use the **raw** module:

```
[elliot@control plays]$ ansible node4 -m raw -a "cat /etc/os-release"
node4 | CHANGED | rc=0 >>
NAME="Ubuntu"
VERSION="18.04.5 LTS (Bionic Beaver)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 18.04.5 LTS"
VERSION_ID="18.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
VERSION_CODENAME=bionic
```

```
UBUNTU_CODENAME=bionic
Shared connection to node4 closed.
```

As you can see, the **raw** module was the only module out of three modules to carry out the task successfully. Now I will go back fix the mess that I did on **node4**:

```
root@node4:/usr/bin/hide# mv * ..
```

Figure 7 summarizes the different use cases for the three modules:

	command	shell	raw
Run simple commands	✓	✓	✓
Run commands with pipes or redirection	✗	✓	✓
Run commands without python	✗	✗	✓

Figure 7: **command** vs. **shell** vs. **raw** modules

Alright! This takes us to the end of the second chapter. In the next chapter, you are going to learn how to create and run Ansible playbooks.

Knowledge Check

Create a bash script named **adhoc-cmds.sh** that will run few ad-hoc commands on your managed nodes.

Your script will accomplish the following three tasks:

1. Installs python on **node4**.
2. Displays the **uptime** on all the managed nodes.
3. Create the file **/tmp/hello.txt** with the contents "Hello, Friend!" on **node3**.

Hint: Use the **shell** or **copy** modules to create the **/tmp/hello.txt** file.
Solution to the exercise is provided at the end of the book.

Chapter 3: Ansible Playbooks

In the previous chapter, you learned how to use Ansible ad-hoc commands to run a single task on your managed hosts. In this chapter, you will learn how to automate multiple tasks on your managed hosts by creating and running Ansible playbooks.

To better understand the differences between Ansible ad-hoc command and Ansible playbooks; you can think of Ansible ad-hoc commands as Linux commands and playbooks as bash scripts.

Ansible ad-hoc commands are ideal to perform tasks that are not executed frequently such us getting servers uptime, retrieving system information, etc.]]
On the other hand, Ansible playbooks are ideal to automate complex tasks like system patches, application deployments, firewall configurations, user management, etc.

Please note that I have included all the playbooks, scripts, and files in this book in this GitHub repository: https://github.com/kabary/rhce8

Creating your first Ansible playbook

Playbooks are written in **YAML** (Yet Another Markup Language) format. If you don't know **YAML**; I have included the most important YAML syntax rules in Figure 8 so you can easily follow along with all the playbook examples:

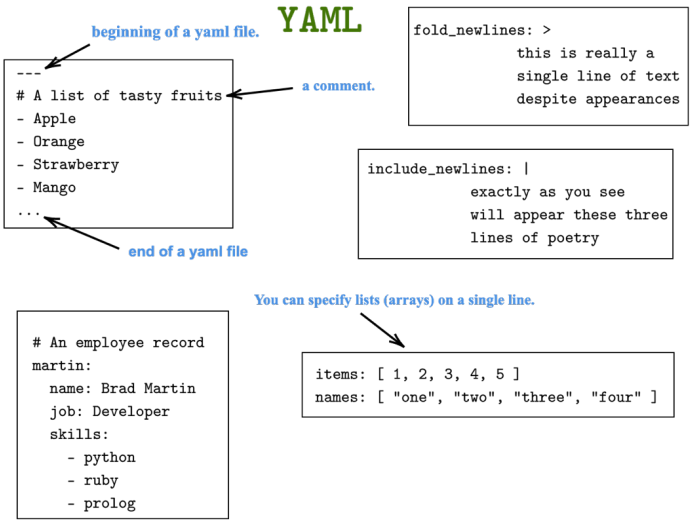

Figure 8: YAML Basics

You should also be aware that **YAML** files also must have either a **.yaml** or **.yml** extension. I personally prefer **.yml** because it's less typing, and I am lazy.

Also, **YAML** is indentation sensitive. A two-spaces indentation is the recommended indentation to use in **YAML**; however, **YAML** will follow whatever indentation system a file uses as long as it's consistent.

It is beyond annoying to keep hitting two spaces in your keyboard and so do yourself a favor and include the following line in the **/.vimrc** file:

```
autocmd FileType yaml setlocal ai ts=2 sw=2 et
```

This will convert the tabs into two spaces whenever you are working on a **YAML** file.

Now let's create your first playbook. In your project directory, create a file named **first-playbook.yml** that has the following contents:

```
[elliot@control plays]$ cat first-playbook.yml
---
- name: first play
  hosts: all
  tasks:
    - name: create a new file
      file:
        path: /tmp/foo.conf
        mode: 0664
        owner: elliot
        state: touch
```

This playbook will run on all hosts and uses the **file** module to create a file named **/tmp/foo.conf**; you also set the **mode: 0664** and **owner: elliot** module options to specify the file permissions and the owner of the file. Finally, you set the **state: touch** option to make sure the file gets created if it doesn't already exist.

To run the playbook, you can use the **ansible-playbook** command followed by the playbook filename:

```
[elliot@control plays]$ ansible-playbook first-playbook.yml

PLAY [first play] ************

TASK [Gathering Facts] *****************
ok: [node4]
ok: [node3]
ok: [node1]
ok: [node2]
```

```
TASK [create a new file] ***************
changed: [node4]
changed: [node3]
changed: [node1]
changed: [node2]

PLAY RECAP *****************
node1: ok=2    changed=1    unreachable=0    failed=0    skipped=0
node2: ok=2    changed=1    unreachable=0    failed=0    skipped=0
node3: ok=2    changed=1    unreachable=0    failed=0    skipped=0
node4: ok=2    changed=1    unreachable=0    failed=0    skipped=0
```

The output of the playbook run is pretty self-explanatory. For now, pay special attention to **changed=1** in the **PLAY RECAP** summary which means that one change was executed successfully on the managed node.

Let's run the following ad-hoc command to verify that the file **/tmp/foo.conf** is indeed created on all the managed hosts:

```
[elliot@control plays]$ ansible all -m command -a "ls -l /tmp/foo.conf"
node4 | CHANGED | rc=0 >>
-rw-rw-r-- 1 elliot root 0 Oct 25 03:20 /tmp/foo.conf
node1 | CHANGED | rc=0 >>
-rw-rw-r--. 1 elliot root 0 Oct 25 03:20 /tmp/foo.conf
node2 | CHANGED | rc=0 >>
-rw-rw-r--. 1 elliot root 0 Oct 25 03:20 /tmp/foo.conf
node3 | CHANGED | rc=0 >>
-rw-rw-r--. 1 elliot root 0 Oct 25 03:20 /tmp/foo.conf
```

Notice that the following ad-hoc command will accomplish the same task as **first-playbook.yml** playbook:

```
ansible all -m file -a "path=/tmp/foo.conf mode=0664 owner=elliot state=touch"
```

To read more about the **file** module, check its Ansible documentation page:

```
[elliot@control plays]$ ansible-doc file
```

Running multiple plays with Ansible Playbook

You have only created one play that contains one task in **first-playbook.yml** playbook. A playbook can contain multiple plays and each play can in turn contains multiple tasks.

Let's create a playbook named **multiple-plays.yml** that has the following content:

```
[elliot@control plays]$ cat multiple-plays.yml
---
- name: first play
  hosts: all
  tasks:
    - name: install tmux
      package:
        name: tmux
        state: present

    - name: create an archive
      archive:
        path: /var/log
        dest: /tmp/logs.tar.gz
        format: gz

- name: second play
  hosts: node4
  tasks:
    - name: install git
      apt:
        name: git
        state: present
```

This playbook has two plays:

1. **first play** (contains two tasks) → runs on all hosts.
2. **second play** (contains one task) → only runs on **node4**.

Notice that I used the **package** module on the first play as it is the generic module to manage packages and it autodetects the default package manager on the managed nodes. I used the **apt** module on the second play as I am only running it on an Ubuntu host (**node4**).

The **yum** and **dnf** modules also exists and they work on **CentOS** and **RHEL** systems.

I also used the **archive** module to create a gzip compressed archive **/tmp/logs.tar.gz** that contains all the files in the **/var/log** directory.

Go ahead and run the **multiple-plays.yml** playbook:

```
[elliot@control plays]$ ansible-playbook multiple-plays.yml

PLAY [first play] *************

TASK [install tmux] ******************
changed: [node4]
changed: [node2]
changed: [node3]
changed: [node1]

TASK [create an archive] *************
changed: [node2]
changed: [node3]
changed: [node1]
changed: [node4]

PLAY [second play] **************

TASK [install git] **************
changed: [node4]

PLAY RECAP **********************
node1: ok=2    changed=2    unreachable=0    failed=0    skipped=0
node2: ok=2    changed=2    unreachable=0    failed=0    skipped=0
node3: ok=2    changed=2    unreachable=0    failed=0    skipped=0
node4: ok=3    changed=3    unreachable=0    failed=0    skipped=0
```

Everything looks good. You can quickly check if the **/tmp/logs.tar.gz** archive exists on all nodes by running the following ad-hoc command:

```
[elliot@control plays]$ ansible all -m command -a "file -s /tmp/logs.tar.gz"
node4 | CHANGED | rc=0 >>
/tmp/logs.tar.gz: gzip compressed data, was "/tmp/logs.tar",
last modified: Sun Oct 25 04:40:46 2020, max compression
node1 | CHANGED | rc=0 >>
/tmp/logs.tar.gz: gzip compressed data, was "/tmp/logs.tar",
last modified: Sun Oct 25 04:40:47 2020, max compression, original size 107458560
node3 | CHANGED | rc=0 >>
/tmp/logs.tar.gz: gzip compressed data, was "/tmp/logs.tar",
last modified: Sun Oct 25 04:40:47 2020, max compression, original size 75560960
node2 | CHANGED | rc=0 >>
/tmp/logs.tar.gz: gzip compressed data, was "/tmp/logs.tar",
last modified: Sun Oct 25 04:40:47 2020, max compression, original size 52326400
```

I also recommend you check the following Ansible documentation pages and check the examples section:

```
[elliot@control plays]$ ansible-doc package
[elliot@control plays]$ ansible-doc archive
[elliot@control plays]$ ansible-doc apt
[elliot@control plays]$ ansible-doc yum
```

Verifying your playbooks

Although I have already shown you the steps to run Ansible playbooks, it is always a good idea to verify your playbook before actually running it. This ensures that your playbook is free of potential errors.

You can use the **--syntax-check** option to check if your playbook has syntax errors:

```
[elliot@control plays]$ ansible-playbook --syntax-check first-playbook.yml
playbook: first-playbook.yml
```

You may also want to use the **--check** option to do a dry run of your playbook before actually running the playbook:

```
[elliot@control plays]$ ansible-playbook --check first-playbook.yml

PLAY [first play] ********************

TASK [Gathering Facts] ********************
ok: [node4]
ok: [node3]
ok: [node1]
ok: [node2]

TASK [create a new file] ************
ok: [node4]
ok: [node1]
ok: [node2]
ok: [node3]

PLAY RECAP ***************
node1: ok=2    changed=0    unreachable=0    failed=0    skipped=0
node2: ok=2    changed=0    unreachable=0    failed=0    skipped=0
node3: ok=2    changed=0    unreachable=0    failed=0    skipped=0
node4: ok=2    changed=0    unreachable=0    failed=0    skipped=0
```

Notice that doing a dry run of the playbook will not commit any change on the managed nodes.

You can use the **--list-hosts** option to list the hosts of each play in your playbook:

```
[elliot@control plays]$ ansible-playbook --list-hosts multiple-plays.yml

playbook: multiple-plays.yml
```

```
  play #1 (all): first play TAGS: []
    pattern: ['all']
    hosts (4):
      node4
      node2
      node1
      node3

  play #2 (node4): second play TAGS: []
    pattern: ['node4']
    hosts (1):
      node4
```

You can also list the tasks of each play in your playbook by using the **--list-tasks** option:

```
[elliot@control plays]$ ansible-playbook --list-tasks multiple-plays.yml

playbook: multiple-plays.yml

  play #1 (all): first play TAGS: []
    tasks:
      install tmux TAGS: []
      create an archive TAGS: []

  play #2 (node4): second play TAGS: []
    tasks:
      install git TAGS: []
```

You can also check the **ansible-playbook** man page for a comprehensive list of options.

Re-using tasks and playbooks

You may find yourself writing multiple playbooks that all share a common list of tasks. In this case, it's better to create a file that contains a list of all the common tasks and then you can reuse them in your playbooks.

To demonstrate, let's create a file named **group-tasks.yml** that contains the following tasks:

```
[elliot@control plays]$ cat group-tasks.yml
- name: create developers group
  group:
    name: developers
- name: create security group
  group:
    name: security
- name: create finance group
  group:
    name: finance
```

Now you can use the **import_tasks** module to run all the tasks in **group-tasks.yml** in your first playbook as follows:

```
[elliot@control plays]$ cat first-playbook.yml
---
- name: first play
  hosts: all
  tasks:
    - name: create a new file
      file:
        path: /tmp/foo.conf
        mode: 0664
        owner: elliot
        state: touch

    - name: create groups
      import_tasks: group-tasks.yml
```

You can also use the **import_playbook** module to reuse an entire playbook. For example, you can create a new playbook named **reuse-playbook.yml** that has the following content:

```
[elliot@control plays]$ cat reuse-playbook.yml
---
- name: Reusing playbooks
```

```
  hosts: all
  tasks:
    - name: Reboot the servers
      reboot:
        msg: Server is rebooting ...

- name: Run first playbook
  import_playbook: first-playbook.yml
```

Also notice that you can only import a playbook on a new play level; that is, you can't import a play within another play.

You can also use the **include** module to reuse tasks and playbooks. For example, you can replace the **import_playbook** statement with the include statement as follows:

```
[elliot@control plays]$ cat reuse-playbook.yml
---
- name: Reusing playbooks
  hosts: all
  tasks:
    - name: Reboot the servers
      reboot:
        msg: Server is rebooting ...

- name: Run first playbook
  include: first-playbook.yml
```

The only difference is that the **import** statements are pre-processed at the time playbooks are parsed. On the other hand, **include** statements are processed as they are encountered during the execution of the playbook. So, in summary, **import** is static while **include** is dynamic.

Running selective tasks and plays

You can choose not to run a whole playbook and instead may want to run specific task(s) or play(s) in a playbook. To do this, you can use tags.

For example, you can tag the **install git** task in the **multiple-plays.yml** playbook as follows:

```
[elliot@control plays]$ cat multiple-plays.yml
---
- name: first play
  hosts: all
  tasks:
    - name: install tmux
      package:
        name: tmux
        state: present

    - name: create an archive
      archive:
        path: /var/log
        dest: /tmp/logs.tar.gz
        format: gz

- name: second play
  hosts: node4
  tasks:
    - name: install git
      apt:
        name: git
        state: present
      tags: git
```

Now you can use the **--tags** option followed by the tag name **git** to only run the **install git** task:

```
[elliot@control plays]$ ansible-playbook multiple-plays.yml --tags git

PLAY [first play] ****************

PLAY [second play] *************************

TASK [install git] ****************
ok: [node4]

PLAY RECAP ********************
node4: ok=1    changed=0    unreachable=0    failed=0    skipped=0
```

As you can see, the first two plays were skipped and only the **install git** task did run. You can also see **changed=0** in the **PLAY RECAP** and that's because **git** is already installed on **node4**.

You can also apply tags to a play in a similar fashion.

Alright! This takes us to the end of this chapter. In the next chapter, you are going to learn how to work with Ansible variables, facts, and registers.

Knowledge Check

Create a playbook named **lab3.yml** that will accomplish the following tasks:

1. Installs the **nmap** package on all managed nodes.
2. Create a **bzip2** compressed archive **/tmp/home.tar.xz** of all the files in **/home**.

Hint: Use the **package** module for the first task.

Solution to the exercise is provided at the end of the book.

Chapter 4: Ansible Variables, Facts, and Registers

There will always be a lot of variances across your managed systems. For this reason, you need to learn how to work with Ansible variables.

In this chapter, you will learn how to define and reference variables Ansible. You will also learn how to use Ansible facts to retrieve information on your managed nodes.

Furthermore, you will also learn how to use registers to capture task output.

Working with variables

Let's start with variables first. Keep in mind that everything will be written in YAML format.

Defining and referencing variables

You can use the **vars** keyword to define variables directly inside a playbook.

For example, you can define a **fav_color** variable and set its value to yellow as follows:

```
---
- name: Working with variables
  hosts: all
  vars:
    fav_color: yellow
```

Now how do you use (reference) the **fav_color** variable? Ansible uses the **Jinja2** templating system to work with variables. You will learn all about **Jinja2** in chapter 7, but for now you just need to know the very basics.

To get the value of the **fav_color** variable; you need to surround it by a pair of curly brackets as follows:

```
My favorite color is {{ fav_color }}
```

Notice that if your variable is the first element (or only element) in the line, then using quotes is mandatory as follows:

```
"{{ fav_color }} is my favorite color."
```

Now let's write a playbook named **variables-playbook.yml** that puts all this together:

```
[elliot@control plays]$ cat variables-playbook.yml
---
- name: Working with variables
  hosts: node1
  vars:
    fav_color: yellow
  tasks:
    - name: Show me fav_color value
      debug:
        msg: My favorite color is {{ fav_color }}.
```

I have used the **debug** module along with the msg module option to print the value of the **fav_color** variable.

Now run the playbook and you shall see your favorite color displayed as follows:

```
[elliot@control plays]$ ansible-playbook variables-playbook.yml

PLAY [Working with variables] ************

TASK [Gathering Facts] **************
ok: [node1]

TASK [Show me fav_color value] **********
ok: [node1] => {
    "msg": "My favorite color is yellow."
}

PLAY RECAP ********************
node1: ok=2    changed=0    unreachable=0    failed=0    skipped=0
```

Creating lists and dictionaries

You can also use lists and dictionaries to define multivalued variables. For example, you may define a list named **port_nums** and set its value as follows:

```
vars:
  port_nums: [21,22,23,25,80,443]
```

You could have also used the following way to define **port_nums** which is equivalent:

```
vars:
  port_nums:
    - 21
    - 22
    - 23
    - 25
    - 80
    - 443
```

You can print all the values in **port_nums** as follows:

```
All the ports {{ port_nums }}
```

You can also access a specific list element:

```
First port is {{ port_nums[0] }}
```

Notice that you use an index (position) to access list elements.

You can also define a dictionary named **users** as follows:

```
vars:
  users:
    bob:
      username: bob
      uid: 1122
      shell: /bin/bash
    lisa:
      username: lisa
      uid: 2233
      shell: /bin/sh
```

There are two different ways you can use to access dictionary elements:

- **dict_name['key']** → users['bob']['shell']

- **dict_name.key** → users.bob.shell

Notice that you use a key to access dictionary elements.

Now you can edit the **variables-playbook.yml** playbook to show lists and dictionaries in action:

```
[elliot@control plays]$ cat variables-playbook.yml
---
- name: Working with variables
  hosts: node1
  vars:
    port_nums: [21,22,23,25,80,443]

    users:
      bob:
        username: bob
        uid: 1122
        shell: /bin/bash
      lisa:
        username: lisa
        uid: 2233
        shell: /bin/sh
  tasks:
    - name: Show 2nd item in port_nums
      debug:
        msg: SSH port is {{ port_nums[1] }}

    - name: Show the uid of bob
      debug:
        msg: UID of bob is {{ users.bob.uid }}
```

You can now run the playbook to display the second element in **port_nums** and show bob's uid:

```
[elliot@control plays]$ ansible-playbook variables-playbook.yml

PLAY [Working with variables] ********************

TASK [Show 2nd item in port_nums] ************
ok: [node1] => {
    "msg": "SSH port is 22"
}

TASK [Show the uid of bob] *************
ok: [node1] => {
    "msg": "UID of bob is 1122"
}
```

Including external variables

Just like you can import (or include) tasks in a playbook. You can do the same thing with variables as well. That is, in a playbook, you can include variables defined in an external file.

To demonstrate, let's create a file named **myvars.yml** that contains our **port_nums** list and **users** dictionary:

```
[elliot@control plays]$ cat myvars.yml
---
port_nums: [21,22,23,25,80,443]

users:
  bob:
    username: bob
    uid: 1122
    shell: /bin/bash
  lisa:
    username: lisa
    uid: 2233
    shell: /bin/sh
```

Now you can use the **vars_files** keyword in your **variables-playbook.yml** to include the variables in **myvars.yml** as follows:

```
[elliot@control plays]$ cat variables-playbook.yml
---
- name: Working with variables
  hosts: node1
  vars_files: myvars.yml
  tasks:
    - name: Show 2nd item in port_nums
      debug:
        msg: SSH port is {{ port_nums[1] }}

    - name: Show the uid of bob
      debug:
        msg: UID of bob is {{ users.bob.uid }}
```

Keep in mind that **vars_files** preprocesses and load the variables right at the start of the playbook. You can also use the **include_vars** module to dynamically load your variables in your playbook:

```
[elliot@control plays]$ cat variables-playbook.yml
---
- name: Working with variables
  hosts: node1
  tasks:
    - name: Load the variables
      include_vars: myvars.yml

    - name: Show 2nd item in port_nums
      debug:
        msg: SSH port is {{ port_nums[1] }}
```

Getting user input

You can use the **vars_prompt** keyword to prompt the user to set a variable's value at runtime.

For example, the following **greet.yml** playbook asks the running user to enter his name and then displays a personalized greeting message:

```
[elliot@control plays]$ cat greet.yml
---
- name: Greet the user
  hosts: node1
  vars_prompt:
     - name: username
       prompt: What's your name?
       private: no

  tasks:
    - name: Greet the user
      debug:
        msg: Hello {{ username }}
```

Notice I used **private: no** so that you can see your input on the screen as you type it; by default, it's hidden.

Now run the playbook and enter your name:

```
[elliot@control plays]$ ansible-playbook greet.yml
What's your name?: Elliot

PLAY [Greet the user] ******************
```

```
TASK [Greet the user] *************************
ok: [node1] => {
    "msg": "Hello Elliot"
}
```

Setting host and group variables

You can set variables that are specific to your managed hosts. By doing so, you can create much more efficient playbooks as you don't need to write repeated tasks for different nodes or groups.

To demonstrate, edit your inventory file so that your managed nodes are grouped in the following three groups:

```
[elliot@control plays]$ cat myhosts
[proxy]
node1

[webservers]
node2
node3

[dbservers]
node4
```

Now to create variables that are specific to your managed nodes; first, you need to create a directory named **host_vars**. Then inside **host_vars**, you can create variables files with filenames that corresponds to your nodes hostname as follows:

```
[elliot@control plays]$ mkdir host_vars
[elliot@control plays]$ echo "message: I am a Proxy Server" >> host_vars/node1.yml
[elliot@control plays]$ echo "message: I am a Web Server" >> host_vars/node2.yml
[elliot@control plays]$ echo "message: I am a Web Server" >> host_vars/node3.yml
[elliot@control plays]$ echo "message: I am a Database Server" >> host_vars/node4.yml
```

Now let's create a playbook named **motd.yml** that demonstrates how **host_vars** work:

```
[elliot@control plays]$ cat motd.yml
---
- name: Set motd on all nodes
```

```
hosts: all
tasks:
  - name: Set motd = value of message variable.
    copy:
      content: "{{ message }}"
      dest: /etc/motd
```

I used the **copy** module to copy the contents of the message variable onto the **/etc/motd** file on all nodes. Now after running the playbook; you should see that the contents of **/etc/motd** has been updated on all nodes with the corresponding **message** value:

```
[elliot@control plays]$ ansible all -m command -a "cat /etc/motd"
node1 | CHANGED | rc=0 >>
I am a Proxy Server
node2 | CHANGED | rc=0 >>
I am a Web Server
node3 | CHANGED | rc=0 >>
I am a Web Server
node4 | CHANGED | rc=0 >>
I am a Database Server
```

Awesome! Similarly, you can use create a **group_vars** directory and then include all group related variables in a filename that corresponds to the group name as follows:

```
[elliot@control plays]$ mkdir group_vars
[elliot@control plays]$ echo "pkg: squid" >> group_vars/proxy
[elliot@control plays]$ echo "pkg: httpd" >> group_vars/webservers
[elliot@control plays]$ echo "pkg: mariadb-server" >> group_vars/dbservers
```

I will let you create a playbook that runs on all nodes; each node will install the package that is set in the node's corresponding group **pkg** variable.

Understanding variable precedence

As you have seen so far; Ansible variables can be set at different scopes (or levels).

If the same variable is set at different levels; the most specific level gets precedence. For example, a variable that is set on a **play level** takes precedence over the same variable set on a **host level** (**host_vars**).

Furthermore, a variable that is set on the **command line** using the **--extra-vars** takes the highest precedence, that is, it will overwrite anything else.

To demonstrate, let's create a playbook named **variable-precedence.yml** that contains the following content:

```
[elliot@control plays]$ cat variable-precedence.yml
---
- name: Understanding Variable Precedence
  hosts: node1
  vars:
    fav_distro: "Ubuntu"
  tasks:
    - name: Show value of fav_distro
      debug:
        msg: Favorite distro is {{ fav_distro }}
```

Now let's run the playbook while using the **-e** (**--extra-vars**) option to set the value of the **fav_distro** variable to "CentOS" from the command line:

```
[elliot@control plays]$ ansible-playbook variable-precedence.yml -e "fav_distro=CentOS"

PLAY [Understanding Variable Precedence] ***************************************

TASK [Show value of fav_distro] ************************************************
ok: [node1] => {
    "msg": "Favorite distro is CentOS"
}
```

Notice how the command line's **fav_distro** value "CentOS" took precedence over the play's **fav_distro** value "Ubuntu".

Gathering and showing facts

You can retrieve or discover variables that contain information about your managed hosts. These variables are called **facts** and Ansible uses the **setup** module to gather these facts. The IP address on one of your managed nodes is an example of a fact.

You can run the following ad-hoc command to gather and show all the facts on **node1**:

```
[elliot@control plays]$ ansible node1 -m setup

node1 | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "10.0.0.5"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::20d:3aff:fe0c:54aa"
        ],
        "ansible_apparmor": {
            "status": "disabled"
        },
        "ansible_architecture": "x86_64",
        "ansible_bios_date": "06/02/2017",
        "ansible_bios_version": "090007",
        "ansible_cmdline": {
            "BOOT_IMAGE": "(hd0,gpt1)/vmlinuz-4.18.0-193.6.3.el8_2.x86_64",
            "console": "ttyS0,115200n8",
            "earlyprintk": "ttyS0,115200",
            "ro": true,
            "root": "UUID=6785aa9a-3d19-43ba-a189-f73916b0c827",
            "rootdelay": "300",
            "scsi_mod.use_blk_mq": "y"
        },
        "ansible_default_ipv4": {
            "address": "10.0.0.5",
            "alias": "eth0",
            "broadcast": "10.0.0.255",
            "gateway": "10.0.0.1",
            "interface": "eth0",
            "macaddress": "00:0d:3a:0c:54:aa",
```

This is only a fraction of all the facts related to **node1** that you are going to see displayed on your terminal. Notice how the facts are stored in dictionaries or lists and they all belong to the **ansible_facts** dictionary.

By default, the **setup** module is automatically called by playbooks to do the facts discovery. You may have noticed that facts discovery happens right at the start when you run any of your playbooks:

```
[elliot@control plays]$ ansible-playbook motd.yml

PLAY [Set motd on all nodes] ***************************************************

TASK [Gathering Facts] *********************************************************
ok: [node4]
ok: [node3]
ok: [node2]
ok: [node1]
.
.
.
```

You can turn off facts gathering by setting **gather_facts** boolean to false right in your play header as follows:

```
[elliot@control plays]$ cat motd.yml
---
- name: Set motd on all nodes
  gather_facts: false
  hosts: all
  tasks:
    - name: Set motd = value of message variable.
      copy:
        content: "{{ message }}"
        dest: /etc/motd
```

If you run the **motd.yaml** playbook again; it will skip facts gathering:

```
[elliot@control plays]$ ansible-playbook motd.yml

PLAY [Set motd on all nodes] ******************

TASK [Set motd = value of message variable.] ******************
```

The same way you show a variable's value; you can also use to show a fact's value. The following **show-facts.yml** playbook displays the value of few facts on **node1**:

```
[elliot@control plays]$ cat show-facts.yml
---
- name: show some facts
  hosts: node1
```

```
  tasks:
    - name: display node1 ipv4 address
      debug:
        msg: IPv4 address is {{ ansible_facts.default_ipv4.address }}

    - name: display node1 fqdn
      debug:
        msg: FQDN is {{ ansible_facts.fqdn }}

    - name: display node1 OS distribution
      debug:
        msg: OS Distro is {{ ansible_facts.distribution }}
```

Now run the playbook to display the facts values:

```
[elliot@control plays]$ ansible-playbook show-facts.yml

PLAY [show some facts] ******

TASK [Gathering Facts] *******
ok: [node1]

TASK [display node1 ipv4 address] *******
ok: [node1] => {
    "msg": "IPv4 address is 10.0.0.5"
}

TASK [display node1 fqdn] ********
ok: [node1] => {
    "msg": "FQDN is node1.linuxhandbook.local"
}

TASK [display node1 OS distribution] ******
ok: [node1] => {
    "msg": "OS Distro is CentOS"
}

PLAY RECAP **********
node1:  ok=4    changed=0    unreachable=0    failed=0    skipped=0
```

Creating customs facts

You may want to create your own custom facts. To do this, you can either use the **set_fact** module to add temporarily facts or the **/etc/ansible/facts.d** directory to add permanent facts on your managed nodes.

I am going to show you how to add permanent facts to your managed nodes. It's a three steps process:

1. Create a facts file on your control node.
2. Create the **/etc/ansible/facts.d** directory on your managed node(s).
3. Copy your facts file (step 1) from the control node to your managed node(s).

So, first, let's create a **cool.fact** file on your control node that includes some cool facts:

```
[elliot@control plays]$ cat cool.fact
[fun]
kiwi=fruit
matrix=movie
octupus='8 legs'
```

Notice that your facts filename must have the **.fact** extension.

For the second step, you are going to use the file module to create and the **/etc/ansible/facts.d** directory on the managed node(s). And lastly for the third step, you are going to use the **copy** module to copy **cool.fact** file from the control node to the managed node(s).

The following **custom-facts.yml** playbook combines step 2 and 3:

```
[elliot@control plays]$ cat custom-facts.yml
---
- name: Adding custom facts to node1
  hosts: node1
  tasks:
    - name: Create the facts.d directory
      file:
        path: /etc/ansible/facts.d
        owner: elliot
        mode: 775
        state: directory

    - name: Copy cool.fact to the facts.d directory
      copy:
        src: cool.fact
        dest: /etc/ansible/facts.d
```

Now run the playbook:

```
[elliot@control plays]$ ansible-playbook custom-facts.yml

PLAY [Adding custom facts to node1] **********************************

TASK [Gathering Facts] ***************************
ok: [node1]

TASK [Create the facts.d directory] ******************************
changed: [node1]

TASK [Copy cool.fact to the facts.d directory] ***********************
changed: [node1]
```

The **cool** facts are now permanently part of **node1** facts; you can verify with the following ad-hoc command:

```
[elliot@control plays]$ ansible node1 -m setup -a "filter=ansible_local"

"ansible_local": {
            "cool": {
                "fun": {
                    "kiwi": "fruit",
                    "matrix": "movie",
                    "octupus": "'8 legs'"
                }
            }
        }
```

You can now display the octopus's fact in a playbook as follows:

```
An octopus has {{ ansible_local.cool.fun.octupus }}
```

Capturing output with registers

Some tasks will not show any output when running a playbook. For instance, running commands on your managed nodes using the **command**, **shell**, or **raw** modules will not display any output when running a playbook.

You can use a **register** to capture the output of a task and save it to a variable. This allows you to make use of a task output elsewhere in a playbook by simply addressing the registered variable.

The following **register-playbook.yml** shows you how to capture a task output in a registered variable and later display its content:

```
[elliot@control plays]$ cat register-playbook.yml
---
- name: Register Playbook
  hosts: proxy
  tasks:
    - name: Run a command
      command: uptime
      register: server_uptime

    - name: Inspect the server_uptime variable
      debug:
        var: server_uptime

    - name: Show the server uptime
      debug:
        msg: "{{ server_uptime.stdout }}"
```

The playbook starts by running the **uptime** command on the **proxy** group hosts (**node1**) and registers the command output to the **server_uptime** variable.

Then, you use the **debug** module along with the **var** module option to inspect the **server_uptime** variable. Notice, that you don't need to surround the variable with curly brackets here.

Finally, the last task in the playbook shows the output (**stdout**) of the registered variable **server_uptime**.

Run the playbook to see all this in action:

```
[elliot@control plays]$ ansible-playbook register-playbook.yml

PLAY [Register Playbook Showcase] **************

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

```
ok: [node1]

TASK [Run a command] *****************
changed: [node1]

TASK [Inspect the server_uptime variable] ***********************
ok: [node1] => {
    "server_uptime": {
        "changed": true,
        "cmd": [
            "uptime"
        ],
        "delta": "0:00:00.004221",
        "end": "2020-10-29 05:04:36.646712",
        "failed": false,
        "rc": 0,
        "start": "2020-10-29 05:04:36.642491",
        "stderr": "",
        "stderr_lines": [],
        "stdout": " 05:04:36 up 3 days,  6:56,  1 user,
        load average: 0.24, 0.07, 0.02",
        "stdout_lines": [
            " 05:04:36 up 3 days,  6:56,  1 user,  load average: 0.24, 0.07, 0.02"
        ]
    }
}

TASK [Show the server uptime] **********
ok: [node1] => {
    "msg": " 05:04:36 up 3 days,  6:56,  1 user,  load average: 0.24, 0.07, 0.02"
}

PLAY RECAP ************
node1: ok=4    changed=1    unreachable=0    failed=0    skipped=0
```

Notice how the registered variable **server_uptime** is actually a dictionary that contains a lot of other keys beside the **stdout** key. It also contains other keys like **rc** (return code), **start** (time when command run), **end** (time when command finished), **stderr** (any errors), etc.

Alright! This takes us to the end of this chapter. In the next chapter, you are going to learn how to use loops in Ansible.

Knowledge Check

Create a playbook named **lab4.yml** that will accomplish the following tasks:

1. Displays the output of the "**free -h**" command on all managed nodes.
2. Displays the IPv4 address of all managed nodes.

Hint: Use Ansible Registers for the first task.

Solution to the exercise is provided at the end of the book.

Chapter 5: Ansible Loops

You may sometimes want to repeat a task multiple times. For example, you may want to create multiple users, start/stop multiple services, or change ownership on several files on your managed hosts.

In this chapter, you will learn how to use Ansible loops to repeat a task multiple times without having to rewrite the whole task over and over again.

Looping over lists

Ansible uses the **loop** keyword to iterate over the elements of a list. To demonstrate, let's create a very simple playbook named **print-list.yml** that shows you how to print the elements in a list:

```
[elliot@control plays]$ cat print-list.yml
---
- name: print list
  hosts: node1
  vars:
    prime: [2,3,5,7,11]
  tasks:
    - name: Show first five prime numbers
      debug:
        msg: "{{ item }}"
      loop: "{{ prime }}"
```

Notice that I use the **item** variable with Ansible loops. The task would run five times which is equal to the number of elements in the **prime** list.

On the first run, the **item** variable will be set to first element in the prime array (2). On the second run, the **item** variable will be set to the second element in the prime array (3) and so on.

Go ahead and run the playbook to see all the elements of the **prime** list displayed:

```
[elliot@control plays]$ ansible-playbook print-list.yml

PLAY [print list] ************

TASK [Gathering Facts] ***********
ok: [node1]

TASK [Show first five prime numbers] ******
ok: [node1] => (item=2) => {
    "msg": 2
}
```

```
ok: [node1] => (item=3) => {
    "msg": 3
}
ok: [node1] => (item=5) => {
    "msg": 5
}
ok: [node1] => (item=7) => {
    "msg": 7
}
ok: [node1] => (item=11) => {
    "msg": 11
}

PLAY RECAP *************
node1: ok=2    changed=0    unreachable=0    failed=0
```

Now let's apply loops to a real life application. For example, you can create an **add-users.yml** playbook that would add multiple users on all the hosts in the **dbservers** host group:

```
[elliot@control plays]$ cat add-users.yml
---
- name: Add multiple users
  hosts: dbservers
  vars:
    dbusers:
      - username: brad
        pass: pass1
      - username: david
        pass: pass2
      - username: jason
        pass: pass3
  tasks:
    - name: Add users
      user:
        name: "{{ item.username }}"
        password: "{{ item.pass | password_hash('sha512') }}"
      loop: "{{ dbusers }}"
```

I first created a **dbusers** list which is basically a list of hashes/dictionaries. I then used the **user** module along with a loop to add the users and set the passwords for all users in the **dbusers** list.

Notice that I also used the dotted notation **item.username** and **item.pass** to access the keys values inside the hashes/dictionaries of the **dbusers** list.

It is also worth noting that I used the **password_hash('sha512')** filter to encrypt the user passwords with the **sha512** hashing algorithm as the **user** module wouldn't allow setting unencrypted user passwords.

> **RHCE Exam Tip**: You will have access to the **docs.ansible.com** page on your exam. It a very valuable resource, especially under the "*Frequently Asked Questions*" section; you will find numerous How-to questions with answers and explanations.

Now let's run the **add-users.yml** playbook:

```
[elliot@control plays]$ ansible-playbook add-users.yml

PLAY [Add multiple users] ***********************************

TASK [Gathering Facts] ***************************************
ok: [node4]

TASK [Add users] *********************************************************
changed: [node4] => (item={'username': 'brad', 'pass': 'pass1'})
changed: [node4] => (item={'username': 'david', 'pass': 'pass2'})
changed: [node4] => (item={'username': 'jason', 'pass': 'pass3'})

PLAY RECAP ***************************************************
node4: ok=2    changed=1    unreachable=0    failed=0    skipped=0
```

You can verify that the three users are added by following up with an Ansible ad-hoc command:

```
[elliot@control plays]$ ansible dbservers -m command -a "tail -3 /etc/passwd"
node4 | CHANGED | rc=0 >>
brad:x:1001:1004::/home/brad:/bin/bash
david:x:1002:1005::/home/david:/bin/bash
jason:x:1003:1006::/home/jason:/bin/bash
```

Looping over dictionaries

You can only use loops with lists. You will get an error if you try to loop over a dictionary.

For example, if you run the following **print-dict.yml** playbook:

```
[elliot@control plays]$ cat print-dict.yml
---
- name: Print Dictionary
  hosts: node1
  vars:
    employee:
      name: "Elliot Alderson"
      title: "Penetration Tester"
      company: "Linux Handbook"
  tasks:
    - name: Print employee dictionary
      debug:
        msg: "{{ item }}"
      loop: "{{ employee }}"
```

You will get the following error:

```
[elliot@control plays]$ ansible-playbook print-dict.yml

PLAY [Print Dictionary] ******************

TASK [Gathering Facts] ************************
ok: [node1]

TASK [Print employee dictionary] ******************
fatal: [node1]: FAILED! => {"msg": "Invalid data passed to 'loop', it requires a list,
got this instead: {'name': 'Elliot Alderson', 'title': 'Penetration Tester', 'company':
'Linux Handbook'}. Hint: If you passed a list/dict of just one element,
try adding wantlist=True to your lookup invocation or use q/query instead of lookup."}
```

As you can see in the error message, it explicitly says that it requires a list.

To fix this error; you can use the **dict2items** filter to convert a dictionary to a list. So, in the **print-dict.yml** playbook; edit the line:

```
loop: "{{ employee }}"
```

and apply the **dict2items** filter as follows:

```
loop: "{{ employee | dict2items }}"
```

Then run the playbook again:

```
[elliot@control plays]$ ansible-playbook print-dict.yml

PLAY [Print Dictionary] ****************

TASK [Gathering Facts] ******************
ok: [node1]

TASK [Print employee dictionary] ********************
ok: [node1] => (item={'key': 'name', 'value': 'Elliot Alderson'}) => {
    "msg": {
        "key": "name",
        "value": "Elliot Alderson"
    }
}
ok: [node1] => (item={'key': 'title', 'value': 'Penetration Tester'}) => {
    "msg": {
        "key": "title",
        "value": "Penetration Tester"
    }
}
ok: [node1] => (item={'key': 'company', 'value': 'Linux Handbook'}) => {
    "msg": {
        "key": "company",
        "value": "Linux Handbook"
    }
}

PLAY RECAP ***************
node1: ok=2    changed=0    unreachable=0    failed=0    skipped=0
```

Success! The key/value pairs of the **employee** dictionary were displayed.

Looping over a range of numbers

You can use the **range()** function along with the **list** filter to loop over a range of numbers.

For example, the following task would print all the numbers from 0 → 9:

```
- name: Range Loop
  debug:
    msg: "{{ item }}"
  loop: "{{ range(10) | list }}"
```

You can also start your range at a number other than zero. For example, the following task would print all the numbers from 5 → 14:

```
- name: Range Loop
  debug:
    msg: "{{ item }}"
  loop: "{{ range(5,15) | list }}"
```

By default, the stride is set to 1. However, you can set a different stride.

For example, the following task would print all the even IP addresses in the 192.168.1.x subnet:

```
- name: Range Loop
  debug:
    msg: 192.168.1.{{ item }}
  loop: "{{ range(0,256,2) | list }}"
```

Where start=0, end<256, and stride=2.

Looping over inventories

You can use the Ansible built-in **groups** variable to loop over all your inventory hosts or just a subset of it. For instance, to loop over all your inventory hosts; you can use:

```
loop: "{{ groups['all'] }}"
```

If you want to loop over all the hosts in the **webservers** group, you can use:

```
loop: "{{ groups['webservers'] }}"
```

To see how this works in a playbook; take a look at the following **loop-inventory.yml** playbook:

```
[elliot@control plays]$ cat loop-inventory.yml
---
- name: Loop over Inventory
  hosts: node1
  tasks:
    - name: Ping all hosts
      command: ping -c 1 "{{ item }}"
      loop: "{{ groups['all'] }}"
```

This playbook tests if **node1** is able to ping all other hosts in your inventory. Go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook loop-inventory.yml

PLAY [Loop over Inventory] ****************

TASK [Gathering Facts] **************
ok: [node1]

TASK [Ping all hosts] ***************
changed: [node1] => (item=node1)
changed: [node1] => (item=node2)
changed: [node1] => (item=node3)
changed: [node1] => (item=node4)

PLAY RECAP *****************
node1: ok=2    changed=1    unreachable=0    failed=0    skipped=0
```

If you get any errors; this would mean that your managed hosts are not able to ping (reach) each other.

Pausing within loops

You may want to pause for a certain amount of time between each loop iteration. To do this, you can use the pause directive along with the **loop_control** keyword.

To demonstrate, let's create a **countdown.yml** playbook that would simply do a ten seconds countdown before displaying the message "Happy Birthday!" on the screen:

```
[elliot@control plays]$ cat countdown.yml
---
- name: Happy Birthday Playbook
  hosts: node1
  tasks:
  - name: Ten seconds countdown
    debug:
      msg: "{{ 10 - item }} seconds remaining ..."
    loop: "{{ range(10) | list }}"
    loop_control:
      pause: 1

  - name: Display Happy Birthday
    debug:
      msg: "Happy Birthday!"
```

Go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook countdown.yml

PLAY [Happy Birthday Playbook] ****************

TASK [Gathering Facts] *************
ok: [node1]

TASK [Wait for ten seconds] *****************
ok: [node1] => (item=0) => {
    "msg": "10 seconds remaining ..."
}
ok: [node1] => (item=1) => {
    "msg": "9 seconds remaining ..."
}
ok: [node1] => (item=2) => {
    "msg": "8 seconds remaining ..."
```

```
}
ok: [node1] => (item=3) => {
    "msg": "7 seconds remaining ..."
}
ok: [node1] => (item=4) => {
    "msg": "6 seconds remaining ..."
}
ok: [node1] => (item=5) => {
    "msg": "5 seconds remaining ..."
}
ok: [node1] => (item=6) => {
    "msg": "4 seconds remaining ..."
}
ok: [node1] => (item=7) => {
    "msg": "3 seconds remaining ..."
}
ok: [node1] => (item=8) => {
    "msg": "2 seconds remaining ..."
}
ok: [node1] => (item=9) => {
    "msg": "1 seconds remaining ..."
}

TASK [Display Happy Birthday] ***************************
ok: [node1] => {
    "msg": "Happy Birthday!"
}

PLAY RECAP **********************************************************************
node1: ok=3    changed=0    unreachable=0    failed=0    skipped=0
```

I will end this chapter on the happy birthday note! In the next chapter, you are going to learn how to add decision-making skills to your Ansible playbooks.

Knowledge Check

Create a playbook named **lab5.yml** that will only run on the **localhost** and display all the even numbers that are between 20 and 40 (Inclusive).

Hint: Use the **range()** function.

Solution to the exercise is provided at the end of the book.

Chapter 6: Decision Making in Ansible

In this chapter, you will learn how to add decision making skills to your Ansible playbooks.

You will learn to:

- Use when statements to run tasks conditionally.
- Use block statements to implement exception handling.
- Use Ansible handlers to trigger tasks upon change.

Choosing When to Run Tasks

In this section, you will learn how to put conditions on when to run a certain task in Ansible playbooks.

Using when with facts

You can use **when** conditional statements to run a task only when a certain condition is true. To demonstrate, create a new playbook named **ubuntu-server.yml** that has the following content:

```
[elliot@control plays]$ cat ubuntu-server.yml
---
- name: Using when with facts
  hosts: all
  tasks:
    - name: Detect Ubuntu Servers
      debug:
        msg: "This is an Ubuntu Server."
      when:  ansible_facts['distribution'] == "Ubuntu"
```

Now go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook ubuntu-servers.yml

PLAY [Using when with facts] ********************

TASK [Gathering Facts] ***********************
ok: [node4]
ok: [node1]
ok: [node3]
ok: [node2]

TASK [Detect Ubuntu Servers] ***************
skipping: [node1]
```

```
skipping: [node2]
skipping: [node3]
ok: [node4] => {
    "msg": "This is an Ubuntu Server."
}

PLAY RECAP ********************
node1: ok=1    changed=0    unreachable=0    failed=0    skipped=1

node2: ok=1    changed=0    unreachable=0    failed=0    skipped=1
node3: ok=1    changed=0    unreachable=0    failed=0    skipped=1
node4: ok=2    changed=0    unreachable=0    failed=0    skipped=0
```

Notice how I used the Ansible fact **ansible_facts['distribution']** in the **when** condition to test which nodes are running Ubuntu. Also, notice that you don't need to surround variables with curly brackets in **when** conditional tests.

In the playbook output, notice how **TASK [Detect Ubuntu Servers]** skipped the first three nodes as they are all running CentOS and only ran on **node4** as it is running Ubuntu.

Using when with registers

You can also use **when** conditionals with registered variables. For example, the following playbook **centos-servers.yml** will reveal which nodes are running CentOS:

```
[elliot@control plays]$ cat centos-servers.yml
---
- name: Using when with registers
  hosts: all
  tasks:
    - name: Save the contents of /etc/os-release
      command: cat /etc/os-release
      register: os_release

    - name: Detect CentOS Servers
      debug:
        msg: "Running CentOS ..."
      when: os_release.stdout.find('CentOS') != -1
```

The playbook first starts by saving the contents of the **/etc/os-release** file into the **os_release** register variable. Then the second tasks displays the message "Running CentOS …" only if the word 'CentOS' is found in **os_release** standard output.

Go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook centos-servers.yml

PLAY [Using when with registers] ***************

TASK [Gathering Facts] *******************
ok: [node4]
ok: [node1]
ok: [node3]
ok: [node2]

TASK [Save the contents of /etc/os-release] ***************
changed: [node4]
changed: [node1]
changed: [node2]
changed: [node3]

TASK [Detect CentOS Servers] *************
ok: [node1] => {
    "msg": "Running CentOS ..."
}
ok: [node2] => {
    "msg": "Running CentOS ..."
}
ok: [node3] => {
    "msg": "Running CentOS ..."
}
skipping: [node4]

PLAY RECAP ***********************
node1: ok=3    changed=1    unreachable=0    failed=0    skipped=0
node2: ok=3    changed=1    unreachable=0    failed=0    skipped=0
node3: ok=3    changed=1    unreachable=0    failed=0    skipped=0
node4: ok=2    changed=1    unreachable=0    failed=0    skipped=1
```

Notice how **TASK [Detect CentOS Servers]** only ran on the first three nodes and skipped **node4** (Ubuntu).

Testing multiple conditions with when

You can also test multiple conditions at once by using the logical operators. For example, the following **reboot-centos8.yml** playbook uses the logical **and** operator to reboot servers that are running CentOS version 8:

```
[elliot@control plays]$ cat reboot-centos8.yml
---
- name: Reboot Servers
  hosts: all
  tasks:
    - name: Reboot CentOS 8 servers
```

```
    reboot:
      msg: "Server is rebooting ..."
    when: >
      ansible_facts['distribution'] == "CentOS" and
      ansible_facts['distribution_major_version'] == "8"
```

You can also use the logical **or** operator to run a task if any of the conditions is true. For example, the following task would reboot servers that are running either **CentOS** or **RedHat**:

```
tasks:
  - name: Reboot CentOS and RedHat Servers
    reboot:
      msg: "Server is rebooting ..."
    when: >
      ansible_facts['distribution'] == "CentOS" or
      ansible_facts['distribution'] == "RedHat"
```

Using when with loops

If you combine a **when** conditional with a **loop**, Ansible would **test** the condition for each item in the loop separately.

For example, the following **print-even.yml** playbook will print all the even numbers in the **range(1,11)**:

```
[elliot@control plays]$ cat print-even.yml
---
- name: Print Some Numbers
  hosts: node1
  tasks:
    - name: Print Even Numbers
      debug:
        msg: Number {{ item }} is Even.
      loop: "{{ range(1,11) | list }}"
      when: item % 2 == 0
```

Go ahead and run the playbook to see the list of all even numbers in the **range(1,11)**:

```
[elliot@control plays]$ ansible-playbook print-even.yml

PLAY [Print Some Numbers] **************
```

```
TASK [Gathering Facts] *****************
ok: [node1]

TASK [Print Even Numbers] ******************
skipping: [node1] => (item=1)
ok: [node1] => (item=2) => {
    "msg": "Number 2 is Even."
}
skipping: [node1] => (item=3)
ok: [node1] => (item=4) => {
    "msg": "Number 4 is Even."
}
skipping: [node1] => (item=5)
ok: [node1] => (item=6) => {
    "msg": "Number 6 is Even."
}
skipping: [node1] => (item=7)
ok: [node1] => (item=8) => {
    "msg": "Number 8 is Even."
}
skipping: [node1] => (item=9)
ok: [node1] => (item=10) => {
    "msg": "Number 10 is Even."
}

PLAY RECAP **************
node1: ok=2    changed=0    unreachable=0    failed=0    skipped=0
```

Using when with variables

You can also use **when** conditional statements with your own defined variables. Keep in mind that conditionals require boolean inputs; that is, a test must evaluate to true to trigger the condition and so, you need to use the **bool** filter with non-boolean variables.

To demonstrate, take a look at the following **isfree.yml** playbook:

```
[elliot@control plays]$ cat isfree.yml
---
- name:
  hosts: node1
  vars:
    weekend: true
    on_call: "no"
  tasks:
    - name: Run if "weekend" is true and "on_call" is false
      debug:
```

```
      msg: "You are free!"
    when: weekend and not on_call | bool
```

Notice that I used the **bool** filter here to convert the **on_call** value to its boolean equivalent (no →false).

Also, you should be well aware that **not false** is **true** and so the whole condition will evaluate to **true** in this case; you are free!

You can also test to see whether a variable has been set or not; for example, the following task will only run if the **car** variable is defined:

```
tasks:
  - name: Run only if you got a car
    debug:
      msg: "Let's go on a road trip ..."
    when: car is defined
```

The following task uses the **fail** module to fail if the **keys** variable is undefined:

```
tasks:
  - name: Fail if you got no keys
    fail:
      msg: "This play require some keys"
    when: keys is undefined
```

Handling Exceptions with Blocks

Now let's talk about handling exceptions in Ansible.

Grouping tasks with blocks

You can use **blocks** to group related tasks together. To demonstrate, take a look at the following **install-apache.yml** playbook:

```
[elliot@control plays]$ cat install-apache.yml
---
- name: Install and start Apache Play
  hosts: webservers
  tasks:
    - name: Install and start Apache
      block:
         - name: Install httpd
           yum:
             name: httpd
             state: latest

         - name: Start and enable httpd
           service:
             name: httpd
             state: started
             enabled: yes

    - name: This task is outside the block
      debug:
        msg: "I am outside the block now ..."
```

The playbook runs on the **webservers** group hosts and has one block with the name **Install and start Apache** that includes two tasks:

1. Install httpd
2. Start and enable httpd

The first task **Install httpd** uses the **yum** module to install the httpd apache package. The second task **Start and enable httpd** uses the **service** module to start and enabled httpd to start on boot.

Notice that the playbook has a third task that doesn't belong to the **Install and start Apache** block.

Now go ahead and run the playbook to install and start httpd on the **webservers** nodes:

```
[elliot@control plays]$ ansible-playbook install-apache.yml

PLAY [Install and start Apache Play] **************

TASK [Gathering Facts] **************
ok: [node3]
ok: [node2]

TASK [Install httpd] **************
changed: [node2]
changed: [node3]

TASK [Start and enable httpd] *****************
changed: [node3]
changed: [node2]

TASK [This task is outside the block] ******************
ok: [node2] => {
    "msg": "I am outside the block now ..."
}
ok: [node3] => {
    "msg": "I am outside the block now ..."
}

PLAY RECAP **********
node2: ok=4    changed=2    unreachable=0    failed=0    skipped=0
node3: ok=4    changed=2    unreachable=0    failed=0    skipped=0
```

You can also follow up with an ad-hoc command to verify that **httpd** is indeed up and running:

```
[elliot@control plays]$ ansible webservers -m command -a "systemctl status httpd"
node3 | CHANGED | rc=0 >>
 httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service;
   enabled; vendor preset: disabled)
   Active: active (running) since Tue 2020-11-03 19:35:13 UTC; 1min 37s ago
     Docs: man:httpd.service(8)
Main PID: 47122 (httpd)
   Status: "Running, listening on: port 80"
    Tasks: 213 (limit: 11935)
   Memory: 25.1M
   CGroup: /system.slice/httpd.service
           47122 /usr/sbin/httpd -DFOREGROUND
           47123 /usr/sbin/httpd -DFOREGROUND
           47124 /usr/sbin/httpd -DFOREGROUND
           47125 /usr/sbin/httpd -DFOREGROUND
           47126 /usr/sbin/httpd -DFOREGROUND
```

```
Nov 03 19:35:13 node3 systemd[1]: Starting The Apache HTTP Server...
Nov 03 19:35:13 node3 systemd[1]: Started The Apache HTTP Server.
Nov 03 19:35:13 node3 httpd[47122]: Server configured, listening on: port 80
node2 | CHANGED | rc=0 >>
 httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service;
   enabled; vendor preset: disabled)
   Active: active (running) since Tue 2020-11-03 19:35:13 UTC; 1min 37s ago
     Docs: man:httpd.service(8)
 Main PID: 43695 (httpd)
   Status: "Running, listening on: port 80"
    Tasks: 213 (limit: 11935)
   Memory: 25.1M
   CGroup: /system.slice/httpd.service
           43695 /usr/sbin/httpd -DFOREGROUND
           43696 /usr/sbin/httpd -DFOREGROUND
           43697 /usr/sbin/httpd -DFOREGROUND
           43698 /usr/sbin/httpd -DFOREGROUND
           43699 /usr/sbin/httpd -DFOREGROUND

Nov 03 19:35:13 node2 systemd[1]: Starting The Apache HTTP Server...
Nov 03 19:35:13 node2 systemd[1]: Started The Apache HTTP Server.
Nov 03 19:35:13 node2 httpd[43695]: Server configured, listening on: port 80
```

Handling failure with Blocks

You can also use blocks to handle task errors by using the **rescue** and **always** sections. This is pretty much similar to exception handling in programming languages like the **try-catch** in Java or **try-except** in Python.

You can use the rescue section to include all the tasks that you want to run in case one or more tasks in the block has failed.

To demonstrate, let's take a look at the following example:

```
tasks:
  - name: Handling error example
    block:
      - name: run a command
        command: uptime

      - name: run a bad command
        command: blabla

      - name: This task will not run
        debug:
          msg: "I never run because the above task failed."
```

```
    rescue:
      - name: Runs when the block failed
        debug:
          msg: "Block failed; let's try to fix it here ..."
```

Notice how the second task in the block **run a bad command** generates an error and in turn the third task in the block never gets a chance to run. The tasks inside the **rescue** section will run because the second task in the block has failed.

You can also use **ignore_errors: yes** to ensure that Ansible continue executing the tasks in the playbook even if a task has failed:

```
tasks:
  - name: Handling error example
    block:
      - name: run a command
        command: uptime

      - name: run a bad command
        command: blabla
        ignore_errors: yes

      - name: This task will run
        debug:
          msg: "I run because the above task errors were ignored."

    rescue:
      - name: This will not run
        debug:
          msg: "Errors were ignored! ... not going to run."
```

Notice that in this example, you ignored the errors in the second task **run a bad command** in the block and that's why the third task was able to run. Also, the **rescue** section will not run as you ignored the error in the second task in the block.

You can also add an **always** section to a block. Tasks in the **always** section will always run regardless whether the block has failed or not.

To demonstrate, take a look at the following **handle-errors.yml** playbook that has all the three sections (**block**, **rescue**, **always**) of a block:

```
[elliot@control plays]$ cat handle-errors.yml
---
- name: Handling Errors with Blocks
  hosts: node1
```

```
  tasks:
    - name: Handling Errors Example
      block:
        - name: run a command
          command: uptime

        - name: run a bad command
          command: blabla

        - name: This task will not run
          debug:
            msg: "I never run because the task above has failed!"

      rescue:
        - name: Runs when the block fails
          debug:
            msg: "Block failed! let's try to fix it here ..."

      always:
        - name: This will always run
          debug:
            msg: "Whether the block has failed or not ... I will always run!"
```

Go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook handle-errors.yml

PLAY [Handling Errors with Blocks] ********************

TASK [Gathering Facts] ************************
ok: [node1]

TASK [run a command] ******************************
changed: [node1]

TASK [run a bad command] ******************
fatal: [node1]: FAILED! => {"changed": false, "cmd": "blabla", "msg":
"[Errno 2] No such file or directory: b'blabla': b'blabla'", "rc": 2}

TASK [Runs when the block fails] ******************
ok: [node1] => {
    "msg": "Block failed! let's try to fix it here ..."
}

TASK [This will always run] ***************
ok: [node1] => {
    "msg": "Whether the block has failed or not ... I will always run!"
}
```

```
PLAY RECAP *********************************************************************
node1: ok=4    changed=1    unreachable=0    failed=0    skipped=0
```

As you can see; the **rescue** section did run as the second task in the block has failed and you didn't ignore the errors. Also, the **always** section did (and will always) run.

Running Tasks upon Change with Handlers

Now let's demonstrate how you can use handlers to run tasks upon change in Ansible.

Running your first handler

You can use handlers to trigger tasks upon a change on your managed nodes. To demonstrate, take a look at the following **handler-example.yml** playbook:

```
[elliot@control plays]$ cat handler-example.yml
---
- name: Simple Handler Example
  hosts: node1
  tasks:
    - name: Create engineers group
      group:
        name: engineers
      notify: add elliot

    - name: Another task in the play
      debug:
        msg: "I am just another task."

  handlers:
    - name: add elliot
      user:
        name: elliot
        groups: engineers
        append: yes
```

The first task **Create engineers group** creates the engineers group and also notifies the **add elliot** handler.

Let's run the playbook to see what happens:

```
[elliot@control plays]$ ansible-playbook handler-example.yml

PLAY [Simple Handler Example] *****************
```

```
TASK [Gathering Facts] ************************
ok: [node1]

TASK [Create engineers group] *********************
changed: [node1]

TASK [Another task in the play] ******************
ok: [node1] => {
    "msg": "I am just another task."
}

RUNNING HANDLER [add elliot] ***********************
changed: [node1]

PLAY RECAP **************************
node1: ok=4    changed=2    unreachable=0    failed=0    skipped=0
```

Notice that creating the engineers caused a change on **node1** and as a result triggered the **add elliot** handler.

You can also run a quick ad-hoc command to verify that user **elliot** is indeed a member of the **engineers** group:

```
[elliot@control plays]$ ansible node1 -m command -a "id elliot"
node1 | CHANGED | rc=0 >>
uid=1000(elliot) gid=1000(elliot) groups=1000(elliot),4(adm),190(systemd-journal),
1004(engineers)
```

Ansible playbooks and modules are idempotent which means that if a change in configuration occurred on the managed nodes; it will not redo it again!

To fully understand the concept of Ansible's idempotency; run the **handler-example.yml** playbook one more time:

```
[elliot@control plays]$ ansible-playbook handler-example.yml

PLAY [Simple Handler Example] ************************

TASK [Gathering Facts] ********************
ok: [node1]

TASK [Create engineers group] ******************
ok: [node1]

TASK [Another task in the play] ********************
```

```
ok: [node1] => {
    "msg": "I am just another task."
}

PLAY RECAP ******************
node1: ok=3    changed=0    unreachable=0    failed=0    skipped=0
```

As you can see; the **Create engineers group** task didn't not cause or report a change this time because the **engineers** group already exists on **node1** and as a result; the add **elliot handler** did not run.

Controlling when to report a change

You can use the **changed_when** keyword to control when a task should report a change. To demonstrate, take a look at the following **control-change.yml** playbook:

```
[elliot@control plays]$ cat control-change.yml
---
- name: Control Change
  hosts: node1
  tasks:
    - name: Run the date command
      command: date
      notify: handler1

    - name: Run the uptime command
      command: uptime

  handlers:
     - name: handler1
       debug:
         msg: "I can handle dates"
```

Notice how the first task **Run the date command** triggers **handler1**. Now go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook control-change.yml

PLAY [Control Change] ************************

TASK [Gathering Facts] ************************
ok: [node1]

TASK [Run the date command] ********************
changed: [node1]
```

```
TASK [Run the uptime command] ******************
changed: [node1]

RUNNING HANDLER [handler1] ******************
ok: [node1] => {
    "msg": "I can handle dates"
}

PLAY RECAP **********************
node1: ok=4    changed=2    unreachable=0    failed=0    skipped=0
```

Both tasks **Run the date command** and **Run the uptime command** reported changes and **handler1** was triggered. You can argue that running **date** and **uptime** commands don't really change anything on the managed node and you are totally right!

Now let's edit the playbook to stop the **Run the date command** task from reporting changes:

```
[elliot@control plays]$ cat control-change.yml
---
- name: Control Change
  hosts: node1
  tasks:
    - name: Run the date command
      command: date
      notify: handler1
      changed_when: false

    - name: Run the uptime command
      command: uptime

  handlers:
     - name: handler1
       debug:
         msg: "I can handle dates"
```

Now run the playbook again:

```
[elliot@control plays]$ ansible-playbook control-change.yml

PLAY [Control Change] ********************

TASK [Gathering Facts] ***********************
ok: [node1]
```

```
TASK [Run the date command] *********************
ok: [node1]

TASK [Run the uptime command] *************
changed: [node1]

PLAY RECAP ************************
node1: ok=3    changed=1    unreachable=0    failed=0    skipped=0
```

As you can see, the **Run the date command** task didn't report a change this time and as a result, **handler1** was not triggered.

Configuring services with handlers

Handlers are especially useful when you are editing services configurations with Ansible. That's because you only want to restart a service when there is a change in its service configuration.

To demonstrate, take a look at the following **configure-ssh.yml** playbook:

```
[elliot@control plays]$ cat configure-ssh.yml
---
- name: Configure SSH
  hosts: all
  tasks:
     - name: Edit SSH Configuration
       blockinfile:
         path: /etc/ssh/sshd_config
         block: |
            MaxAuthTries 4
            Banner /etc/motd
            X11Forwarding no
       notify: restart ssh

  handlers:
    - name: restart ssh
      service:
        name: sshd
        state: restarted
```

Notice I used the **blockinfile** module to insert multiple lines of text into the **/etc/ssh/sshd_config** configuration file. The **Edit SSH Configuration** task also triggers the **restart ssh** handler upon change.

Go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook configure-ssh.yml

PLAY [Configure SSH] ********************

TASK [Gathering Facts] *******************
ok: [node4]
ok: [node3]
ok: [node1]
ok: [node2]

TASK [Edit SSH Configuration] ***********************
changed: [node4]
changed: [node2]
changed: [node3]
changed: [node1]

RUNNING HANDLER [restart ssh] ***********************
changed: [node4]
changed: [node3]
changed: [node2]
changed: [node1]

PLAY RECAP ***********************
node1: ok=3    changed=2    unreachable=0    failed=0    skipped=0
node2: ok=3    changed=2    unreachable=0    failed=0    skipped=0
node3: ok=3    changed=2    unreachable=0    failed=0    skipped=0
node4: ok=3    changed=2    unreachable=0    failed=0    skipped=0
```

Everything looks good! Now let's quickly take a look the last few lines in the **/etc/ssh/sshd_config** file:

```
[elliot@control plays]$ ansible node1 -m command -a "tail -5 /etc/ssh/sshd_config"
node1 | CHANGED | rc=0 >>
# BEGIN ANSIBLE MANAGED BLOCK
MaxAuthTries 4
Banner /etc/motd
X11Forwarding no
# END ANSIBLE MANAGED BLOCK
```

Amazing! Exactly as you expected it to be. Keep in mind that if you rerun the **configure-ssh.yml** playbook, Ansible will not edit (or append) the **/etc/ssh/sshd_config** file. You can try it for yourself!

I also recommend you take a look at the **blockinfile** and **lineinfile** documentation pages to understand the differences and the use of each module:

```
[elliot@control plays]$ ansible-doc blockinfile
[elliot@control plays]$ ansible-doc lineinfile
```

Alright! This takes us to the end of the sixth chapter. In the next chapter, you are going to learn how to use **Jinja2** Templates to deploy files and configure services dynamically in Ansible.

Knowledge Check

Create a playbook named **lab6.yml** that will accomplish the following tasks:

1. The playbook will run on all managed nodes.
2. Installs the **nfs-utils** package only on CentOS servers.

Hint: Use the **when** conditional.

Solution to the exercise is provided at the end of the book.

Chapter 7: Jinja2 Templates

In the previous chapter, you learned how to do simple file modifications by using the **blockinfile** or **lineinfile** Ansible modules.

In this chapter, you will learn how to use **Jinja2** templating engine to carry out more involved and dynamic file modifications.

You will learn how to access variables and facts in **Jinja2** templates. Furthermore, you will learn how to use conditional statements and loop structures in **Jinja2**.

Accessing variables in Jinja2

Ansible will look for **Jinja2** template files in your project directory or in a directory named **templates** under your project directory.

Let's create a **templates** directory to keep thing cleaner and more organized:

```
[elliot@control plays]$ mkdir templates
[elliot@control plays]$ cd templates/
```

Now create your first **Jinja2** template file with the name **index.j2**:

```
[elliot@control templates]$ cat index.j2
A message from {{ inventory_hostname }}
{{ webserver_message }}
```

Notice that **Jinja2** template filenames must end with the **.j2** extension.

The **inventory_hostname** is another Ansible built-in (aka special or magic) variable that references the 'current' host being iterated over in the play. The **webserver_message** is a variable that you will define in your playbook.

Now go one step back to your project directory and create the following **check-apache.yml**:

```
[elliot@control plays]$ cat check-apache.yml
---
- name: Check if Apache is Working
  hosts: webservers
  vars:
    webserver_message: "I am running to the finish line."
  tasks:
    - name: Start httpd
```

```
    service:
      name: httpd
      state: started

  - name: Create index.html using Jinja2
    template:
      src: index.j2
      dest: /var/www/html/index.html
```

Note that the **httpd** package was already installed in a previous chapter.

In this playbook, you first make sure Apache is running in the first task **Start httpd**. Then use the **template** module in the second task **Create index.html using Jinja2** to process and transfer the **index.j2** Jinja2 template file you created to the destination **/var/www/html/index.html**.

Go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook check-apache.yml

PLAY [Check if Apache is Working] *********************

TASK [Gathering Facts] *****************
ok: [node3]
ok: [node2]

TASK [Start httpd] ****************
ok: [node2]
ok: [node3]

TASK [Create index.html using Jinja2] ******************
changed: [node3]
changed: [node2]

PLAY RECAP ******************************
node2: ok=3    changed=1    unreachable=0    failed=0    skipped=0
node3: ok=3    changed=1    unreachable=0    failed=0    skipped=0
```

Everything looks good so far; let's run a quick ad-hoc Ansible command to check the contents of **index.html** on the **webservers** nodes:

```
[elliot@control plays]$ ansible webservers -m command -a "cat /var/www/html/index.html"
node3 | CHANGED | rc=0 >>
A message from node3
I am running to the finish line.
node2 | CHANGED | rc=0 >>
A message from node2
I am running to the finish line.
```

Amazing! Notice how **Jinja2** was able to pick up the values of the **inventory_hostname** built-in variable and the **webserver_message** variable in your playbook.

You can also use the **curl** command to see if you get a response from both webservers:

```
[elliot@control plays]$ curl node2.linuxhandbook.local
A message from node2
I am running to the finish line.
[elliot@control plays]$ curl node3.linuxhandbook.local
A message from node3
I am running to the finish line.
```

Accessing facts in Jinja2

You can access facts in **Jinja2** templates the same way you access facts from your playbook.

To demonstrate, change to your templates directory and create the **info.j2** Jinja2 file with the following contents:

```
[elliot@control templates]$ cat info.j2
Server Information Summary
--------------------------

hostname={{ ansible_facts['hostname'] }}
fqdn={{ ansible_facts['fqdn'] }}
ipaddr={{ ansible_facts['default_ipv4']['address'] }}
distro={{ ansible_facts['distribution'] }}
distro_version={{ ansible_facts['distribution_version'] }}
nameservers={{ ansible_facts['dns']['nameservers'] }}
totalmem={{ ansible_facts['memtotal_mb'] }}
freemem={{ ansible_facts['memfree_mb'] }}
```

Notice that **info.j2** accesses eight different facts. Now go back to your project directory and create the following **server-info.yml** playbook:

```
[elliot@control plays]$ cat server-info.yml
---
- name: Server Information Summary
  hosts: all
  tasks:
   - name: Create server-info.txt using Jinja2
     template:
       src: info.j2
       dest: /tmp/server-info.txt
```

Notice that you are creating **/tmp/server-info.txt** on all hosts based on the **info.j2** template file. Go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook server-info.yml

PLAY [Server Information Summary] *********************

TASK [Gathering Facts] ******************
ok: [node4]
ok: [node1]
ok: [node3]
ok: [node2]
```

```
TASK [Create server-info.txt using Jinja2] ********
changed: [node4]
changed: [node1]
changed: [node3]
changed: [node2]

PLAY RECAP *************************
node1: ok=2    changed=1    unreachable=0    failed=0    skipped=0
node2: ok=2    changed=1    unreachable=0    failed=0    skipped=0
node3: ok=2    changed=1    unreachable=0    failed=0    skipped=0
node4: ok=2    changed=1    unreachable=0    failed=0    skipped=0
```

Everything looks good! Now let's run a quick ad-hoc command to inspect the contents of the **/tmp/server-info.txt** file on one of the nodes:

```
[elliot@control plays]$ ansible node1 -m command -a "cat /tmp/server-info.txt"
node1 | CHANGED | rc=0 >>
Server Information Summary
--------------------------

hostname=node1
fqdn=node1.linuxhandbook.local
ipaddr=10.0.0.5
distro=CentOS
distro_version=8.2
nameservers=['168.63.129.16']
totalmem=1896
freemem=1087
```

As you can see, **Jinja2** was able to access and process all the facts.

Conditional statements in Jinja2

You can use the **if** conditional statement in **Jinja2** for testing various conditions and comparing variables. This allows you to determine your file template execution flow according to your test conditions.

To demonstrate, go to your templates directory and create the following **selinux.j2** template file:

```
[elliot@control templates]$ cat selinux.j2
{% set selinux_status = ansible_facts['selinux']['status'] %}

{% if selinux_status == "enabled" %}
    "SELINUX IS ENABLED"
{% elif selinux_status == "disabled" %}
    "SELINUX IS DISABLED"
{% else %}
    "SELINUX IS NOT AVAILABLE"
{% endif %}
```

The first statement in the template creates a new variable **selinux_status** and set its value to **ansible_facts['selinux']['status']**.

You then use **selinux_status** in your **if** test condition to determine whether **SELinux** is enabled, disabled, or not installed. In each of the three different cases, you display a message that reflects **SELinux** status.

Notice how the **if** statement in **Jinja2** mimics Python's **if** statement; just don't forget to use **{% endif %}**.

Now go back to your project directory and create the following **selinux-status.yml** playbook:

```
[elliot@control plays]$ cat selinux-status.yml
---
- name: Check SELinux Status
  hosts: all
  tasks:
    - name: Display SELinux Status
      debug:
        msg: "{{ ansible_facts['selinux']['status'] }}"

    - name: Create selinux.out using Jinja2
      template:
        src: selinux.j2
        dest: /tmp/selinux.out
```

Go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook selinux-status.yml

PLAY [Check SELinux Status] *********

TASK [Gathering Facts] *************
ok: [node4]
ok: [node2]
ok: [node3]
ok: [node1]

TASK [Display SELinux Status] ********
ok: [node1] => {
    "msg": "enabled"
}
ok: [node2] => {
    "msg": "disabled"
}
ok: [node3] => {
    "msg": "enabled"
}
ok: [node4] => {
    "msg": "Missing selinux Python library"
}

TASK [Create selinux.out using Jinja2] ********************
changed: [node4]
changed: [node1]
changed: [node3]
changed: [node2]

PLAY RECAP ******************
node1: ok=3    changed=1    unreachable=0    failed=0    skipped=0
node2: ok=3    changed=1    unreachable=0    failed=0    skipped=0
node3: ok=3    changed=1    unreachable=0    failed=0    skipped=0
node4: ok=3    changed=1    unreachable=0    failed=0    skipped=0
```

From the playbook output; you can see that **SELinux** is enabled on both **node1** and **node3**. I disabled **SELinux** on **node2** before running the playbook and **node4** doesn't have **SELinux** installed because Ubuntu uses **AppArmor** instead of **SELinux**.

Finally, you can run the following ad-hoc command to inspect the contents of **selinux.out** on all the managed hosts:

```
[elliot@control plays]$ ansible all -m command -a "cat /tmp/selinux.out"
node4 | CHANGED | rc=0 >>

"SELINUX IS NOT AVAILABLE"

node2 | CHANGED | rc=0 >>

"SELINUX IS DISABLED"

node3 | CHANGED | rc=0 >>

"SELINUX IS ENABLED"

node1 | CHANGED | rc=0 >>

"SELINUX IS ENABLED"
```

Looping in Jinja2

You can use the **for** statement in **Jinja2** to loop over items in a list, range, etc. For example, the following **for** loop will iterate over the numbers in the **range(1,11)** and will hence display the numbers from 1 →10:

```
{% for i in range(1,11) %}
    Number {{ i }}
{% endfor %}
```

Notice how the **for** loop in **Jinja2** mimics the syntax of Python's for loop; again don't forget to end the loop with **{% endfor %}**.

Now let's create a full example that shows off the power of **for** loops in **Jinja2**. Change to your templates directory and create the following **hosts.j2** template file:

```
[elliot@control templates]$ cat hosts.j2
{% for host in groups['all'] %}
{{ hostvars[host].ansible_facts.default_ipv4.address }}  {{ hostvars[host].ansible_facts.fqdn }}
{{ hostvars[host].ansible_facts.hostname }}
{% endfor %}
```

Notice here you used a new built-in special (magic) variable **hostvars** which is basically a dictionary that contains all the hosts in inventory and variables assigned to them.

You iterated over all the hosts in your inventory and then for each host; you displayed the value of three variables:

1. hostvars[host].ansible_facts.default_ipv4.address
2. hostvars[host].ansible_facts.fqdn
3. hostvars[host].ansible_facts.hostname

Notice also that you must include those three variables on the same line side by side to match the format of the **/etc/hosts** file.

Now go back to your projects directory and create the following **local-dns.yml** playbook:

```
[elliot@control plays]$ cat local-dns.yml
---
- name: Dynamically Update /etc/hosts File
  hosts: all
  tasks:
```

```
    - name: Update /etc/hosts using Jinja2
      template:
        src: hosts.j2
        dest: /etc/hosts
```

Then go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook local-dns.yml

PLAY [Dynamically Update /etc/hosts File] ******************

TASK [Gathering Facts] **************
ok: [node4]
ok: [node2]
ok: [node1]
ok: [node3]

TASK [Update /etc/hosts using Jinja2] ***************
changed: [node4]
changed: [node3]
changed: [node1]
changed: [node2]

PLAY RECAP ***********************
node1: ok=2    changed=1    unreachable=0    failed=0    skipped=0
node2: ok=2    changed=1    unreachable=0    failed=0    skipped=0
node3: ok=2    changed=1    unreachable=0    failed=0    skipped=0
node4: ok=2    changed=1    unreachable=0    failed=0    skipped=0
```

Everything looks good so far; now run the following ad-hoc command to verify that **/etc/hosts** file is properly updated on **node1**:

```
[elliot@control plays]$ ansible node1 -m command -a "cat /etc/hosts"
node1 | CHANGED | rc=0 >>
10.0.0.5  node1.linuxhandbook.local  node1
10.0.0.6  node2.linuxhandbook.local  node2
10.0.0.7  node3.linuxhandbook.local  node3
10.0.0.8  node4.linuxhandbook.local  node4
```

Perfect! Looks properly formatted as you expected it to be.

I hope you now realize the power of **Jinja2** templates in Ansible. In the next chapter, you are going to learn how to protect sensitive information and files using Ansible Vault.

Knowledge Check

Create a Jinja2 template file named **motd.j2** that has the following contents:

```
Welcome to {{ inventory_hostname }}.
My IP address is {{ ansible_facts['default_ipv4']['address'] }}.
```

Then create a playbook named **lab7.yml** that will edit the **/etc/motd** file on all managed hosts based on the **motd.j2** Jinja2 template file.

Hint: Use the **template** module.

Solution to the exercise is provided at the end of the book.

Chapter 8: Ansible Vault

There are many situations where you would want to use sensitive information in Ansible. For instance, you may want to set user's password, transfer certificates or keys, etc.

In this chapter, you will learn to:

- Use Ansible Vault to protect and deal with sensitive information.
- Create, view, and edit vault encrypted files.
- Decrypt vault encrypted files and to change the password of a vault encrypted file.

Furthermore, you will learn how to use encrypted variables and files in your playbooks.

Creating encrypted files

To create a new encrypted file; you can use the **ansible-vault create** command. To demonstrate, let's create a new encrypted file named **secret.txt**:

```
[elliot@control plays]$ ansible-vault create secret.txt
New Vault password:
Confirm New Vault password:
```

It will first prompt you for a vault password that you can use whenever you want to open the file later afterwards. After you enter the password, it will open the file with your default file editor and so you can go ahead and insert the following line:

```
I hold the key to the universe.
```

Now save and exit then try to view the contents of the **secret.txt** file:

```
[elliot@control plays]$ cat secret.txt
$ANSIBLE_VAULT;1.1;AES256
33313639663832656232616363326134313939643464626335366639 36
32653637343262623138323835393864353639363762663532623062333 13
5626465390a32373665393162386432366164326462373939336233623635
31356364663166643961653137303635313162373939336135326365613 4
```

As you can see, you can't really view the line you just inserted because the file is now encrypted with Ansible vault.

If you want to view the original content of a vault encrypted file; you can use the **ansible-vault view** command as follows:

```
[elliot@control plays]$ ansible-vault view secret.txt
Vault password:
I hold the key to the universe.
```

You can also store your vault password in a separate file. For example, if your vault password for the encrypted file **secret.txt** was **L!n*Xh#N%b,Ook**; you can store in a separate file **secret-vault.txt** as follows:

```
[elliot@control plays]$ echo 'L!n*Xh#N%b,Ook' > secret-vault.txt
[elliot@control plays]$ cat secret-vault.txt
L!n*Xh#N%b,Ook
```

Now you can use the **--vault-password-file** option followed by the path to your vault password file along with the **ansible-vault view** command to view the contents of **secret.txt** as follows:

```
[elliot@control plays]$ ansible-vault view secret.txt --vault-password-file secret-vault.txt
I hold the key to the universe.
```

As you can see; I didn't get prompted for a vault password this time around.

To modify the contents of a vault encrypted file; you can use the **ansible-vault edit** command as follows:

```
[elliot@control plays]$ ansible-vault edit secret.txt
Vault password:
```

You can also use the **--vault-password-file** option here as well:

```
[elliot@control plays]$ ansible-vault edit secret.txt --vault-password-file secret-vault.txt
```

Decrypting encrypted files

Let's create another encrypted file named **secret2.txt**:

```
[elliot@control plays]$ ansible-vault create secret2.txt
New Vault password:
Confirm New Vault password:
```

You can insert the following line in **secret2.txt**:

```
I hold another key to the universe.
```

You may decide later that the information in **secret2**.txt is no longer sensitive. If you want to decrypt a vault encrypted file; you can use the **ansible-vault decrypt** command as follows:

```
[elliot@control plays]$ cat secret2.txt
$ANSIBLE_VAULT;1.1;AES256
61333533346135323762623366393566356636631616365396
23838373138326161356337633032323165646539666238 30
38363735613763393536306235303 30a63643231666665326
66438366133376431313935383266386666316463

[elliot@control plays]$ ansible-vault decrypt secret2.txt
Vault password:
Decryption successful

[elliot@control plays]$ cat secret2.txt
I hold another key to the universe.
```

As you can see; the contents of the file **secret2.txt** is no longer encrypted.

Changing an encrypted file's password

You can also encrypt existing files using the **ansible-create encrypt** command; for instance, you can encrypt the unencrypted **secret2.txt** file again as follows:

```
[elliot@control plays]$ ansible-vault encrypt secret2.txt
New Vault password:
Confirm New Vault password:
Encryption successful

[elliot@control plays]$ cat secret2.txt
$ANSIBLE_VAULT;1.1;AES256
64633234613936303362626132363735623763613534323430 3
63261356232353839393066333235383938623731396165643 3
65646236306164626631613862 0a656266316633623039343 53
866643735383866353833303061656130343461
```

You may notice that the vault password of **secret2.txt** got compromised. In this case, you can use the **ansible-vault rekey** command to change the vault password as follows:

```
[elliot@control plays]$ ansible-vault rekey secret2.txt
Vault password:
New Vault password:
Confirm New Vault password:
Rekey successful
```

Notice that you needed to enter the old vault password before entering the new one.

Decrypting content at run time in playbooks

You can use vault encrypted files in your Ansible playbooks. To demonstrate, let's first create a new encrypted file named **web-secrets.yml**:

```
[elliot@control plays]$ ansible-vault create web-secrets.yml
New Vault password:
Confirm New Vault password:
```

The **web-secrets.yml** should contain **secret1** variable as follows:

```
[elliot@control plays]$ ansible-vault view web-secrets.yml
Vault password:
secret1: "webkey"
```

Now create a new Ansible playbook named **vault-playbook.yml** with the following contents:

```
[elliot@control plays]$ cat vault-playbook.yml
---
- name: Accessing Vaults in Playbooks
  hosts: node2
  vars_files: web-secrets.yml
  tasks:
    - name: Show secret1 value
      debug:
        msg: "{{ secret1 }}"
```

Notice how **vault-playbook.yml** is accessing variables in the vault encrypted file **web-secrets.yml**. Now try running the playbook:

```
[elliot@control plays]$ ansible-playbook vault-playbook.yml
ERROR! Attempting to decrypt but no vault secrets found
```

As you can see; it is complaining as it didn't receive a vault password to decrypt the **web-secrets.yml** file.

Now run the playbook again but pass the **--ask-vault-pass** option this time around:

```
[elliot@control plays]$ ansible-playbook --ask-vault-pass vault-playbook.yml
Vault password:

PLAY [Accessing Vaults in Playbooks] ***************

TASK [Gathering Facts] *********************************
ok: [node2]

TASK [Show secret1 value] ******************************
ok: [node2] => {
    "msg": "webkey"
}

PLAY RECAP *************************
node2: ok=2    changed=0    unreachable=0    failed=0    skipped=0
```

As you can see; the playbook prompted you for the vault password and then runs successfully.

You could have also used the **--vault-password-file** if you have stored your vault password in a file:

```
[elliot@control plays]$ ansible-playbook --vault-password-file vault-pass.txt vault-playbook.yml
```

You can also use the **--vault-id** option to access multiple encrypted files in your playbook. To demonstrate, let's create another encrypted file named **db-secrets.yml**:

```
[elliot@control plays]$ ansible-vault create db-secrets.yml
New Vault password:
Confirm New Vault password:
```

The **db-secrets.yml** should contain **secret2** variable as follows:

```
[elliot@control plays]$ ansible-vault view web-secrets.yml
Vault password:
secret2: "dbkey"
```

Now let's edit **vault-playbook.yml** so it uses **db-secrets.yml** as well:

```
[elliot@control plays]$ cat vault-playbook.yml
---
```

```
- name: Accessing Vaults in Playbooks
  hosts: node2
  vars_files:
    - web-secrets.yml
    - db-secrets.yml
  tasks:
    - name: Show secret1 value
      debug:
        msg: "{{ secret1 }}"

    - name: Show secret2 value
      debug:
        msg: "{{ secret2 }}"
```

To run the playbook; you have to provide the vault password for each encrypted file using the **--vault-id** option as follows:

```
[elliot@control plays]$ ansible-playbook --vault-id web-secrets.yml@prompt
--vault-id db-secrets@prompt vault-playbook.yml
Vault password (web-secrets.yml):
Vault password (db-secrets):

PLAY [Accessing Vaults in Playbooks] ************************

TASK [Gathering Facts] **************************
ok: [node2]

TASK [Show secret1 value] ***********************
ok: [node2] => {
    "msg": "webkey"
}

TASK [Show secret2 value] ********************
ok: [node2] => {
    "msg": "dbkey"
}

PLAY RECAP ********************
node2: ok=3    changed=0    unreachable=0    failed=0    skipped=0
```

The playbook prompts you to enter vault password for both files: **web-secrets.yml** and **db-secrets.yml** and then runs successfully. You could have also used files to store your vault passwords; in this case, this is the command you need to run your playbook:

```
[elliot@control plays]$ ansible-playbook --vault-id /path-to-web-vault-file
--vault-id /path-to-db-vault-file vault-playbook.yml
```

I hope you have now learned how to protect sensitive information and how to use encrypted files in your playbooks by using Ansible vault. In the next chapter, you will learn how to create and use Ansible roles.

Knowledge Check

Using the vault password: **7uZAcMBVz**

Create a vault encrypted file named **mysecret.txt** that has the following contents:

```
I like pineapple pizza!
```

Store the vault password in the file **mypass.txt**.

Solution to the exercise is provided at the end of the book.

Chapter 9: Ansible Roles

So far you have been creating Ansible playbooks to automate a certain task on your managed nodes. There is a huge chance that someone else has already designed an Ansible solution to the problem/task you are trying to solve and that's exactly what Ansible roles is all about.

In this chapter, you will understand how roles are structured in Ansible. You will also learn to use ready-made roles from Ansible Galaxy.

Furthermore, you will learn to create your own custom Ansible roles.

Understanding Ansible Roles

An Ansible role is a collection of files, tasks, templates, variables, and handlers that together serve a certain purpose like configuring a service. Roles allows you to easily re-use code and share Ansible solutions with other users which makes working with large environments more manageable.

Role directory structure

A typical Ansible role follows a defined directory structure that is usually composed of the following directories:

1. **defaults** → contains default variables for the role that are meant to be easily overwritten.
2. **vars** → contains standard variables for the role that are not meant to be overwritten in your playbook.
3. **tasks** → contains a set of tasks to be performed by the role.
4. **handlers** → contains a set of handlers to be used in the role.
5. **templates** → contains the Jinja2 templates to be used in the role.
6. **files** → contains static files which are accessed in the role tasks.
7. **tests** → may contain an optional inventory file, as well as **test.yml** playbook that can be used to test the role.
8. **meta** → contains role metadata such as author information, license, dependencies, etc.

Keep in mind that a role may have all the aforementioned directories or just a subset of them. In fact, you can define an empty role that has zero directories, although won't be useful!

Storing and locating roles

By default, Ansible will look for roles in two locations:

1. In **roles/** directory (same directory level as your playbook file).
2. In **/etc/ansible/roles** directory.

You can however choose to store your roles in a different location; if you choose to do so, you have to specify and set the **roles_path** configuration option in your Ansible configuration file (**ansible.cfg**).

```
[defaults]
inventory = myhosts
remote_user = elliot
host_key_checking = false
roles_path = /path/to/your/roles

[privilege_escalation]
become = true
become_method = sudo
become_user = root
become_ack_pass = false
```

Figure 9: Setting **roles_path**

Using roles in playbooks

There are two different ways you can use to import roles in an Ansible playbook:

1. Using the **roles** keyword to statically import roles.
2. Using the **include_role** module to dynamically import roles.

For instance, to statically import roles in a playbook, you can use the **roles** keyword in the playbook header as follows:

```
---
- name: Including roles statically
  hosts: all
  roles:
    - role1
    - role2
```

To dynamically import roles; you can use the **include_role** module as follows:

```
---
- name: Including roles dynamically
  hosts: all
  tasks:
    - name: import dbserver role
      include_role:
        name: db_role
      when: inventory_hostname in groups['dbservers']
```

Using Ansible Galaxy for Ready-Made Roles

Imagine a place where all the Ansible roles you need are already provided for free; That place is called Ansible Galaxy and it's real!

Ansible Galaxy is a public website that where community provided roles are offered. It is basically a public repository that hosts a massive amount of Ansible roles. I recommend you pay a visit to Ansible Galaxy's website galaxy.ansible.com and check out its amazing content.

Searching for roles

Ansible Galaxy has got its own CLI (Command Line Interface) utility and you can use it to carry out roles related operations.

For example, you can search for role in the Galaxy's repository by using the **ansible-galaxy search** command as follows:

```
[elliot@control plays]$ ansible-galaxy search mariadb

Found 370 roles matching your search:
Name                                          Description
 ----                                          -----------
 0x_peace.mariadb                              Mariadb role
 5003.mariadb                                  SQL Server relational database >
 aalaesar.install_nextcloud                    Add a new Nextcloud instance in>
 aalaesar.upgrade-nextcloud                    Upgrade an Nextcloud instance i>
 aaronpederson.mariadb                         MariaDB - An enhanced, drop-in >
 acandid.mariadb                               Install and Configure MariaDB 1>
 acandid.mariadb_apache_wordpress              Install and Configure MariaDB, >
 acandid.mariadb_galera_cluster                Install and Configure MariaDB G>
 acandid.zabbix                                Install Zabbix 4.2 Server
 achaussier.mariadb-server                     Ansible role to install MariaDB>
 adfinis-sygroup.mariadb                       Install and manage mariadb/mysq>
 AdnanHodzic.containerized-wordpress           Deploy & run Docker Compose pro>
 ajeeshbas.ansible_role_mariadb                your description
 alainvanhoof.alpine_mariadb                   MariaDB for Alpine Linux
 alexandrem.mariadb                            MariaDB server role
 alexeymedvedchikov.galera                     An ansible role for Galera Mari>
 alexfeig.guacamole                            Ansible role for installing Apa>
 alikins.mysql                                 MySQL server for RHEL/CentOS an>
 geerlingguy.mysql                             MySQL server for RHEL/CentOS an
```

As you can see; it listed all the Galaxy's roles related to my search term **mariadb**.

Getting role information

You can use the **ansible-galaxy info** command to display a role's information.

For example, based on the search results that you got from **ansible-galaxy search mariadb**; you may decide to get more information on the **geerlingguy.mysql** role:

```
[elliot@control ~]$ ansible-galaxy info geerlingguy.mysql

Role: geerlingguy.mysql
        description: MySQL server for RHEL/CentOS and Debian/Ubuntu.
        active: True
        commit: 0a354d6ad1e4f466aad5f789ba414f31b97296fd
        commit_message: Switch to travis-ci.com.
        commit_url: https://api.github.com/repos/geerlingguy/ansible-role-mysql>
        company: Midwestern Mac, LLC
        created: 2014-03-01T03:32:33.675832Z
        download_count: 1104067
        forks_count: 665
        github_branch: master
        github_repo: ansible-role-mysql
        github_user: geerlingguy
        id: 435
        imported: 2020-10-29T19:36:43.709197-04:00
        is_valid: True
        issue_tracker_url: https://github.com/geerlingguy/ansible-role-mysql/is>
        license: license (BSD, MIT)
        min_ansible_version: 2.4
        modified: 2020-10-29T23:36:43.716291Z
        open_issues_count: 24
```

As you can see; It showed you a detailed description of the **geerlingguy.mysql** role.

Installing and using roles

Before I show you how you to install Ansible Galaxy roles; Go ahead and create a new roles directory under your project directory (**plays**):

```
[elliot@control plays]$ mkdir roles
```

You can now use the **ansible-galaxy install** command to install the **geerlingguy.mysql** role as follows:

```
[elliot@control plays]$ ansible-galaxy install geerlingguy.mysql -p ./roles
- downloading role 'mysql', owned by geerlingguy
- downloading role from https://github.com/geerlingguy/ansible-role-mysql/archive/
3.3.0.tar.gz
- extracting geerlingguy.mysql to /home/elliot/plays/roles/geerlingguy.mysql
- geerlingguy.mysql (3.3.0) was installed successfully
```

Notice that I used the **-p** option to specify the path where I want the role to be installed. By default, Ansible Galaxy would install the role in the ~/**.ansible/roles** directory.

Now go ahead and change to your **roles** directory and list the contents of the **geerlingguy.mysql** role directory:

```
[elliot@control plays]$ cd roles/
[elliot@control roles]$ ls
geerlingguy.mysql
[elliot@control roles]$ tree geerlingguy.mysql/
geerlingguy.mysql/
   defaults
       main.yml
   handlers
       main.yml
   LICENSE
   meta
       main.yml
   README.md
   tasks
       configure.yml
       databases.yml
       main.yml
       replication.yml
       secure-installation.yml
       setup-Debian.yml
       setup-RedHat.yml
       users.yml
       variables.yml
   templates
       my.cnf.j2
       root-my.cnf.j2
       user-my.cnf.j2
   vars
        Archlinux.yml
        Debian-10.yml
        Debian.yml
        RedHat-6.yml
        RedHat-7.yml
        RedHat-8.yml

8 directories, 26 files
```

As you can see; the **geerlingguy.mysql** role follows the standard directory structure that I had shown you earlier.

Now let's go back to the **plays** directory and create a new playbook named **mysql-role.yml** that applies the **geerlingguy.mysql** role on the **dbservers** group host:

```
[elliot@control plays]$ cat mysql-role.yml
---
- name: Applying geerlingguy.mysql role
  hosts: dbservers
  roles:
    - geerlingguy.mysql
```

Now go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook mysql-role.yml

PLAY [Applying geerlingguy.mysql role] ************

TASK [Gathering Facts] ***************
ok: [node4]

TASK [geerlingguy.mysql : include_tasks] ************************************
included: /home/elliot/plays/roles/geerlingguy.mysql/tasks/variables.yml for node4

TASK [geerlingguy.mysql : Include OS-specific variables.] ***********************
ok: [node4] => (item=/home/elliot/plays/roles/geerlingguy.mysql/vars/Debian.yml)

TASK [geerlingguy.mysql : Define mysql_packages.] **********
ok: [node4]

TASK [geerlingguy.mysql : Define mysql_daemon.]

.
.
.

RUNNING HANDLER [geerlingguy.mysql : restart mysql] **********

PLAY RECAP **********************
node4: ok=32   changed=10   unreachable=0    failed=0   skipped=12
```

After the playbook is done running; **mysql** should be up and running on **node4**:

```
[elliot@control plays]$ ansible node4 -m command -a "systemctl status mysql"
node4 | CHANGED | rc=0 >>
 mysql.service - MySQL Community Server
   Loaded: loaded (/lib/systemd/system/mysql.service; enabled;
   vendor preset: enabled)
```

```
   Active: active (running) since Tue 2020-11-10 23:31:51 UTC; 7min ago
  Process: 26544 ExecStart=/usr/sbin/mysqld --daemonize
  --pid-file=/run/mysqld/mysqld.pid (code=exited, status=0/SUCCESS)
  Process: 26524 ExecStartPre=/usr/share/mysql/mysql-systemd-start
  pre (code=exited, status=0/SUCCESS)
 Main PID: 26546 (mysqld)
    Tasks: 28 (limit: 2265)
   CGroup: /system.slice/mysql.service
           26546 /usr/sbin/mysqld --daemonize --pid-file=/run/mysqld/mysqld.pid

Nov 10 23:31:39 node4 systemd[1]: Starting MySQL Community Server...
Nov 10 23:31:51 node4 systemd[1]: Started MySQL Community Server.
```

If you no longer need a role; you can delete it using the **ansible-galaxy remove** command as follows:

```
[elliot@control plays]$ ansible-galaxy remove geerlingguy.mysql -p ./roles
- successfully removed geerlingguy.mysql
```

Using a requirements file to install multiple roles

Ansible galaxy can install multiple roles at once using a requirements file.

Each role you define in your requirements file will have one more of the following attributes:

1. **src** → The source of the role (mandatory attribute).
2. **scm** → If the source (src) is a URL, specify the SCM. Defaults to **git**; only **git** and **hg** are supported. (optional)
3. **name** → Download the role to a specific name. Defaults to the Galaxy name or repo name when downloading from **git**. (optional)
4. **version** → The version of the role to download. Defaults to the master version. (optional)

To demonstrate, go ahead and create a **requirements.yml** file with the following three roles:

```
[elliot@control plays]$ cat requirements.yml
# From Ansible Galaxy
- src: geerlingguy.haproxy

# From Github
- src: https://github.com/bennojoy/nginx
```

```
  name: nginx_role
  version: master

# From Ansible Galaxy
- src: geerlingguy.jenkins
  name: jenkins_role
```

Now you can use the **ansible-galaxy install** command along with the **-r** option to install the three roles in your **requirements.yml** file:

```
[elliot@control plays]$ ansible-galaxy install -r requirements.yml -p ./roles
- downloading role 'haproxy', owned by geerlingguy
- downloading role from https://github.com/geerlingguy/ansible-role-haproxy/
archive/1.1.2.tar.gz
- extracting geerlingguy.haproxy to /home/elliot/plays/roles/geerlingguy.haproxy
- geerlingguy.haproxy (1.1.2) was installed successfully
- extracting nginx_role to /home/elliot/plays/roles/nginx_role
- nginx_role (master) was installed successfully
- downloading role 'jenkins', owned by geerlingguy
- downloading role from https://github.com/geerlingguy/ansible-role-jenkins/
archive/4.3.0.tar.gz
- extracting jenkins_role to /home/elliot/plays/roles/jenkins_role
- jenkins_role (4.3.0) was installed successfully
```

As you can see; the three roles defined in the **requirements.yml** file are successfully installed. You can also use the **ansible-galaxy list** command to list the installed roles along with their versions:

```
[elliot@control plays]$ ansible-galaxy list -p ./roles
# /home/elliot/plays/roles
- geerlingguy.haproxy, 1.1.2
- nginx_role, master
- jenkins_role, 4.3.0
```

Creating Custom Roles

You can also define your own roles. To do that, you can use the **ansible-galaxy init** command to define your role structure.

To demonstrate, let's create a new role named **httpd-role**. First, change to your **roles** directory and then run the **ansible-galaxy init** command followed by the new role name as follows:

```
[elliot@control plays]$ cd roles/
[elliot@control roles]$ ansible-galaxy init httpd-role
- Role httpd-role was created successfully
```

Notice that a new role was created with the name **httpd-role** and it has all the typical role directories.

```
[elliot@control roles]$ tree httpd-role/
httpd-role/
   defaults
       main.yml
   files
   handlers
       main.yml
   meta
       main.yml
   README.md
   tasks
       main.yml
   templates
   tests
       inventory
       test.yml
   vars
        main.yml

8 directories, 8 files
```

Now start defining tasks, templates, and default variables for the new **httpd-role**.

First, you can start by defining the tasks by editing **tasks/main.yml** so that it contains the following contents:

```
[elliot@control httpd-role]$ cat tasks/main.yml
---
# tasks file for httpd-role

- name: Install httpd
  yum:
    name: httpd
    state: latest

- name: Start and enable httpd
  service:
    name: httpd
    state: started
    enabled: true

- name: Create index.html using Jinja2
  template:
    src: index.j2
    dest: /var/www/html/index.html
```

Then you can create the Jinja2 template file **index.j2** inside the role's **templates** directory:

```
[elliot@control httpd-role]$ cat templates/index.j2
Welcome to {{ inventory_hostname }}

This is an Apache Web Server.

Please contact {{ sysadmin }} for any questions or concerns.
```

Finally, you can define and set the **sysadmin** variable in **defaults/main.yml**:

```
[elliot@control httpd-role]$ cat defaults/main.yml
---
# defaults file for httpd-role

sysadmin: elliot@linuxhandbook.com
```

Alright! Now you are done creating the role. Let's create a playbook that uses the **httpd-role**.

Go back to your project directory and create the **apache-role.yml** playbook with the following contents:

```
[elliot@control plays]$ cat apache-role.yml
---
- name: Using httpd-role
  hosts: webservers
  roles:
    - role: httpd-role
      sysadmin: angela@linuxhandbook.com
```

Notice that you did overwrite the **sysadmin** variable in the playbook. Go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook apache-role.yml

PLAY [Using httpd-role] ****************************

TASK [Gathering Facts] *************************************
ok: [node2]
ok: [node3]

TASK [httpd-role : Install httpd] **********************
changed: [node2]
changed: [node3]

TASK [httpd-role : Start and enable httpd] *******************
changed: [node2]
changed: [node3]

TASK [httpd-role : Create index.html using Jinja2] ****************
changed: [node2]
changed: [node3]

PLAY RECAP ******************
node2: ok=4    changed=3    unreachable=0    failed=0    skipped=0
node3: ok=4    changed=3    unreachable=0    failed=0    skipped=0
```

Everything looks good. Let's verify by checking the response you get on both webservers (**node2** and **node3**):

```
[elliot@control plays]$ curl node2
Welcome to node2

This is an Apache Web Server.

Please contact angela@linuxhandbook.com for any questions or concerns.

[elliot@control plays]$ curl node3
```

```
Welcome to node3

This is an Apache Web Server.

Please contact angela@linuxhandbook.com for any questions or concerns.
```

Perfect! Your custom-made **httpd-role** has worked flawlessly.

> If you ever create an awesome Ansible role that you think a lot of people can benefit from; don't forget to publish your role to Ansible Galaxy to share it with the World!

Managing Order of Task Execution

You need to be well aware of the order of task execution in an Ansible playbook.

If you use the **roles** keywork to import a role statically; then all the tasks in the role will run before all other tasks (included under the **tasks** section) in your play.

You can use the **pre_tasks** keyword to include any tasks you want to run before statically imported roles. You can also use the **post_tasks** keyword to include any tasks you want to run after all the tasks under the **tasks** section.

In summary, Ansible executes your playbook in the following order:

1. **pre_tasks** will run first.
2. handlers triggered by **pre_tasks** run next.
3. statically imported roles listed under **roles** will run.
4. tasks listed under the **tasks** section.
5. handlers triggered by **roles** or **tasks**.
6. **post_tasks** will run last.
7. handlers triggered by **post_tasks** will run very last.

For a demonstration, take a look at the following **pre-post.yml** playbook which uses **pre_tasks**, **roles**, **tasks**, and **post_tasks**:

```
[elliot@control plays]$ cat pre-post.yml
---
- name: Understanding Order of Task Execution
  hosts: node1
  tasks:
    - name: A regular task
      debug:
        msg: "I am just a regular task."

  post_tasks:
    - name: Runs last
      debug:
        msg: "I will run last (post_task)."

  pre_tasks:
    - name: Runs first
      debug:
        msg: "I will run first (pre_task)."

  roles:
    - role: myrole
```

The playbook uses a role named **myrole** that I have created that just has two simple tasks:

```
[elliot@control plays]$ tree roles/myrole
roles/myrole
   tasks
       main.yml

1 directory, 1 file
[elliot@control plays]$ cat roles/myrole/tasks/main.yml
- name:
  debug:
    msg: "I am the first task in myrole."

- name:
  debug:
    msg: "I am the second task in myrole."
```

Now go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook pre-post.yml

PLAY [Understanding Order of Task Execution] *************

TASK [Gathering Facts] *********************************
ok: [node1]

TASK [Runs first] **************************
    "msg": "I will run first (pre_tasks)."
}

TASK [myrole : debug] ******************************
ok: [node1] => {
    "msg": "I am the first task in myrole."
}

TASK [myrole : debug] ************************
ok: [node1] => {
    "msg": "I am the second task in myrole."
}

TASK [A regular task] ****************************
ok: [node1] => {
    "msg": "I am just a regular task."
}
```

```
TASK [Runs last] ****************************
ok: [node1] => {
    "msg": "I will run last (post_tasks)."
}

PLAY RECAP ****************************
node1: ok=6    changed=0    unreachable=0    failed=0    skipped=0
```

As you can see; **pre_tasks** runs first followed by the two tasks in **myrole** then **tasks** and finally **post_tasks** runs last.

I hope you enjoyed learning how to use and create Ansible roles. In the next chapter, you are going to learn how to install and use roles that are specific to RHEL systems using **RHEL System Roles**.

Knowledge Check

Download the **geerlingguy.haproxy** role via Ansible Galaxy.

Then create a playbook named **lab9.yml** that will run on the **proxy** host group and will accomplish the following tasks:

1. Load balance http requests between the hosts in the **webservers** host group by using the **geerlingguy.haproxy** role.
2. Use the **Round Robin** load balancing method.
3. HAProxy backend servers will only handle HTTP requests (**port 80**).

To test if your playbook works, run the following command twice:

```
curl http://node1
```

and you should get two different responses (one from **node2** and the other from **node3**).

Hint: check **README.md** to see example playbooks.

Solution to the exercise is provided at the end of the book.

Chapter 10: RHEL System Roles

In the previous chapter; you learned to use Ansible Galaxy's roles and create your own custom roles. Let's continue the discussion on Ansible roles but this time; we will focus on RHEL System Roles.

Red Hat has created a collection of Ansible roles that is primarily targeting RHEL systems; these collections of roles are referred to as **Red Hat Enterprise Linux (RHEL) System Roles**.

In this chapter, you will learn how to install and use RHEL System Roles to manage and automate standard RHEL operations.

Installing RHEL System Roles

The RHEL System Roles is provided by the **rhel-system-roles** package. So, let's go ahead and install the **rhel-system-roles** package:

```
[elliot@control plays]$ sudo yum search rhel-system-roles
================== Name Exactly Matched: rhel-system-roles ===================
rhel-system-roles.noarch : Set of interfaces for unified system management

[elliot@control plays]$ sudo yum -y install rhel-system-roles

Installed:
  rhel-system-roles-1.0-10.el8_1.noarch
Complete!
```

The RHEL system roles will be installed in the **/usr/share/ansible/roles** directory:

```
[elliot@control plays]$ ls -l /usr/share/ansible/roles
total 0
lrwxrwxrwx.  1 root root  23 Oct 22  2019 linux-system-roles.kdump ->
rhel-system-roles.kdump
lrwxrwxrwx.  1 root root  25 Oct 22  2019 linux-system-roles.network ->
rhel-system-roles.network
lrwxrwxrwx.  1 root root  25 Oct 22  2019 linux-system-roles.postfix ->
rhel-system-roles.postfix
lrwxrwxrwx.  1 root root  25 Oct 22  2019 linux-system-roles.selinux ->
rhel-system-roles.selinux
lrwxrwxrwx.  1 root root  25 Oct 22  2019 linux-system-roles.storage ->
rhel-system-roles.storage
lrwxrwxrwx.  1 root root  26 Oct 22  2019 linux-system-roles.timesync ->
rhel-system-roles.timesync
drwxr-xr-x.  9 root root 156 Nov 14 22:44 rhel-system-roles.kdump
drwxr-xr-x.  8 root root 177 Nov 14 22:44 rhel-system-roles.network
```

```
drwxr-xr-x.  6 root root 114 Nov 14 22:44 rhel-system-roles.postfix
drwxr-xr-x.  8 root root 138 Nov 14 22:44 rhel-system-roles.selinux
drwxr-xr-x. 10 root root 215 Nov 14 22:44 rhel-system-roles.storage
drwxr-xr-x. 11 root root 187 Nov 14 22:44 rhel-system-roles.timesync
```

As you can see from listing the contents of **/usr/share/ansible/roles**; the following RHEL system roles are currently provided:

1. **rhel-system-roles.kdump** → configures the kdump crash recovery service.
2. **rhel-system-roles.network** → configures the network interfaces.
3. **rhel-system-roles.postfix** → configures a host as a Postfix MTA.
4. **rhel-system-roles.selinux** → manages all aspects of SELinux.
5. **rhel-system-roles.storage** → configures local storage.
6. **rhel-system-roles.timesync** → configures Network Time Protocol (NTP) or Precision Time Protocol (PTP).

It's highly likely that additional RHEL system roles will be introduced in the future.

You can also find documentation and example playbooks for RHEL system roles in the **/usr/share/doc/rhel-system-roles** directory:

```
[elliot@control plays]$ ls -l /usr/share/doc/rhel-system-roles
total 4
drwxr-xr-x. 2 root root   57 Nov 14 22:44 kdump
drwxr-xr-x. 2 root root 4096 Nov 14 22:44 network
drwxr-xr-x. 2 root root   57 Nov 14 22:44 postfix
drwxr-xr-x. 2 root root   93 Nov 14 22:44 selinux
drwxr-xr-x. 2 root root   57 Nov 14 22:44 storage
drwxr-xr-x. 2 root root  136 Nov 14 22:44 timesync

[elliot@control plays]$ ls -l /usr/share/doc/rhel-system-roles/timesync/
total 48
-rw-r--r--. 1 root root  1057 Jun 13  2019 COPYING
-rw-r--r--. 1 root root   316 Oct 22  2019 example-timesync-playbook.yml
-rw-r--r--. 1 root root   176 Oct 22  2019 example-timesync-pool-playbook.yml
-rw-r--r--. 1 root root 27975 Oct 22  2019 README.html
-rw-r--r--. 1 root root  4320 Oct 22  2019 README.md
```

I will now show you how to use two of the most popular RHEL system roles:

- **rhel-system-roles.selinux**
- **rhel-system-roles.timesync**

Using RHEL SELinux System Role

You can use the RHEL SELinux system role (**rhel-system-roles.selinux**) to do any of the following:

1. Setting SELinux mode (Enforcing or Permissive)
2. Setting SELinux Booleans
3. Setting SELinux file contexts
4. Resorting default SELinux security contexts (restorecon)
5. Setting SELinux user mappings

Keep in mind that a reboot maybe required to apply an SELinux change; for example, switching between enabled and disabled SELinux modes will require a reboot for the change to take effect.

The **rhel-system-roles.selinux** doesn't reboot hosts and leaves it up to the administrator to do that if he/she chooses to. However, the role will set the **selinux_reboot_required** variable to true and **fail** to give you a hint that a reboot is required.

You are then free to choose to reboot the host and reapply the role again to finish the changes; this can be done using a **block-rescue** construct as you can see in the example playbook in the SELinux role's documentation directory:

```
[elliot@control plays]$ cat /usr/share/doc/rhel-system-roles/
selinux/example-selinux-playbook.yml
---
- hosts: all
  become: true
  become_method: sudo
  become_user: root
  vars:
    selinux_policy: targeted
    selinux_state: enforcing
    selinux_booleans:
      - { name: 'samba_enable_home_dirs', state: 'on' }
      - { name: 'ssh_sysadm_login', state: 'on', persistent: 'yes' }
    selinux_fcontexts:
      - { target: '/tmp/test_dir(/.*)?', setype: 'user_home_dir_t', ftype: 'd' }
    selinux_restore_dirs:
      - /tmp/test_dir
    selinux_ports:
      - { ports: '22100', proto: 'tcp', setype: 'ssh_port_t', state: 'present' }
    selinux_logins:
      - { login: 'sar-user', seuser: 'staff_u', serange: 's0-s0:c0.c1023',
      state: 'present' }
```

```
# prepare prerequisites which are used in this playbook
tasks:
  - name: Creates directory
    file:
      path: /tmp/test_dir
      state: directory
  - name: Add a Linux System Roles SELinux User
    user:
      comment: Linux System Roles SELinux User
      name: sar-user
  - name: execute the role and catch errors
    block:
      - include_role:
          name: rhel-system-roles.selinux
    rescue:
      # Fail if failed for a different reason than selinux_reboot_required.
      - name: handle errors
        fail:
          msg: "role failed"
        when: not selinux_reboot_required

      - name: restart managed host
        shell: sleep 2 && shutdown -r now "Ansible updates triggered"
        async: 1
        poll: 0
        ignore_errors: true

      - name: wait for managed host to come back
        wait_for_connection:
          delay: 10
          timeout: 300

      - name: reapply the role
        include_role:
          name: rhel-system-roles.selinux
```

To demonstrate, let's use the **rhel-system-roles.selinux** role to disable SELinux on **node1**. First, make sure that SELinux is enabled on **node1** with a quick ad-hoc command:

```
[elliot@control plays]$ ansible node1 -m command -a "sestatus"
node1 | CHANGED | rc=0 >>
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   permissive
Mode from config file:          enforcing
Policy MLS status:              enabled
```

```
Policy deny_unknown status:     allowed
Memory protection checking:     actual (secure)
Max kernel policy version:      31
```

Now create a playbook named **disable-selinux.yml** that has the following contents:

```
[elliot@control plays]$ cat disable-selinux.yml
---
- name: Disable SELinux using RHEL SELinux System Role
  hosts: node1
  vars:
    selinux_state: disabled
  tasks:
    - name: execute the role and catch errors
      block:
        - name: apply the role
          include_role:
           name: rhel-system-roles.selinux
      rescue:
        #Fail if failed for a different reason that selinux_reboot_required.
        - name: handler errors
          fail:
            msg: "role failed"
          when: not selinux_reboot_required

        #Otherwise, reboot the host (as selinux_reboot_required is true)
        - name: reboot the host
          reboot:
            msg: "System is rebooting"
          ignore_errors: true

        - name: Apply the role again
          include_role:
            name: rhel-system-roles.selinux
```

Also, make sure you set **roles_path** to **/usr/share/ansible/roles** in your **ansible.cfg** file; otherwise, your playbooks will not be able to find any of the RHEL system roles:

```
[elliot@control plays]$ grep roles ansible.cfg
roles_path = /usr/share/ansible/roles
```

Finally, go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook disable-selinux.yml

PLAY [Disable SELinux using RHEL SELinux System Role] *****************

TASK [Gathering Facts] *******************
ok: [node1]

TASK [apply the role] ***************************

TASK [rhel-system-roles.selinux : Install SELinux tool semanage on Fedora] ********
changed: [node1]

TASK [rhel-system-roles.selinux : Set permanent SELinux state if enabled] ******
[WARNING]: SELinux state change will take effect next reboot
changed: [node1]

TASK [rhel-system-roles.selinux : Set permanent SELinux state if disabled] *******
skipping: [node1]

TASK [rhel-system-roles.selinux : Set ansible facts if needed] *************
ok: [node1]

TASK [rhel-system-roles.selinux : Fail if reboot is required] ******
fatal: [node1]: FAILED! => {"changed": false, "msg": "Reboot is required
to apply changes.Re-execute the role after boot."}

TASK [handler errors] ******************
skipping: [node1]

TASK [reboot the host] *******************
changed: [node1]

TASK [Apply the role again] *********************

TASK [rhel-system-roles.selinux : Install SELinux python2 tools] ************
skipping: [node1]

TASK [rhel-system-roles.selinux : Fail if reboot is required] ************
skipping: [node1]

TASK [rhel-system-roles.selinux : debug] ***************************
ok: [node1] => {
    "msg": "SELinux is disabled on system - some SELinux modules can crash"
}


PLAY RECAP ****************
node1: ok=13   changed=3    unreachable=0    failed=0    skipped=18   rescued=1
```

Notice how the first time you applied the role; it failed as a **reboot** was required.

The failure was then handled in the **rescue** section in the playbook where you reboot the host and then reapply the role again to make sure that it succeeded.

Now let's run an ad-hoc command to verify that SELinux is now disabled on **node1**:

```
[elliot@control plays]$ ansible node1 -m command -a "sestatus"
node1 | CHANGED | rc=0 >>
SELinux status:                 disabled
```

Success! Now that you have learned how to use the SELinux RHEL system role; don't forget to check the role's documentation thoroughly to learn how to apply the role in different scenarios. There are many variables that you can overwrite as you can see in the role's **defaults** directory:

```
[elliot@control plays]$ cat /usr/share/ansible/roles/rhel-system-roles.selinux/
defaults/main.yml
---
selinux_state: null
selinux_policy: null

# Set up empty lists for SELinux changes.
selinux_booleans: []
selinux_fcontexts: []
selinux_logins: []
selinux_ports: []
selinux_restore_dirs: []

# Purging local modifications is disabled by default.
selinux_all_purge: no
selinux_booleans_purge: no
selinux_fcontexts_purge: no
selinux_ports_purge: no
selinux_logins_purge: no
```

Using RHEL TimeSync System Role

You can use the RHEL TimeSync system role (**rhel-system-roles.timesync**) to configure NTP on your managed hosts.

There are various variables that you can overwrite as you can see in the role's **defaults** directory:

```
[elliot@control plays]$ cat /usr/share/ansible/roles/rhel-system-roles.timesync/
defaults/main.yml
---
timesync_ntp_servers: []
timesync_ptp_domains: []
timesync_dhcp_ntp_servers: false
timesync_step_threshold: -1.0
timesync_min_sources: 1
timesync_ntp_provider: ''
```

The most important variable is the **timesync__ntp__servers** list. Each item in the **timesync__ntp__servers** will consist of different attributes, of which the following two are the most common:

- **hostname** → shows the hostname of the NTP time server.
- **iburst** → specifies that fast iburst synchronization should be used.

To demonstrate how to use the timesync rhel system role, go ahead and copy the **example-timesync-playbook.yml** playbook in **/usr/share/doc/rhel-system-roles/timesync** to your project directory:

```
[elliot@control plays]$ cp/usr/share/doc/rhel-system-roles/timesync/
example-timesync-playbook.yml .
```

Now go ahead and edit the playbook so that it has the following contents:

```
[elliot@control plays]$ cat example-timesync-playbook.yml
---
- hosts: node1
  vars:
    timesync_ntp_servers:
      - hostname: 0.north-america.pool.ntp.org
        iburst: yes
      - hostname: 1.north-america.pool.ntp.org
        iburst: yes
      - hostname: 2.north-america.pool.ntp.org
        iburst: yes
```

```
    - hostname: 3.north-america.pool.ntp.org
      iburst: yes
  tzone: America/Chicago
roles:
  - rhel-system-roles.timesync
tasks:
  - name: Set Timezone
    timezone:
      name: "{{ tzone }}"
```

Notice that I also defined a new variable named **tzone** and created a task that sets the host time zone using the **timezone** module.

Now go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook example-timesync-playbook.yml

PLAY [node1] ******************

TASK [Gathering Facts] ******************
ok: [node1]

TASK [rhel-system-roles.timesync : Check if only NTP is needed] **********
ok: [node1]

TASK [rhel-system-roles.timesync : Get chrony version] ***************
ok: [node1]

TASK [rhel-system-roles.timesync : Generate chrony.conf file] **********
changed: [node1]

TASK [rhel-system-roles.timesync : Generate chronyd sysconfig file] *********
changed: [node1]

TASK [rhel-system-roles.timesync : Update network sysconfig file] ***********
changed: [node1]

TASK [rhel-system-roles.timesync : Enable chronyd] *********
ok: [node1]

TASK [Set Timezone] ***********
changed: [node1]

RUNNING HANDLER [rhel-system-roles.timesync : restart chronyd] **************
changed: [node1]

PLAY RECAP ****************
node1: ok=18   changed=5    unreachable=0    failed=0    skipped=18
```

Now run the following two ad-hoc Ansible commands to verify that the NTP servers and time zone are updated on **node1**:

```
[elliot@control plays]$ ansible node1 -m command -a "cat /etc/chrony.conf"
node1 | CHANGED | rc=0 >>
# Ansible managed

server 0.north-america.pool.ntp.org iburst
server 1.north-america.pool.ntp.org iburst
server 2.north-america.pool.ntp.org iburst
server 3.north-america.pool.ntp.org iburst

# Allow the system clock to be stepped in the first three updates.
makestep 1.0 3

# Enable kernel synchronization of the real-time clock (RTC).
rtcsync

# Record the rate at which the system clock gains/losses time.
driftfile /var/lib/chrony/drift

[elliot@control plays]$ ansible node1 -m command -a "timedatectl"
node1 | CHANGED | rc=0 >>
               Local time: Sat 2020-11-14 21:08:46 CST
           Universal time: Sun 2020-11-15 03:08:46 UTC
                 RTC time: Sun 2020-11-15 03:08:45
                Time zone: America/Chicago (CST, -0600)
System clock synchronized: yes
              NTP service: n/a
          RTC in local TZ: no
```

Everything looks good as it is expected to be!

> Please Notice: I didn't explain how SELinux or NTP works here as this book only focuses on the Ansible side of things. It is assumed that you are already on an RHCSA skill level as Red Hat also recommends.

I hope you enjoyed learning about RHEL System roles. In the next chapter, you are going to learn the most common Ansible modules used for automating day-to-day administrative tasks and operations.

Knowledge Check

Create a playbook named **lab10.yml** that will run on **node3** and will accomplish the following tasks:

1. Create a new directory named **/web**
2. Set the SELinux file context of **/web** to **httpd_sys_content_t**

Hint: Use the selinux RHEL System role and check **/usr/share/doc/rhel-system-roles/selinux/** to see the example playbook.

Solution to the exercise is provided at the end of the book.

Chapter 11: Managing Systems with Ansible

So far, you have learned about all the core components of Ansible. Now it's time to learn about the most common Ansible modules that are used for performing daily administrative tasks.

In this chapter, you will learn how to manage users, groups, software and processes with Ansible. You will also learn how to configure networking and local storage on your Ansible managed systems.

Managing users and groups

You can use the following modules to manage users and groups in Ansible:

- **user** → manages user accounts and user attributes. For Windows targets, use the **win_user** module instead.
- **group** → manages presence of groups on a host. For Windows targets, use the **win_group** module instead.
- **pamd** → edits PAM service's type, control, module path and module arguments.
- **authorized_key** → copies SSH public key from Ansible control node to the target user **.ssh/authorized_keys** file in the managed node.
- **acl** → sets and retrieves file ACL information.
- **selogin** → manages linux user to SELinux user mapping.

You need to be aware that the **authorized_key** module doesn't generate SSH keys; To generate, SSH keys, you can use the **generate_ssh_key** option with the **user** module.

Also, keep in mind that there is no **sudo** module in Ansible. You can use Jinja2 and other modules like **lineinfile**, **blockinfile**, **replace**, or **copy** to edit sudo configurations.

Now let's create a playbook that uses some of the aforementioned modules to show you how you can manage users and groups in Ansible. But first, let's create three new users (**angela**, **tyrell**, and **darlene**) on our control node.

Go ahead and create a bash script named **adhoc.sh** that contains the following three ad-hoc commands:

```
[elliot@control plays]$ cat adhoc.sh
#!/bin/bash

ansible localhost -m user -a "name=angela uid=887
password={{ 'L!n*X'| password_hash('sha512') }} generate_ssh_key=yes"
```

```
ansible localhost -m user -a "name=tyrell uid=888
password={{ 'L!n*X' | password_hash('sha512') }} generate_ssh_key=yes"

ansible localhost -m user -a "name=darlene uid=889
password={{ 'L!n*X' | password_hash('sha512') }} generate_ssh_key=yes"
```

Notice that you used **localhost** in the ad-hoc commands to refer to the control node. The **generate_ssh_key** option is used to generate SSH key pairs for the three users.

You could have also used the **ssh_key_passphrase** option to set a non-empty ssh key pass phrase. In this case, you are going to create empty ssh key pass phrases as you didn't use **ssh_key_passphrase** option.

Now make your bash script executable and then run it:

```
[elliot@control plays]$ chmod u+x adhoc.sh
[elliot@control plays]$ ./adhoc.sh
localhost | CHANGED => {
    "changed": true,
    "comment": "",
    "create_home": true,
    "group": 1002,
    "home": "/home/angela",
    "name": "angela",
    "password": "NOT_LOGGING_PASSWORD",
    "shell": "/bin/bash",
    "ssh_fingerprint": "3072 SHA256:Cy5bC+gk5mX/6menlJOurJTLjfXrlZrcsaQpQ3zDGH0
    ansible-generated on control (RSA)",
    "ssh_key_file": "/home/angela/.ssh/id_rsa",
    "ssh_public_key": "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDUdH6F7U/e2E50n3Ob
    I7U2qbbtaZQshm8rbX5xZ7xBqOfTqnsIS8Ht1PTEuc4wPcNUTx4V3qVVt6AbVkNgONkynb7zRfc
    Ck61EQbjKeE1L7uI2ZDzpmKqUSnzvNZeDo7JyHbt1QGL9IrcBntVXNcYL+d/20eYVKUBsNU8VC0
    f5VILwSGIEcGLJ9bsnKzMqpZ1df9t/Ha3QOwmTD2AyrxPITzOBtknRuRuSP8onDI1TTJ+8NUA48
    pdJh17O44VPGKNcHIgSsbDDXpMnbRMCHf6XtgazdfwcVSZXpGs4ByrjUQyH75VDwbZKwleCGIQz
    d7YU3IKGrUKbPteDbG+tKWF+9xTObbxkNcb344fNgd4SR3AatucX9cRf1eOUBKDyeGOEdhW+4Jy
    EH4eTy8LmsL2V77bAsnOUKU60YPtZdxdZeSdpq7JQwxlTbhWs+8RUrGJwyMb97qisWfT8UuDadr
    ssM7KB5rx/h9rwr/ELHwxQN1lhtJf0tQzY9n9SIz/ewY8= ansible-generated on control",
    "state": "present",
    "system": false,
    "uid": 887
}
localhost | CHANGED => {
    "changed": true,
    "comment": "",
    "create_home": true,
    "group": 1003,
    "home": "/home/tyrell",
```

```
    "name": "tyrell",
    "password": "NOT_LOGGING_PASSWORD",
    "shell": "/bin/bash",
    "ssh_fingerprint": "3072 SHA256:10OzT7IGVdrjCk9ONgAwnvTTiuV+wp38nCju9g2eMHM
    ansible-generated on control (RSA)",
    "ssh_key_file": "/home/tyrell/.ssh/id_rsa",
    "ssh_public_key": "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQCXLWgkQ6uBPxI6IUKA
    zmshDO/C5zoBMG52q/BukV7l58F/wa5xyEPvLPtoVEst8kQ00Zmqs51inamLO45KHjoV1sG2mjb
    f4REbo6hgojAssGMvjIdPpdZtTALkLNXG+WLU5PbUnYdx5cAmnSETt1ul16GZ3Rox6grQto8sWj
    CO4d8gnqqxQevKvfv8pNqQoOMImZ5+bSNcvqBWpwfp3CsPMGC3qYIQG/wi0GAjfu+oXWp2dtA7W
    mBRHBhQ3oCcMjSjE9GujAMp0+IeMnkFZUZIf/fZrDyDFZzW740+Vxa8f5aEtgxwk2Obzkyrg4ib
    zWbmGlfmxs3OpnDDvblDRjcWmFzGLp5LONlqPStGbMWYDZi04GKMthw6/XzIkThRBVqIwNQD1N2
    W/cT4aAvcaIUc18ba8KzDQAMgso9mE1QFtLX5C+Z/3PBGp5rbQ3pTBjXKFuHh9UTrTh/A0bCi8t
    W00jURInR/gAAupZG7FmUcHhCz31pWfpmr1r1YoepoHuc= ansible-generated on control",
    "state": "present",
    "system": false,
    "uid": 888
}
localhost | CHANGED => {
    "changed": true,
    "comment": "",
    "create_home": true,
    "group": 1004,
    "home": "/home/darlene",
    "name": "darlene",
    "password": "NOT_LOGGING_PASSWORD",
    "shell": "/bin/bash",
    "ssh_fingerprint": "3072 SHA256:c9W+8AjuCsvNSMcokHzhJ91k27hd8HI1YhE6OPbkzgE
    ansible-generated on control (RSA)",
    "ssh_key_file": "/home/darlene/.ssh/id_rsa",
    "ssh_public_key": "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDNEj6cS/xgNVfO+IO5
    rq1Y8s9SSORmDrUpKmjIp6jWEq29vl9O1a1IAgmvPWAPfCYZPdtomqUbrL2R+7qdUq3AiWsLtds
    IBiXhCaY3twQlGVKW4Ou7CnohFEuFrRT1DHQN6IjTbSu/z1fT7QGfQKc4OZ6tiBaHmaFFDByzIc
    BWW5LAvGvqHF4cDFJOEyjEJ9Ih4CAYjO2smGP8tX4hFbJbKvhyI4G4C2KMCCtVlhcBnFENkB42G
    2gJEf+4hFAPdV2YR+O1vAfV7FSPu3WI6qly2BlqiYrEq17B+UodxdEg9EgWi7PivzsKdiHm5TY8
    jscpO05lIOBKpeIXpbabTuhbAFwrFbhG722H1C5QlADpPQIHsFASkehESqnSYEgvr9rMzXUfMnm
    8EJG/RvNLG4iuiJBoVom5ST3Xkg9auytLeHDciLKpCIGeWNl1MIhzeDopS2CBeh6xJPUAiShhtQ
    ZzubBAqsC3SgNCqGhsOR6BMOqtF42pncVfKOtpLF1Zyqc= ansible-generated on control",
    "state": "present",
    "system": false,
    "uid": 889
}
```

Awesome! The three users are now created. You can verify it by listing the users:

```
[elliot@control ~]$ ls -l /home
total 0
drwx------. 3 angela  angela   74 Nov 16 23:11 angela
drwx------. 3 darlene darlene  74 Nov 16 23:11 darlene
```

```
drwx------. 5 elliot  elliot  170 Nov 16 23:11 elliot
drwx------. 3 tyrell  tyrell   74 Nov 16 23:11 tyrell
```

And you can also verify that the ssh keys are generated:

```
[elliot@control plays]$ sudo ls -l /home/angela/.ssh
total 8
-rw-------. 1 angela angela 2622 Nov 16 23:11 id_rsa
-rw-r--r--. 1 angela angela  582 Nov 16 23:11 id_rsa.pub
```

Now go ahead and create a directory named **keys** in your project directory that will store the public ssh key of the newly created users:

```
[elliot@control plays]$ mkdir keys
[elliot@control plays]$ mkdir keys/angela
[elliot@control plays]$ mkdir keys/tyrell
[elliot@control plays]$ mkdir keys/darlene
[elliot@control plays]$ sudo cp -p /home/angela/.ssh/id_rsa.pub keys/angela/
[elliot@control plays]$ sudo cp -p /home/tyrell/.ssh/id_rsa.pub keys/tyrell/
[elliot@control plays]$ sudo cp -p /home/darlene/.ssh/id_rsa.pub keys/darlene/
```

Alright! Now let's create an Ansible playbook that will accomplish the following tasks on **node4**:

1. Create two groups. (**devops** and **secops**)
2. Create three users. (**angela**, **tyrell**, and **darlene**)
3. Add all three users to the **secops** group.
4. Give the **secops** group full **sudo** access with no password prompt.
5. Copy the **SSH** public key for all three users from the control node to the managed **node4**.

Go ahead and create a playbook named **manage-users-groups.yml** that has the following contents:

```
[elliot@control plays]$ cat manage-users-groups.yml
---
- name: Manage Users & Groups
  hosts: node4
  vars:
    grps:
```

```
    - devops
    - secops
  usrs:
    - username: angela
      id: 887
      pass: 'L!n*X'
    - username: tyrell
      id: 888
      pass: 'L!n*X'
    - username: darlene
      id: 889
      pass: 'L!n*X'
tasks:
  - name: Create groups
    group:
      name: "{{ item }}"
    loop: "{{ grps }}"

  - name: Create users (and add users to group secops)
    user:
      name: "{{ item.username }}"
      uid: "{{ item.id }}"
      password: "{{ item.pass | password_hash('sha512') }}"
      groups: "{{ grps[1] }}"
    loop: "{{ usrs }}"

  - name: Give secops sudo access
    copy:
      content: "%secops  ALL=(ALL)  NOPASSWD: ALL"
      dest: /etc/sudoers.d/secops
      mode: 0400

  - name: Copy users SSH public key
    authorized_key:
      user: "{{ item.username }}"
      key: "{{ lookup('file', './keys/' + item.username + '/id_rsa.pub') }}"
    loop: "{{ usrs }}"
```

Notice the use of the **lookup** plugin in the last task **Copy users SSH public key** to access the users public ssh keys files. You can also check the **authorized_key** documentation for more information and examples on how to use the **lookup** plugin:

```
[elliot@control plays]$ ansible-doc authorized_key
```

Alright! Now go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook manage-users-groups.yml
```

```
PLAY [Manage Users & Groups] ******************

TASK [Gathering Facts] **************
ok: [node4]

TASK [Create groups] ***************
changed: [node4] => (item=devops)
changed: [node4] => (item=secops)

TASK [Create users (and add users to group secops)] *************
changed: [node4] => (item={'username': 'angela', 'id': 887, 'pass': 'L!n*X'})
changed: [node4] => (item={'username': 'tyrell', 'id': 888, 'pass': 'L!n*X'})
changed: [node4] => (item={'username': 'darlene', 'id': 889, 'pass': 'L!n*X'})

TASK [Give secops sudo access] *****************
changed: [node4]

TASK [Copy users SSH public key] *****************
changed: [node4] => (item={'username': 'angela', 'id': 887, 'pass': 'L!n*X'})
changed: [node4] => (item={'username': 'tyrell', 'id': 888, 'pass': 'L!n*X'})
changed: [node4] => (item={'username': 'darlene', 'id': 889, 'pass': 'L!n*X'})

PLAY RECAP *****************************
node4: ok=5    changed=4    unreachable=0    failed=0    skipped=0
```

Everything looks good!

You should now be able to ssh from the control node to **node4** with any of the three users (**angela**, **tyrell**, and **darlene**).

```
[angela@control ~]$ ssh node4
angela@node4:~$ exit
[tyrell@control ~]$ ssh node4
tyrell@node4:~$ exit
[darlene@control ~]$ ssh node4
darlene@node4:~$ exit
```

Managing software

You can use the following modules to manage software in Ansible:

- **package** → installs, upgrade and removes packages using the underlying OS package manager. For Windows targets, use the **win_package** module instead.
- **yum** → installs, upgrade, downgrades, removes, and lists packages and groups with the "yum" package manager.
- **apt** → manages "apt" packages (such as for Debian/Ubuntu).
- **yum_repository** → adds or removes YUM repositories in RPM-based Linux distributions.
- **package_facts** → returns information about installed packages as facts.
- **rpm_key** → adds or removes a gpg key to your rpm database.
- **redhat_subscription** → manages registration and subscription to the Red Hat Subscription Management entitlement platform using the 'subscription-manager' command.
- **rhn_register** → manages registration to the Red Hat Network.
- **rhn_channel** → adds or removes Red Hat software channels.

Now let's create a playbook that uses some of the aforementioned modules to show you how you can manage software in Ansible. The playbook will accomplish the following things on **node1**:

1. Create a new repository with the following attributes
    - **file** → zabbix.repo
    - **name** → "zabbix-monitoring"
    - **baseurl** → https://repo.zabbix.com/zabbix/5.2/rhel/8/x86_64/
    - **description** → "Zabbix 5.2 Repo"
    - **enabled** → yes
    - **gpgcheck** → no
2. Install the **zabbix-agent** package.
3. Display the information on the installed **zabbix-agent** package.

Go ahead and create a playbook named **manage-software.yml** that has the following contents:

```
[elliot@control plays]$ cat manage-software.yml
---
- name: Manage Software
  hosts: node1
  vars:
    pkg_name: zabbix-agent
  tasks:
    - name: Create a new repo
      yum_repository:
        file: zabbix
        name: zabbix-monitoring
        baseurl: https://repo.zabbix.com/zabbix/5.2/rhel/8/x86_64/
        description: "Zabbix 5.2 Repo"
        enabled: yes
        gpgcheck: no

    - name: Install zabbix agent
      yum:
        name: "{{ pkg_name }}"
        state: present

    - name: Get package facts
      package_facts:
        manager: auto

    - name: Display zabbix-agent package facts
      debug:
        var: ansible_facts.packages[pkg_name]
      when: pkg_name in ansible_facts.packages
```

Now go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook manage-software.yml

PLAY [Manage Software] ************

TASK [Gathering Facts] ************
ok: [node1]

TASK [Create a new repo] *************
changed: [node1]

TASK [Install zabbix agent] ****************
changed: [node1]

TASK [Get package facts] *************
ok: [node1]

TASK [Display zabbix-agent package facts] ****************
```

```
ok: [node1] => {
    "ansible_facts.packages[pkg_name]": [
        {
            "arch": "x86_64",
            "epoch": null,
            "name": "zabbix-agent",
            "release": "1.el8",
            "source": "rpm",
            "version": "5.2.1"
        }
    ]
}

PLAY RECAP ******************
node1: ok=5    changed=2    unreachable=0    failed=0    skipped=0
```

Now let's quickly verify the contents of the newly created **zabbix** repository:

```
[elliot@control plays]$ ansible node1 -m command -a "cat /etc/yum.repos.d/zabbix.repo"
node1 | CHANGED | rc=0 >>
[zabbix-monitoring]
baseurl = https://repo.zabbix.com/zabbix/5.2/rhel/8/x86_64/
enabled = 1
gpgcheck = 0
name = Zabbix 5.2 Repo
```

As you can see; the repository is configured exactly how you specified in the playbook.

By default, Ansible doesn't gather package facts. Thus, I used the **package_facts** module in the playbook to gather package information as facts. You can check the **package_facts** documentation for more information:

```
[elliot@control plays]$ ansible-doc package_facts
```

Managing processes and tasks

You can use the following modules to manage processes and tasks in Ansible:

- **cron** → use this module to schedule cron jobs.
- **at** → use this module to schedule a command or a script to run once in the future.
- **service** → use this module to manage services (start, restart, enable on boot, disable, etc).
- **service_facts** → returns service state information as fact data for various service management utilities.
- **systemd** → controls systemd services on remote hosts.

Keep in mind that the **cron** module has a **name** option which is used to uniquely identify entries in the crontab. The **name** option has no meaning for Cron, but it helps Ansible in managing crontab entries.

For example, Ansible uses the **name** option so that it can remove specific entries in crontab. You will see how it works in the next playbook.

Now let's create a playbook that schedules a new cron job on **node2**. The cron job will run as user **elliot** and will append the message "Two minutes have passed!" to the system log and will run every two minutes. You should let the cron job run only run twice!

Go ahead and create a playbook named **cronjob.yml** that has the following contents:

```
[elliot@control plays]$ cat cronjob.yml
---
- name: Managing cron jobs
  hosts: node2
  tasks:
    - name: Run this cron job every two mins
      cron:
        name: "two-mins"
        user: elliot
        job: logger 'Two minutes have passed!'
        minute: '*/2'

    - name: wait for five minutes
      pause:
        minutes: 5

    - name: Remove the two-mins cron job
      cron:
        name: "two-mins"
```

```
        user: elliot
        state: absent
```

Notice that I also used the **pause** module to wait for five minutes (let the job run twice) before removing the "**two-mins**" cron job.

Go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook cronjob.yml

PLAY [Managing cron jobs] ******************

TASK [Gathering Facts] ************
ok: [node2]

TASK [Run this cron job every two mins] ******************
changed: [node2]

TASK [wait for five minutes] ***************************
Pausing for 300 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [node2]

TASK [Remove the two-mins cron job] *************
changed: [node2]

PLAY RECAP *****************
node2: ok=4    changed=2    unreachable=0    failed=0    skipped=0
```

As you can; see the playbook created the "**two-mins**" cron job then waited for five minutes and then removed the "**two-mins**" cron job.

You can check the contents of the **/var/log/messages** file to see if the logger has successfully appended the messages to the system log (**syslog**):

```
[elliot@control plays]$ ansible node2 -m shell -a "grep passed /var/log/messages"
node2 | CHANGED | rc=0 >>
Nov 17 05:30:37 node2 ansible-cron[295224]: Invoked with name=two-mins
user=elliot job=logger 'Two minutes have passed!' minute=*/2 state=present
backup=False hour=* day=* month=* weekday=* reboot=False disabled=False
cron_file=None special_time=None env=None insertafter=None insertbefore=None
Nov 17 05:32:01 node2 elliot[295290]: Two minutes have passed!
Nov 17 05:34:01 node2 elliot[295349]: Two minutes have passed!
```

Indeed! The "**two-mins**" cron job ran exactly twice as you wanted.

You should also know that there is no dedicated module to interact with Linux processes in Ansible. If you want to send signals to a process; you can just use regular Linux commands like **kill**, **pkill**, **killall** with the **command**, **shell**, or **raw** Ansible modules.

Configuring local storage

You can use the following modules to configure local storage in Ansible:

- **parted** → This module allows configuring block device partition using the 'parted' command line tool.
- **lvg** → This module creates, removes or resizes volume groups.
- **lvol** → This module creates, removes or resizes logical volumes.
- **filesystem** → This module creates a filesystem.
- **mount** → This module controls active and configured mount points in '**/etc/fstab**'.
- **vdo** → This module controls the VDO dedupe and compression device.

Now let's create a new playbook that will accomplish the following tasks on **node4**:

1. Create two new 2GB partitions on any available secondary disk (**/dev/sdc** in my case).
2. Create a new volume group named **dbvg** that is composed of the two partitions you created from task 1 (**/dev/sdc1** and **/dev/sdc2** in my case).
3. Inside the **dbvg** volume group, create a logical volume named **dblv** of size 3GB.
4. Create an **ext4** filesystem on the **dblv** logical file.
5. Create a new directory **/database** owned with permissions set to **0755**.
6. Permanently mount the **dblv** ext4 filesystem on **/database**.

You need to create (or attach) a new secondary disk (size at least 4 GB) to your **node4** virtual machine to be able to follow along with the playbook.

I have attached a new 10 GB disk (**/dev/sdc**) to my **node4** virtual machine:

```
[elliot@control plays]$ ansible node4 -m command -a "lsblk"
node4 | CHANGED | rc=0 >>
NAME    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda       8:0    0   30G  0 disk
 sda1    8:1    0 29.9G  0 part /
 sda14   8:14   0    4M  0 part
 sda15   8:15   0  106M  0 part /boot/efi
sdb       8:16   0    4G  0 disk
 sdb1    8:17   0    4G  0 part /mnt
sdc       8:32   0   10G  0 disk
sr0      11:0    1 1024M  0 rom
```

Go ahead and create a new playbook named **manage-storage.yml** that has the following contents:

```
[elliot@control plays]$ cat manage-storage.yml
---
- name: Manage Storage
  hosts: node4
  vars:
    partitions:
      - number: 1
        start: 0%
        end: 2GiB
      - number: 2
        start: 2GiB
        end: 4GiB
    vgs:
      - name: dbvg
        devices: ['/dev/sdc1','/dev/sdc2']
    lvs:
      - name: dblv
        size: 3G
        vgrp: dbvg
        mnt: /database
  tasks:
    # Replace /dev/sdc with your secondary disk name
    - name: Create two new partitions on /dev/sdc
      parted:
        device: /dev/sdc
        number: "{{ item.number }}"
        part_start: "{{ item.start }}"
        part_end: "{{ item.end }}"
        state: present
      loop: "{{ partitions }}"

    - name: Create volume group
      lvg:
        vg: "{{ item.name }}"
        pvs: "{{ item.devices }}"
      loop: "{{ vgs }}"

    - name: Create logical volume
      lvol:
        vg: "{{ item.vgrp }}"
        lv: "{{ item.name }}"
        size: "{{ item.size }}"
      loop: "{{ lvs }}"

    - name: Create filesystem on logical volume
      filesystem:
```

```
        dev: "/dev/{{ item.vgrp }}/{{ item.name }}"
        fstype: ext4
      loop: "{{ lvs }}"

    - name: Create /database directory
      file:
        path: /database
        mode: 0755
        state: directory

    - name: Mount logical volume
      mount:
        src: "/dev/{{ item.vgrp }}/{{ item.name }}"
        path: "{{ item.mnt }}"
        fstype: ext4
        state: mounted
      loop: "{{ lvs }}"
```

Now go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook manage-storage.yml

PLAY [Manage Storage] ***********************

TASK [Gathering Facts] ************************
ok: [node4]

TASK [Create two new partitions on /dev/sdc] *******
changed: [node4] => (item={'number': 1, 'start': '0%', 'end': '2GiB'})
changed: [node4] => (item={'number': 2, 'start': '2GiB', 'end': '4GiB'})

TASK [Create volume group] **************
changed: [node4] => (item={'name': 'dbvg', 'devices': ['/dev/sdc1', '/dev/sdc2']})

TASK [Create logical volume] ****************
changed: [node4] => (item={'name': 'dblv', 'size': '3G', 'vgrp': 'dbvg',
'mnt': '/database'})

TASK [Create filesystem on logical volume] ********=
changed: [node4] => (item={'name': 'dblv', 'size': '3G', 'vgrp': 'dbvg',
'mnt': '/database'})

TASK [Create /database directory] **********************
changed: [node4]

TASK [Mount logical volume] *********************
changed: [node4] => (item={'name': 'dblv', 'size': '3G', 'vgrp': 'dbvg',
'mnt': '/database'})
```

```
PLAY RECAP *********************
node4: ok=7    changed=6    unreachable=0    failed=0    skipped=0
```

Looks good so far! Now let's run an ad-hoc Ansible command to verify everything is correct on **node4**:

```
[elliot@control plays]$ ansible node4 -m shell -a "vgs;lvs;tail -1
/etc/fstab;ls -ld /database;lsblk"

node4 | CHANGED | rc=0 >>
  VG   #PV #LV #SN Attr   VSize VFree
  dbvg   2   1   0 wz--n- 3.99g 1016.00m
  LV   VG   Attr       LSize Pool Origin Data%  Meta%  Move Log Cpy%Sync Convert
  dblv dbvg -wi-ao---- 3.00g
/dev/dbvg/dblv /database ext4 defaults 0 0
drwxr-xr-x 3 root root 4096 Nov 17 23:58 /database
NAME          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda             8:0    0   30G  0 disk
 sda1           8:1    0 29.9G  0 part /
 sda14          8:14   0    4M  0 part
 sda15          8:15   0  106M  0 part /boot/efi
sdb             8:16   0    4G  0 disk
 sdb1           8:17   0    4G  0 part /mnt
sdc             8:32   0   10G  0 disk
 sdc1           8:33   0    2G  0 part
   dbvg-dblv 253:0    0    3G  0 lvm  /database
 sdc2           8:34   0    2G  0 part
   dbvg-dblv 253:0    0    3G  0 lvm  /database
sr0            11:0    1 1024M  0 rom
```

Everything is correct; by now you should have realized how powerful Ansible is! Imagine trying to write this Ansible playbook as a bash script ... Good Luck!

Again; a friendly reminder that you don't need to memorize any of the modules options. Just consult with the documentation pages:

```
[elliot@control plays]$ ansible-doc parted
[elliot@control plays]$ ansible-doc lvg
[elliot@control plays]$ ansible-doc lvol
[elliot@control plays]$ ansible-doc filesystem
[elliot@control plays]$ ansible-doc file
[elliot@control plays]$ ansible-doc mounted
```

Configuring network interfaces

You can use the following modules to configure network interfaces in Ansible:

- **nmcli** → manages the network devices. Create, modify and manage various connection and device type e.g., ethernet, teams, bonds, vlans, etc.
- **hostname** → set system's hostname, supports most OSs/Distributions, including those using systemd.
- **firewalld** → This module allows for addition or deletion of services and ports (either TCP or UDP) in either running or permanent firewalld rules.

Now let's create a new playbook that will accomplish the following tasks on **node1**:

1. Configure the secondary network interface (**eth1** in my case) with the following settings:
    - Connection Name → ether-two
    - Type → ethernet
    - IPv4 Address → 192.168.177.3
    - Subnet Mask → 255.255.0.0
    - DNS Servers → 8.8.8.8 and 8.8.4.4
2. Change the hostname of **node1** to **node1.linuxhandbook.local**
3. Allow **HTTP** and **HTTPS** traffic in the firewall public zone.

You need to create (or attach) a new secondary network interface to follow along with the playbook.

You can run the following ad-hoc command to list the available network interfaces on **node1**:

```
[elliot@control plays]$ ansible node1 -m setup -a "filter=ansible_interfaces"
node1 | SUCCESS => {
    "ansible_facts": {
        "ansible_interfaces": [
            "eth0",
            "eth1",
            "lo"
        ],
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false
}
```

As you can see; I have two network interfaces: **eth0** and **eth1**.

```
[elliot@node1 ~]$ nmcli connection show
NAME                UUID                                  TYPE      DEVICE
System eth0         5fb06bd0-0bb0-7ffb-45f1-d6edd65f3e03  ethernet  eth0
Wired connection 1  f536da88-7978-33dc-8afc-632274ba5661  ethernet  eth1
```

Go ahead and create a new playbook named **manage-network.yml** that has the following contents:

```
[elliot@control plays]$ cat manage-network.yml
---
- name: Configure a NIC
  hosts: node1
  tasks:
    - name: Configure eth1
      nmcli:
        ifname: eth1
        conn_name: ether-two
        type: ethernet
        ip4: 192.168.177.3/16
        dns4:
          - 8.8.8.8
          - 8.8.4.4
        state: present

    - name: Set hostname
      hostname:
        name: node1.linuxhandbook.local

    - name: Enable http and https in firewall public zone
      firewalld:
        zone: public
        service: "{{ item }}"
        permanent: yes
        state: enabled
      loop: [http, https]
      notify: restart firewalld

  handlers:
    - name: restart firewalld
      service:
        name: firewalld
        state: restarted
```

Now go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook manage-network.yml
```

```
PLAY [Configure a NIC] **************

TASK [Gathering Facts] ***********
ok: [node1]

TASK [Configure eth1] ***********
changed: [node1]

TASK [Set hostname] ************
changed: [node1]

TASK [Enable http and https in firewall public zone] *************
changed: [node1] => (item=http)
changed: [node1] => (item=https)

RUNNING HANDLER [restart firewalld] ***************
changed: [node1]

PLAY RECAP ********************
node1: ok=5    changed=4    unreachable=0    failed=0    skipped=0
```

Now login to **node1.linuxhandbook.local** (hostname changed!) and check **eth1** configuration, DNS, and the firewall public zone settings:

```
[elliot@node1 ~]$ hostname
node1.linuxhandbook.local

[elliot@node1 ~]$ sudo firewall-cmd --zone public --list-services
cockpit dhcpv6-client http https ssh

[elliot@node1 ~]$ nmcli connection show
NAME         UUID                                  TYPE      DEVICE
System eth0  5fb06bd0-0bb0-7ffb-45f1-d6edd65f3e03  ethernet  eth0
ether-two    eb410c56-4f1e-4df6-97f4-933334a84c0f  ethernet  eth1

[elliot@node1 ~]$ cat /etc/resolv.conf
# Generated by NetworkManager
search t3nnzgt5u3xefmnwfajknnwnud.ux.internal.cloudapp.net
nameserver 168.63.129.16
nameserver 8.8.8.8
nameserver 8.8.4.4

[elliot@node1 ~]$ ifconfig eth1
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.177.3  netmask 255.255.0.0  broadcast 192.168.255.255
        inet6 fe80::3ec4:612:4aa2:723  prefixlen 64  scopeid 0x20<link>
        ether 00:0d:3a:84:a3:2b  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
```

```
        TX packets 32  bytes 2214 (2.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0


[elliot@node1 ~]$ ping 192.168.177.3
PING 192.168.177.3 (192.168.177.3) 56(84) bytes of data.
64 bytes from 192.168.177.3: icmp_seq=1 ttl=64 time=0.030 ms
64 bytes from 192.168.177.3: icmp_seq=2 ttl=64 time=0.048 ms
^C
--- 192.168.177.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 41ms
rtt min/avg/max/mdev = 0.030/0.039/0.048/0.009 ms
```

Everything looks good!

> Keep in mind that you also have the option to use the network RHEL System Role when you are dealing with RHEL managed hosts.

To summarize; Figure 10 helps you categorize and identify the most commonly used Ansible modules for managing Linux systems.

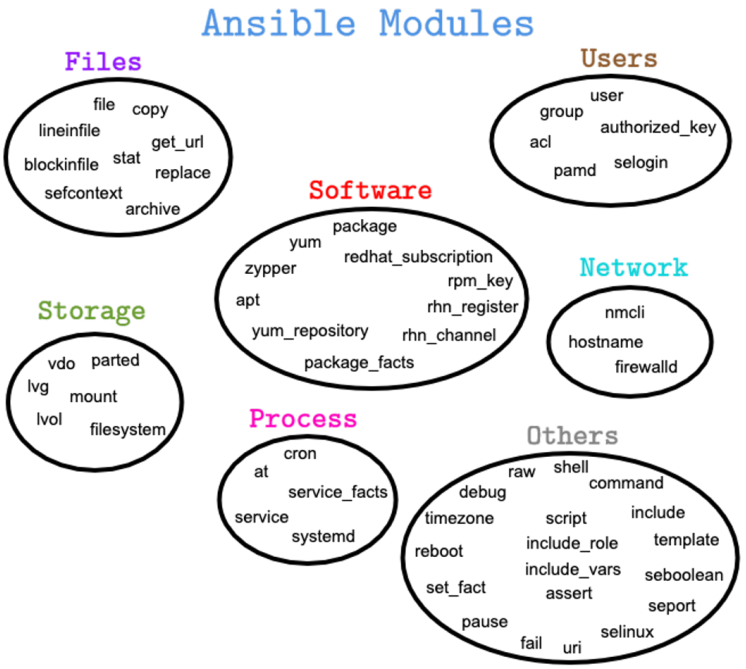

Figure 10: Most Common Ansible Modules

This brings us to the end of this chapter! In the next chapter, you are going to learn various troubleshooting techniques in Ansible.

Knowledge Check

Create a playbook named **lab11.yml** that will accomplish the following tasks:

1. Create a cron job as user **root** named **clean-tmp** on all managed nodes.
2. The cron job will run every day at midnight.
3. The cron job will remove all the files and directories in **/tmp**.

Hint: Use the **cron** module.

Solution to the exercise is provided at the end of the book.

Chapter 12: Ansible Troubleshooting

We all dream of a world where we never make mistakes or errors while running our Ansible playbooks.

In reality, such world doesn't exist and so you need to have troubleshooting skills, so you are ready to deal with errors in your playbooks.

In this chapter, you will learn how to enable logging in your playbooks. You will also learn a few other Ansible modules that can help you with troubleshooting. Finally, you will learn how to troubleshoot connectivity issues in Ansible.

Enable Ansible logging

By default, Ansible is not configured to log its output anywhere. You can however change this behavior by setting the **log_path** configuration setting in your Ansible configuration file (**ansible.cfg**) to allow Ansible to log its output to a specific destination.

To demonstrate, let's first set **log_path** in **ansible.cfg** as shown in Figure 11:

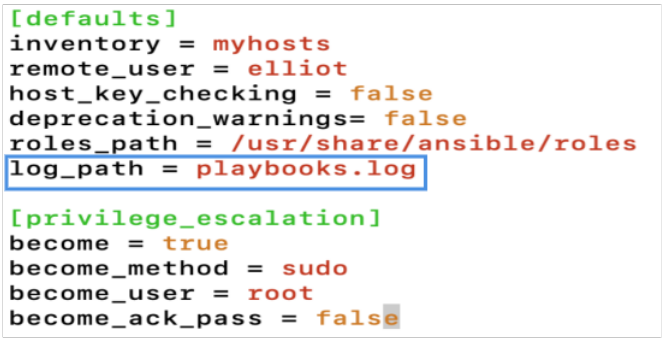

Figure 11: Setting **log_path**

This would enable Ansible playbooks and ad-hoc commands to log its output to a file named **playbooks.log** in your project directory.

Now go ahead and create a playbook named **faulty-playbook.yml** that contains an error as follows:

```
[elliot@control plays]$ cat faulty-playbook.yml
---
- name: Playbook logging enabled
  hosts: node1
  var:
    blog: "Linux Handbook"
```

```
  tasks:
    - name: Print favorite Linux Blog
      debug:
        msg: "{{ blog }}"
```

Go ahead and run the playbook:

```
[elliot@control plays]$ ansible-playbook faulty-playbook.yml
ERROR! 'var' is not a valid attribute for a Play

The error appears to be in '/home/elliot/plays/faulty-playbook.yml': line 2,
column 3, but may be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

---
- name: Playbook logging enabled
  ^ here
```

You can see there is an error 'var' is not a valid attribute for a Play. Now check the contents of the **playbooks.log** file:

```
[elliot@control plays]$ cat playbooks.log
2020-11-21 02:24:36,185 p=1369471 u=elliot n=ansible |
ERROR! 'var' is not a valid attribute for a Play

The error appears to be in '/home/elliot/plays/faulty-playbook.yml': line 2,
column 3, but may be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

---
- name: Playbook logging enabled
  ^ here
```

As you can see; it captured and timestamped the output of the playbook. You should also consider using log rotation if you are going to enable logging in Ansible.

You can fix the error in the **faulty-playbook.yml** playbook by changing **var** to **vars**.

Notice that it's a good practice to use the **--syntax-check** option to check for errors before running any playbook:

```
[elliot@control plays]$ ansible-playbook --syntax-check faulty-playbook.yml
ERROR! 'var' is not a valid attribute for a Play

The error appears to be in '/home/elliot/plays/faulty-playbook.yml': line 2,
column 3, but may be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

---
- name: Playbook logging enabled
  ^ here
```

Using the debug module

The **debug** module comes in handy when troubleshooting as it can show variables values in playbooks. You can also use the **debug** module to show messages in specific error situations.

You can specify the verbosity option to control when a debug task should execute.

To demonstrate, go ahead and create a playbook named **debugging.yml** playbook with the following contents:

```
[elliot@control plays]$ cat debugging.yml
---
- name: debug demo
  hosts: node1
  tasks:
    - name: show os distro
      debug:
        msg: "{{ ansible_distribution }}"

    - name: show ip address
      debug:
        msg: "{{ ansible_facts.default_ipv4.address }}"
        verbosity: 2
```

Now go ahead and run the playbook with one level of verbosity by using the **-v** option:

```
[elliot@control plays]$ ansible-playbook -v debugging.yml
Using /home/elliot/plays/ansible.cfg as config file

PLAY [debug demo] *************************
```

```
TASK [Gathering Facts] *************************
ok: [node1]

TASK [show os distro] ****************
ok: [node1] => {
    "msg": "CentOS"
}

TASK [show ip address] *********************
skipping: [node1] => {"skipped_reason": "Verbosity threshold not met."}

PLAY RECAP ******************
node1: ok=2    changed=0    unreachable=0    failed=0    skipped=1
```

Notice how the second task [show ip address] was skipped! That's because you specified **verbosity: 2**. To trigger task two; you need to specify at least two levels of verbosity (**-vv**) while running the playbook as follows:

```
[elliot@control plays]$ ansible-playbook -vv debugging.yml
ansible-playbook 2.9.14
  config file = /home/elliot/plays/ansible.cfg
  configured module search path = ['/home/elliot/.ansible/plugins/modules',
  '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3.6/site-packages/ansible
  executable location = /usr/bin/ansible-playbook
  python version = 3.6.8 (default, Dec  5 2019, 15:45:45) [GCC 8.3.1 20191121]
Using /home/elliot/plays/ansible.cfg as config file

PLAYBOOK: debugging.yml ************
1 plays in debugging.yml

PLAY [debug demo] ********************

TASK [Gathering Facts] *****************
task path: /home/elliot/plays/debugging.yml:2
ok: [node1]
META: ran handlers

TASK [show os distro] ************
task path: /home/elliot/plays/debugging.yml:5
ok: [node1] => {
    "msg": "CentOS"
}

TASK [show ip address] ***************
task path: /home/elliot/plays/debugging.yml:9
ok: [node1] => {
    "msg": "10.0.0.5"
}
```

```
META: ran handlers
META: ran handlers

PLAY RECAP **********************
node1: ok=3    changed=0    unreachable=0    failed=0    skipped=0
```

Task 2 [show ip address] ran this time as we have specified two levels of verbosity.

You can check the **ansible-playbook** man page to read more about the different verbosity options that you can use when running Ansible playbooks. Also, make sure you check the **debug** module documentation page:

```
[elliot@control plays]$ man ansible-playbook
[elliot@control plays]$ ansible-doc debug
```

Using the assert module

The **assert** module can also come in handy while troubleshooting as you can use it to test whether a specific condition is met.

To demonstrate, go ahead and create a playbook named **assert.yml** that has the following contents:

```
[elliot@control plays]$ cat assert.yml
---
- name:
  hosts: node1
  tasks:
    - name: check free memory
      assert:
        that: "{{ ansible_facts['memfree_mb'] > 500 }}"
        fail_msg: "Low on memory!"

    - name: get file info
      stat:
        path: /etc/motd
      register: motd

    - name: assert /etc/motd is a directory
      assert:
        that: motd.stat.isdir
        fail_msg: "/etc/motd is not a directory!"
```

This playbook contains three tasks. The first task uses the **assert** module to check if node1 has more than 500 MB of free RAM; if it doesn't, it will fail and displays the message "Low on memory!".

The second task uses the **stat** module to retrieve **/etc/motd** file facts (similar to the stat command in Linux) and then registers it to the **motd** variable.

Finally, the third task uses the **assert** module to check if **/etc/motd** is a directory which will indeed fail with the message "/etc/motd is not a directory!"

Now go ahead and run the playbook to see all this in action:

```
[elliot@control plays]$ ansible-playbook assert.yml

PLAY [node1] **************

TASK [Gathering Facts] *************************
ok: [node1]

TASK [check free memory] ****************
```

```
ok: [node1] => {
    "changed": false,
    "msg": "All assertions passed"
}

TASK [get file info] ************************
ok: [node1]

TASK [assert /etc/motd is a directory] ************
fatal: [node1]: FAILED! => {
    "assertion": "motd.stat.isdir",
    "changed": false,
    "evaluated_to": false,
    "msg": "/etc/motd is not a directory!"
}

PLAY RECAP **********************************
node1: ok=3    changed=0    unreachable=0    failed=1    skipped=0
```

You can check the **assert** and **stat** documentation pages to see more examples of how you can use both modules:

```
[elliot@control plays]$ ansible-doc assert
[elliot@control plays]$ ansible-doc stat
```

Running playbooks in check mode

You can use the **--check (-C)** option with the **ansible-playbook** command to do a dry run of a playbook. This will basically just predict and show you what will happen when running the playbook without actually changing anything.

You can also set the boolean **check_mode: yes** within a task to always run that specific task in check mode. On the other hand, you can set the boolean **check_mode: no** so that a task will never run in check mode.

To demonstrate, go ahead and create a playbook named **check-mode.yml** that has the following contents:

```
[elliot@control plays]$ cat check-mode.yml
---
- name: Check mode demo
  hosts: node1
  tasks:
    - name: create a file
      file:
        path: /tmp/file1
        state: touch

    - name: create a second file
      file:
        path: /tmp/file2
        state: touch
      check mode: yes

    - name: create a third file
      file:
        path: /tmp/file3
        state: touch
      check_mode: no
```

Notice that the second task **name: create a second file** will always run in check mode and the third task **name: create a third file** create a third file task will never run in check mode.

Go ahead and run the playbook in check mode:

```
[elliot@control plays]$ ansible-playbook --check check-mode.yml

PLAY [Check mode demo] **********

TASK [Gathering Facts] ******************
ok: [node1]
```

```
TASK [create a file] *****************
ok: [node1]

TASK [create a second file] ******
ok: [node1]

TASK [create a third file] **********
changed: [node1]

PLAY RECAP ****************
node1: ok=4    changed=1    unreachable=0    failed=0    skipped=0
```

Notice that the first two tasks ran in check mode while the third task did actually run on node1 and created the file **/tmp/file3**:

```
[elliot@control plays]$ ansible node1 -m shell -a "ls /tmp/file*"
node1 | CHANGED | rc=0 >>
/tmp/file3
```

You can also use the **--diff** option along with the **--check** option to see the differences that would be made by template files on managed hosts.

Troubleshooting connectivity problems

You may encounter connectivity issues in Ansible; connectivity problems usually fall in one of the following two categories:

1. Network → Issues connecting to managed nodes.
2. Authentication → Issues running tasks on the managed nodes as the target user.

If you are facing network issues, then you may need to check the following settings:

1. If a managed host has more than one IP address or hostname; you can use **ansible_host** in your Ansible inventory file to specify the IP address or hostname you want to connect with.

   node1.linuxhandbook.local ansible_ host=192.169.177.122

2. If SSH is configured to listen on a different port other than the default (22). Then you can use **ansible_port** in your Ansible inventory file to specify the SSH port to connect on.

   node2.linuxhandbook.local ansible_port=5555 ansible_ host=192.169.177.122

If you are facing authentication issues, then you need to check the following settings:

1. In the ansible configuration file **ansible.cfg**, verify that the **remote_user** setting is set and make sure that the remote user exists on all the managed nodes.
2. Make sure you can SSH into the managed nodes as the remote user. Also, verify that the SSH public key of the remote user is copied to all the managed nodes.
3. Verify the become and **become_user** configuration settings are set properly in **ansible.cfg**.
4. Verify that the sudo configuration on all the managed nodes allows the remote user to escalate privileges to the **become_user** (root).

Using ping module to test connectivity

The **ping** module comes in very handy when troubleshooting connectivity. You can run the following ad-hoc command to test if Ansible can reach all your managed nodes:

```
[elliot@control plays]$ ansible all -m ping
node4 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
```

```
}
node1 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
node2 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
node3 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
```

You can also use the **--become** option along with **ping** module to test if the Ansible remote user is able to escalate privileges:

```
[elliot@control plays]$ ansible node1 -m ping --become
node1 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
```

Testing Connectivity to webservices endpoints

The **uri** module comes in very handy when you are testing connectivity to web services endpoints.

For example, the following ad-hoc command will test whether **node2** can connect to its local server webpage:

```
[elliot@control plays]$ ansible node2 -m uri -a "url=http://localhost"
node2 | SUCCESS => {
```

```
    "accept_ranges": "bytes",
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "connection": "close",
    "content_length": "124",
    "content_type": "text/html; charset=UTF-8",
    "cookies": {},
    "cookies_string": "",
    "date": "Sun, 22 Nov 2020 01:48:21 GMT",
    "elapsed": 0,
    "etag": "\"7c-5b3f366f0ce84\"",
    "last_modified": "Fri, 13 Nov 2020 02:01:09 GMT",
    "msg": "OK (124 bytes)",
    "redirected": false,
    "server": "Apache/2.4.37 (centos)",
    "status": 200,
    "url": "http://localhost"
}
```

As you can; a connection was successful as **status: 200** response was returned which indicates a successful connection.

This brings us to the end our book. You have now mastered one of the most popular DevOps tools and you are also very well equipped to pass the RHCE EX294 examination to become a Red Hat Certified Engineer.

Good luck from the entire team of Linux Handbook!

Knowledge Check

Create a playbook named **lab12.yml** that has the following contents:

```
---
- name: Fix the error
  hosts: localhost
  vars:
    grps: ["devops","secops","netops"]
  tasks:
    - name: create groups
      group:
        name: "{{ item }}"
      loop: {{ grps }}
```

Then carry out the following tasks:

1. Enable Ansible logging and save all Ansible logs to the file **lab12.out**.
2. Run the **lab12.yml** playbook in **check mode**.
3. Spot the error in the output and fix it; then run the playbook.

Hint: Set **log_path** in **ansible.cfg**

Solution to the exercise is provided at the end of the book.

Chapter 13: Final Sample Exam

Before you attempt the ***RHCE*** exam; I have few words of advice to you that I believe will help you a lot when you write your exam. So without further ado, here are my advices:

- Practice, Practice, and Practice some more!
- You will have access to the Ansible documentation webpage during the exam; make use of it!
- **ansible-doc** is your best friend!
- Try not to spend more than **15 minutes** on a single task.
- You are allowed to take breaks during the exam; so take one, trust me, they help a lot!
- When you get stuck on one task, skip it and move on to solve other tasks, then if time permits, go back to try and solve the tasks that you skipped.

Now it's time for the final sample exam.

To make the most out of this sample exam, treat it as if it's the real ***RHCE*** exam and so try to finish all the sample exam tasks in less than four hours. Also, take a 10 minutes break after two hours to help you relax and regain focus.

As with the real exam, no answers to the sample exam questions will be provided.

I will only give you few hints for some of the tasks.

Exam Requirements

There are 15 questions in total.

You will need five RHEL 8 (or CentOS 8) virtual machines to be able to successfully complete all questions.

One VM will be configured as an Ansible **control** node. Other four VMs will be used to apply playbooks to solve the sample exam questions.

The following **FQDNs** will be used throughout the sample exam.

- **control.linuxhandbook.local** – Ansible control node
- **node1.linuxhandbook.local** - managed host
- **node2.linuxhandbook.local** - managed host
- **node3.linuxhandbook.local** - managed host
- **node4.linuxhandbook.local** - managed host

There are few other requirements that should be met before starting the sample exam:

- The **control** node has passwordless SSH access to all managed servers (using the **root** user).
- There are no regular users created on any of the servers.

I wish you the best of luck in your sample exam!

Task 1: Installing and configuring Ansible

Install **ansible** on the control node and configure the following:

- On the control node; create a regular user named **automation** with the password **G*Auto1!**. Use this user for all sample exam tasks and playbooks, unless you are working on **Task 2** that requires creating the **automation** user on the managed hosts.
- All playbooks and other Ansible configuration that you create for this sample exam should be stored in **/home/automation/plays**.

Create an Ansible configuration file **/home/automation/plays/ansible.cfg** to meet the following requirements:

- Create a new directory **/home/automation/plays/roles** and set it as Ansible roles path.
- The inventory file path is **/home/automation/plays/inventory**.
- Privilege escalation is **enabled** by default.
- Ansible should be able to manage **10** hosts simultaneously.
- Ansible should connect to all managed nodes using the **automation** user.

Create an inventory file **/home/automation/plays/inventory** with the following:

- **node1.linuxhandbook.local** is a member of the **dev** group.
- **node2.linuxhandbook.local** is a member of the **test** group.
- **node3.linuxhandbook.local** is a member of the **test** group.
- **node4.linuxhandbook.local** is a member of the **prod** group.

Hint: set **forks=10** in **ansible.cfg**

Task 2: Running Ad-Hoc Commands

Generate an SSH key pair on the **control** node for the **automation** user. You can perform this step manually.

Write a bash script **adhoc.sh** that uses Ansible ad-hoc commands to achieve the following:

- User **automation** is created on all managed hosts.
- The SSH public key that you just generated is copied to all managed hosts for the **automation** user.
- The **automation** user has full sudo access on all managed nodes without having to provide a password.

After running the **adhoc.sh** bash script, you should be able to SSH into all managed hosts using the **automation** user without a password, as well as a run all privileged commands.

Hint: Specify the **root** user when running the ad-hoc commands.

Here is an example!

```
ansible all -u root -m authorized_key -a "user=automation
 state=present key={{ lookup('file','/home/automation/.ssh/id_rsa.pub')
 }}"
```

Task 3: Message of the day

Create a playbook **motd.yml** that runs on all managed hosts and does the following:

- The playbook should overwrite the contents of the **/etc/motd** file with the contents of the **message** variable. The value of the **message** variable varies according to the inventory host group.
- For hosts in the **dev** group; **/etc/motd** should have the following message "This is a dev server."
- For hosts in the **test** group; **/etc/motd** should have the following message "This is a test server."
- For hosts in the **prod** group; **/etc/motd** should have the following message "This is a prod server."

Hint: Use **group_vars**

Task 4: Configuring SSH Server

Create a playbook **sshd.yml** that runs on all managed hosts and configures the SSH daemon as follows:

- **banner** is set to **/etc/motd**
- **X11Forwarding** is **disabled**
- **MaxAuthTries** is set to **3**

Make sure that the SSH daemon is restarted after any change in its configuration.

Hint: Use a **handler** to restart the SSH daemon.

Task 5: Using Ansible Vault

Create Ansible vault file **secret.yml** with **L!N*X** as the encryption/decryption password.

Then, add the following variable to the vault:

```
user_pass: T*mP1#$
```

Finally, store Ansible vault password in the file **vault__key.txt**.

Hint: Use **ansible-vault create** command.

Task 6: Users and Groups

Create a variables file **users-list.yml** that has the following content:

```
---
users:
  - username: brad
    uid: 774
  - username: david
    uid: 775
  - username: robert
    uid: 776
  - username: jason
    uid: 777
```

Then, create a playbook **users.yml** that runs on all managed nodes and uses the vault file **secret.yml** to achieve the following:

- Create all the users listed in **users-list.yml**.
- Use the variable **user_pass** (from Task #5) to set the users passwords.
- All users should be members of a supplementary group **wheel**.
- Users account passwords should use the **SHA512** hash format.
- **/bin/bash** should be the default login shell for all users.

Hint: Use a **loop** to iterate over the list of **users**.

Task 7: Using Ansible Galaxy Roles

Create a requirements file **requirements.yml** that will download the following three roles from Ansible Galaxy:

1. geerlingguy.docker
2. geerlingguy.kubernetes
3. geerlingguy.rabbitmq

Then, create a playbook **docker.yml** that runs on hosts in the **dev** group and will use the **geerlingguy.docker** role to install **docker**.

Hint: Use **ansible-galaxy install -r** command.

Task 8: Using RHEL System Roles

Create a playbook **selinux.yml** that runs on hosts in the **test** group and does the following:

- Uses the **selinux** RHEL system role.
- Enables the **httpd__can__network__connect** SELinux boolean.

The change must survive system reboot.

Hint: Use **block-rescue** in your playbook.

Task 9: Scheduling Tasks

Create a playbook **sched.yml** that runs on hosts in the **dev** group and does the following:

- A **root** crontab record is created that runs every hour.
- The cron job appends the message "One hour has passed!" to the system log.

Hint: Use **cron** module.

Task 10: Archiving Files

Create a playbook **archive.yml** that runs on hosts in the database **test** group and does the following:

- A directory **/backup** is created.
- A **gzip** archive of all the files in **/var/log** is created and stored in **/backup/logs.tar.gz**

Hint: Use the **file** and **archive** modules.

Task 11: Creating Custom Facts

Create a playbook **facts.yml** that runs on hosts in the **prod** group and does the following:

- A custom Ansible fact **server_role=apache** is created that can be retrieved from **ansible_local.custom.sample_exam** when using Ansible **setup** module.

Hint: Do not forget to copy your **custom.fact** file from your **control** node to the **/etc/ansible/facts.d** directory on the managed node(s).

Task 12: Creating Custom Roles

Create a role called **httpd-role** and store it in **/home/automation/plays/roles**.

The role should satisfy the following requirements:

- Installs the **httpd** package.
- Starts the **httpd** service.
- Enables the **httpd** service to start automatically on boot.
- Allows **http** and **https** traffic through the firewall public zone.
- **/var/www/html/index.html** is generated from the **index.j2** template file with the following content:

```
Welcome to {{ inventory_hostname }}
This is an Apache Web Server.
Please contact {{ sysadmin }} for any questions or concerns.
```

Create a playbook **httpd.yml** that uses the **httpd-role** and runs on hosts in the **test** group. The **sysadmin** variable will show your email and you can set it the playbook.

Task 13: Software Repositories

Create a playbook **repo.yml** that will runs on hosts in the **prod** group and does the following:

1. Create a new repository with the following attributes
   - **file** → zabbix.repo
   - **name** → "zabbix-monitoring"
   - **baseurl** → https://repo.zabbix.com/zabbix/5.2/rhel/8/x86_64/
   - **description** → "Zabbix 5.2 Repo"
   - **enabled** → yes
   - **gpgcheck** → no
2. Install the **zabbix-agent** package.
3. Display the information on the installed **zabbix-agent** package.

Hint: Use the **repository** module.

Task 14: Using Conditionals

Create a playbook **sysctl.yml** that runs on all managed hosts and does the following:

- If a server has more than **2048 MB** of RAM, then the kernel parameter **vm.swappiness** is set to 10.
- If a server has less than **2048 MB** of RAM, then the error message "Available RAM is less than 2048 MB" is displayed.

Hint: Use the **sysctl** module.

Task 15: Installing Software

Create a playbook **packages.yml** that runs on all managed hosts and does the following:

- Installs the **nmap** and **wireshark** packages on hosts in the **dev** group.
- Installs the **tmux** and **tcpdump** packages on hosts in the **test** group.

Hint: Use a **when** condition on the special built-in variable **inventory_hostname**.

Chapter 14: Knowledge Check Solutions

Exercise 1 Solution

First, make sure your system is registered. Then list all the available ansible repositories:

```
[root@control ~]# yum repolist all | grep ansible
ansible-2-for-rhel-8-x86_64-debug-rpms                Red Hat Ansible E disabled
ansible-2-for-rhel-8-x86_64-rpms                      Red Hat Ansible E disabled
ansible-2-for-rhel-8-x86_64-source-rpms               Red Hat Ansible E disabled
ansible-2.8-for-rhel-8-x86_64-debug-rpms              Red Hat Ansible E disabled
ansible-2.8-for-rhel-8-x86_64-rpms                    Red Hat Ansible E disabled
ansible-2.8-for-rhel-8-x86_64-source-rpms             Red Hat Ansible E disabled
ansible-2.9-for-rhel-8-x86_64-debug-rpms              Red Hat Ansible E disabled
ansible-2.9-for-rhel-8-x86_64-rpms                    Red Hat Ansible E disabled
ansible-2.9-for-rhel-8-x86_64-source-rpms             Red Hat Ansible E disabled
```

Choose the repository that provided the latest ansible version and enable it:

```
[root@control ~]# yum-config-manager --enable ansible-2.9-for-rhel-8-x86_64-rpms
Updating Subscription Management repositories.
```

Finally, go ahead and install Ansible:

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

You can also verify your Ansible installation by running:

```
[root@control ~]# ansible --version
ansible 2.9.15
  config file = /etc/ansible/ansible.cfg
  configured module search path = ['/root/.ansible/plugins/modules',
  '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3.6/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 3.6.8 (default, Dec  5 2019, 15:45:45)
  [GCC 8.3.1 20191121 (Red Hat 8.3.1-5)]
```

Exercise 2 Solution

First, create the bash script **adhoc-cmds.sh**:

```
[elliot@control plays]$ cat adhoc-cmds.sh
#!/bin/bash

ansible node4 -m raw -a "apt-get install -y python"
ansible all -m command -a "uptime"
ansible node3 -m shell -a 'echo "Hello, Friend!" > /tmp/hello.txt'
```

Then make the script executable and finally run it:

```
[elliot@control plays]$ chmod u+x adhoc-cmds.sh
[elliot@control plays]$ ./adhoc-cmds.sh
```

Exercise 3 Solution

Solutions may vary but the following is a possible solution:

```
[elliot@control plays]$ cat lab3.yml
---
- name: Lab 3 Solution
  hosts: all
  tasks:
    - name: install nmap
      package:
        name: nmap
        state: present

    - name: create an XZ archive
      archive:
        path: /home
        dest: /tmp/home.tar.bz2
        format: bz2
```

Exercise 4 Solution

Solutions may vary but the following is a possible solution:

```
[elliot@control plays]$ cat lab4.yml
---
- name: Lab 4 Solution
  hosts: all
  tasks:
    - name: Run free command and capture output
      command: free -h
      register: freemem

    - name: Display free memory
      debug:
        msg: "{{ freemem.stdout }}"

    - name: Display IPv4 address
      debug:
        msg: "{{ ansible_facts.default_ipv4.address }}"
```

Exercise 5 Solution

Solutions may vary but the following is a possible solution:

```
[elliot@control plays]$ cat lab5.yml
---
- name: Lab 5 Solution
  hosts: localhost
  tasks:
    - name: Display even numbers 20<= num <=40
      debug:
        msg: "{{ item }}"
      loop: "{{ range(20,41,2) | list }}"
```

Exercise 6 Solution

Solutions may vary but the following is a possible solution:

```
[elliot@control plays]$ cat lab6.yml
---
- name: Lab 6 Solution
  hosts: all
  tasks:
    - name: Install nfs-utils on CentOS servers
      yum:
        name: nfs-utils
      when:  ansible_facts['distribution'] == "CentOS"
```

Exercise 7 Solution

Solutions may vary but the following is a possible solution:

```
[elliot@control plays]$ cat motd.j2
Welcome to {{ inventory_hostname }}.
My IP address is {{ ansible_facts['default_ipv4']['address'] }}.

[elliot@control plays]$ cat lab7.yml
---
- name: Lab 7 Solution
  hosts: all
  tasks:
    - name: Edit /etc/motd using Jinja2
      template:
        src: motd.j2
        dest: /etc/motd
```

Exercise 8 Solution

Solutions may vary but the following is the "quickest" solution:

```
[elliot@control plays]$ echo 7uZAcMBVz > mypass.txt

[elliot@control plays]$ ansible-vault create mysecret.txt
--vault-password-file mypass.txt

[elliot@control plays]$ ansible-vault view mysecret.txt
--vault-password-file mypass.txt
I like pineapple pizza!
```

Exercise 9 Solution

Solutions may vary but the following is a possible solution.

First, install the role **geerlingguy.haproxy**:

```
[elliot@control plays]$ ansible-galaxy install geerlingguy.haproxy -p ./roles
- downloading role 'haproxy', owned by geerlingguy
- downloading role from https://github.com/geerlingguy/ansible-role-haproxy/
archive/1.1.2.tar.gz
- extracting geerlingguy.haproxy to /home/elliot/plays/roles/geerlingguy.haproxy
- geerlingguy.haproxy (1.1.2) was installed successfully
```

Then you can use the following playbook to apply the role:

```
[elliot@control plays]$ cat lab9.yml
---
- name: Lab 9 Solution
  hosts: proxy
  vars:
    haproxy_backend_servers:
     - name: webserver1
       address: node2:80
     - name: webserver2
       address: node3:80
  roles:
    - role: geerlingguy.haproxy
```

Then run the playbook:

```
[elliot@control plays]$ ansible-playbook lab9.yml

PLAY [Lab 9 Solution] **********************

TASK [Gathering Facts] *************************
ok: [node1]

TASK [geerlingguy.haproxy : Ensure HAProxy is installed.] ******
changed: [node1]

TASK [geerlingguy.haproxy : Get HAProxy version.] *****
ok: [node1]

TASK [geerlingguy.haproxy : Set HAProxy version.] ********
ok: [node1]
```

```
TASK [geerlingguy.haproxy : Copy HAProxy configuration in place.] ***
changed: [node1]

TASK [geerlingguy.haproxy : Ensure HAProxy is started and enabled on boot.] *
changed: [node1]

RUNNING HANDLER [geerlingguy.haproxy : restart haproxy] ***********
changed: [node1]

PLAY RECAP ******************
node1: ok=7    changed=4    unreachable=0    failed=0
```

Finally, test to see if load balancing works properly by running the "**curl http://node1**" command twice:

```
[elliot@control plays]$ curl http://node1
Welcome to node3

This is an Apache Web Server.

Please contact the angela@linuxhandbook.com for any questions or concerns.

[elliot@control plays]$ curl http://node1
Welcome to node2

This is an Apache Web Server.

Please contact the angela@linuxhandbook.com for any questions or concerns.
```

Perfect! Load balancing is working properly.

Please Note: **Round Robin** is the default load balancing method! That's why you don't need to explicitly specify it.

Exercise 10 Solution

Solutions may vary but the following is a possible solution:

```
[elliot@control plays]$ cat lab10.yml
---
- name: Lab 10 Solution
  hosts: node3
  vars:
    selinux_fcontexts:
      - { target: '/web(/.*)?', setype: 'httpd_sys_content_t', ftype: 'd' }
    selinux_restore_dirs:
      - /web
  pre_tasks:
    - name: Create /web directory
      file:
        path: /web
        state: directory
  roles:
    - role: rhel-system-roles.selinux
```

After running the playbook; you can run a quick ad-hoc command to check the SELinux file context of **/web**:

```
[elliot@control plays]$ ansible node3 -m command -a "ls -lZd /web"
node3 | CHANGED | rc=0 >>
drwxr-xr-x. 2 root root unconfined_u:object_r:httpd_sys_content_t:s0 6 Nov 26 05:54 /web
```

As you can see; the **/web** directory SELinux file context is set to **httpd_sys_content_t**.

Please Note: For every time you change the SELinux file context of a file; you then have to restore the file's default SELinux security contexts.

Exercise 11 Solution

Solutions may vary but the following is a possible solution:

```
[elliot@control plays]$ cat lab11.yml
---
- name: Lab 11 Solution
  hosts: all
  tasks:
    - name: Create clean-tmp cron job
      cron:
        name: "clean-tmp"
        user: root
        job: rm -rf /tmp/*
        minute: "0"
        hour: "0"
```

After running the playbook; you can check if it is added to **crontab**:

```
[elliot@control plays]$ ansible all -m command -a "crontab -l"
node4 | CHANGED | rc=0 >>
#Ansible: clean-tmp
0 0 * * * rm -rf /tmp/*
node1 | CHANGED | rc=0 >>
#Ansible: clean-tmp
0 0 * * * rm -rf /tmp/*
node2 | CHANGED | rc=0 >>
#Ansible: clean-tmp
0 0 * * * rm -rf /tmp/*
node3 | CHANGED | rc=0 >>
#Ansible: clean-tmp
0 0 * * * rm -rf /tmp/*
```

Exercise 12 Solution

Solutions may vary but the following is a possible solution.

First, make sure you set the **log_path** configuration setting:

```
[elliot@control plays]$ grep -i log ansible.cfg
log_path = lab12.out
```

Then run the playbook in **check mode**:

```
[elliot@control plays]$ ansible-playbook --check lab12.yml
ERROR! We were unable to read either as JSON nor YAML, these are the errors we
got from each: JSON: Expecting value: line 1 column 1 (char 0)

Syntax Error while loading YAML.
  found unacceptable key (unhashable type: 'AnsibleMapping')

The error appears to be in '/home/elliot/plays/lab12.yml': line 10, column 14, but
may be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

        name: "{{ item }}"
      loop: {{ grps }}
            ^ here
We could be wrong, but this one looks like it might be an issue with
missing quotes. Always quote template expression brackets when they
start a value. For instance:

    with_items:
      - {{ foo }}

Should be written as:

    with_items:
      - "{{ foo }}"
```

Finally, fix the error and run the playbook again:

```
[elliot@control plays]$ cat lab12.yml
---
- name: Fix the error
  hosts: localhost
  vars:
    grps: ["devops","secops","netops"]
```

```
  tasks:
    - name: create groups
      group:
        name: "{{ item }}"
      loop: "{{ grps }}"

[elliot@control plays]$ ansible-playbook lab12.yml

PLAY [Fix the error] ***********************************************************

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

TASK [create groups] ***********************************************************
changed: [localhost] => (item=devops)
changed: [localhost] => (item=secops)
changed: [localhost] => (item=netops)

PLAY RECAP *********************************************************************
localhost: ok=2    changed=1    unreachable=0    failed=0
```

Finally, you can verify the groups are created:

```
[elliot@control plays]$ tail -3 /etc/group
devops:x:1005:
secops:x:1006:
netops:x:1007:
```
