The Ansible Commands

Before moving on to writing and executing more advanced playbooks, we will look at the rest of the built-in Ansible commands. Here, we will cover using the commands that make up Ansible. Toward the end of this chapter, we will install a third-party tool to visualize our host inventory.

This chapter will cover the following topics:

* Inbuilt commands
* Third-party commands

Inbuilt commands

When we installed Ansible, several different commands were installed. These were as follows:

* ansible
* ansible-config
* ansible-console
* ansible-doc
* ansible-galaxy
* ansible-inventory
* ansible-playbook
* ansible-pull
* ansible-vault

We already covered the ansible-galaxy command in [*Chapter 2*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/2), *Exploring Ansible Galaxy*. We will be looking at ansible-playbook throughout the remaining chapters of this book, so I will not go into any detail about that command in this chapter. Let’s start at the top of the list and a command we have already used.

Ansible

Now, you would have thought that ansible would be the most common command we will use throughout this book, but it isn’t.

The ansible command is only ever used for executing ad hoc commands against a single host or collection of hosts. In [*Chapter 1*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/1), *Installing and Running Ansible*, we created a host inventory file that targeted a single local virtual machine.

For this part of the chapter, we’ll look at targeting four different hosts I have running in a cloud provider; my host’s file looks as follows:

ansible01 ansible\_host=139.162.233.174

ansible02 ansible\_host=139.162.233.227

ansible03 ansible\_host=139.144.132.49

ansible04 ansible\_host=139.144.132.71

[london]

ansible01

ansible02

[nyc]

ansible03

ansible04

[demohosts:children]

london

nyc

[demohosts:vars]

ansible\_connection=ssh

ansible\_user=root

ansible\_private\_key\_file=~/.ssh/id\_rsa

host\_key\_checking=False

CopyExplain

As you can see, I have four hosts – ansible01 > ansible04. My first two hosts are in a group called london and my second two are in a group called nyc. I have then taken these two groups and created one containing them called demohosts, and I used this group to apply some basic configurations based on the hosts I have launched.

Using the ping module, I can check connectivity to the hosts by running the following commands. First, let’s check the two hosts in london:

$ ansible -I hosts london -m ping

CopyExplain

This returns the following results:

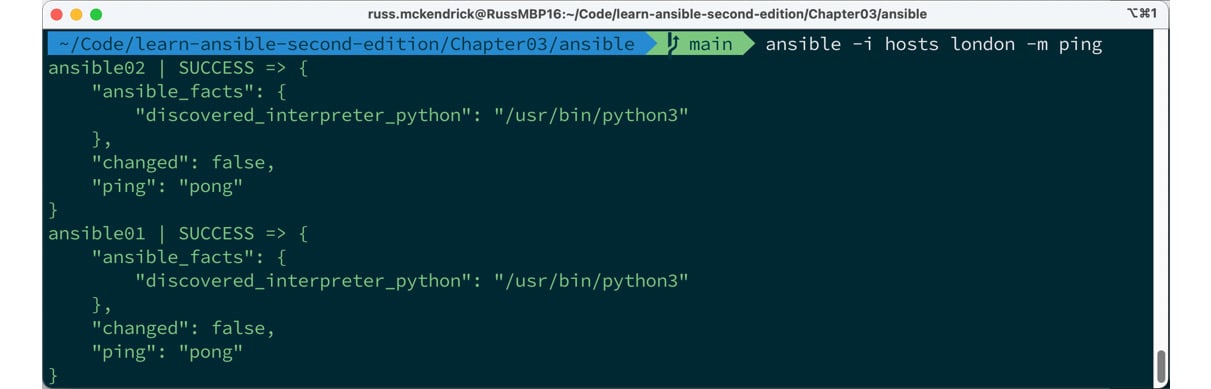


Figure 3.1 – Doing an Ansible ping targeting the london hosts

Now, let’s run the same command, but this time targeting the nyc hosts:

$ ansible -i hosts nyc -m ping

CopyExplain

This gives us the following output:

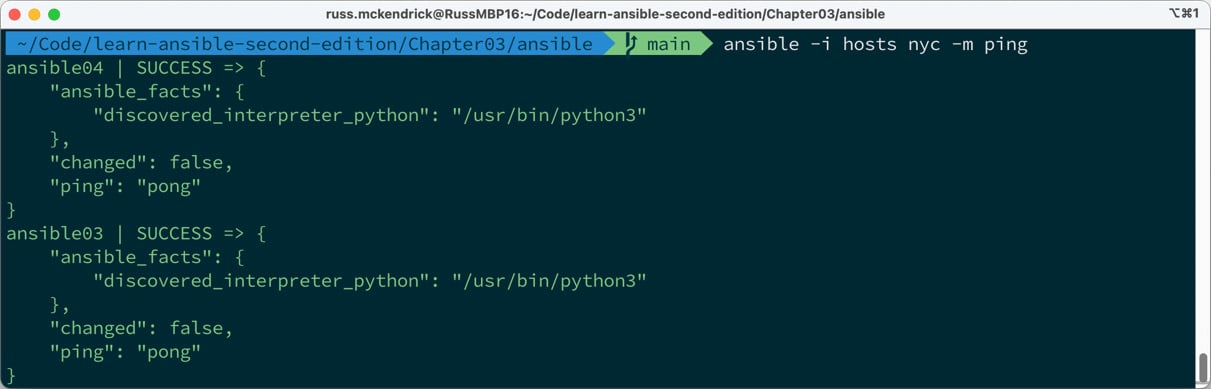


Figure 3.2 – Doing an Ansible ping targeting the nyc hosts

As you can see, all four of my hosts returned pong.

I can also target all four hosts at once by adding all rather than a particular group of hosts:

$ ansible -i hosts all -m ping

CopyExplain

Now that we can access our host through Ansible, we can target them and run some ad hoc commands; let’s start with something basic:

$ ansible -i hosts london -a "ping -c 3 google.com"

CopyExplain

This command will connect to the london hosts and run the ping -c 3 google.com command; this will ping the google.com domain from the hosts and return the results:

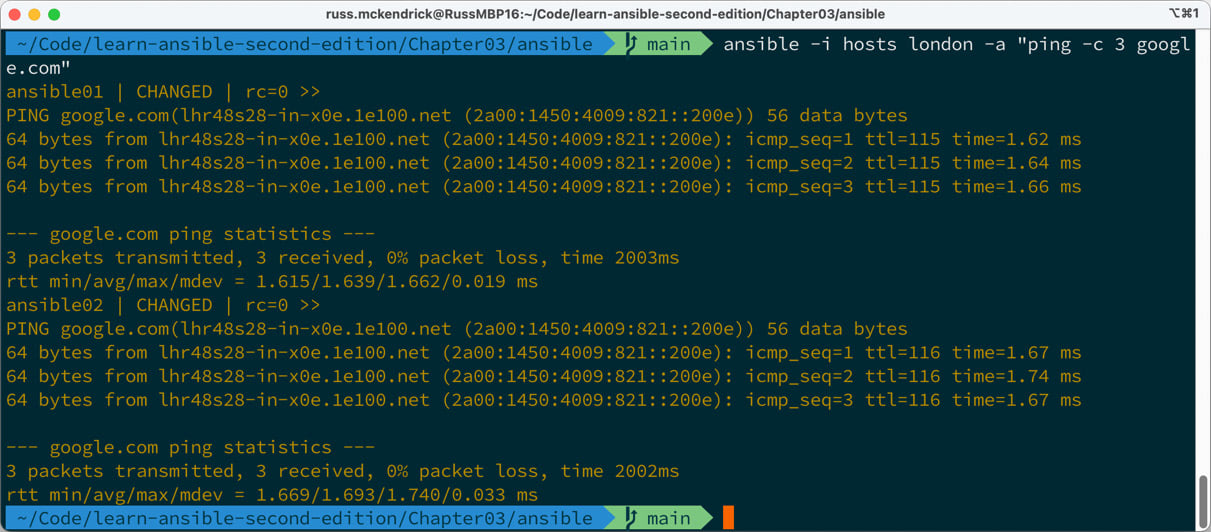


Figure 3.3 – Running the ping command against google.com

We can also run a single module using the ansible command; we did this in [*Chapter 1*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/1), *Installing and Running Ansible*, using the setup module. However, a better example would be updating all the installed packages across all the hosts by running the following command:

$ ansible -i hosts all -m ansible.builtin.apt -a "name=\* state=latest update\_cache=ye"

CopyExplain

As you can see, we have taken the ansible.builtin.apt module, which we defined as follows in [*Chapter 1*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/1), *Installing and Running Ansible*:

- ansible.builtin.apt:

    name:"\*"

    state:"latest"

    update\_cache:"true"

CopyExplain

I’ve passed in the same options, but rather than use YAML, I have formatted it as a key and value, which is typical of what you would pass into any command on the command line:

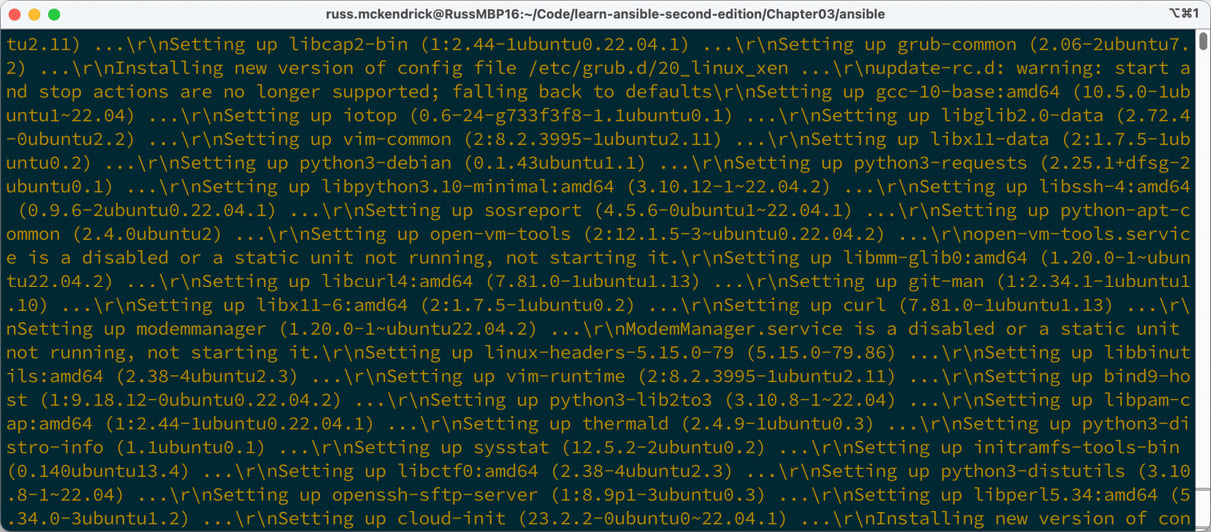


Figure 3.4 – Using the ansible.builtin.apt module to update all the packages

As you can see, the output when running Ansible is quite verbose, and it provides feedback to tell us precisely what it did during the ad hoc execution.

Let’s rerun the command against all our hosts, but this time just for a single package, say ntp:

$ ansible -i hosts all -m ansible.builtin.apt -a "pkg=ntp state=latest"

CopyExplain

Running the command once will install the package on all four of our hosts:

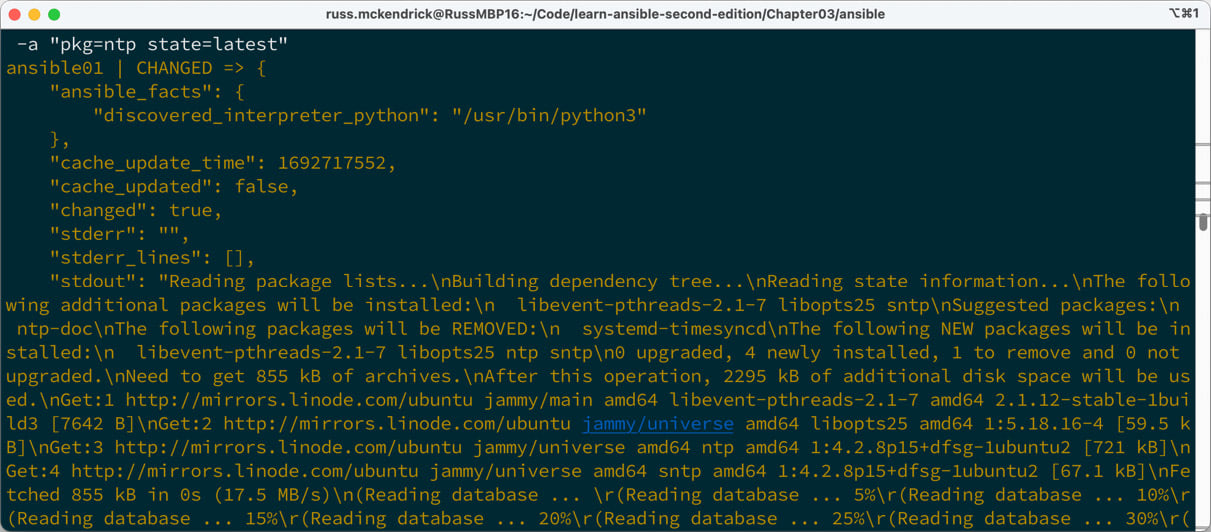


Figure 3.5 – Using the ansible.builtin.apt module to install the ntp package

Now, let’s rerun the command:

$ ansible -i hosts all -m ansible.builtin.apt -a "pkg=ntp state=latest"

CopyExplain

Running the command once will install the package on all four of our hosts and give us the following results:

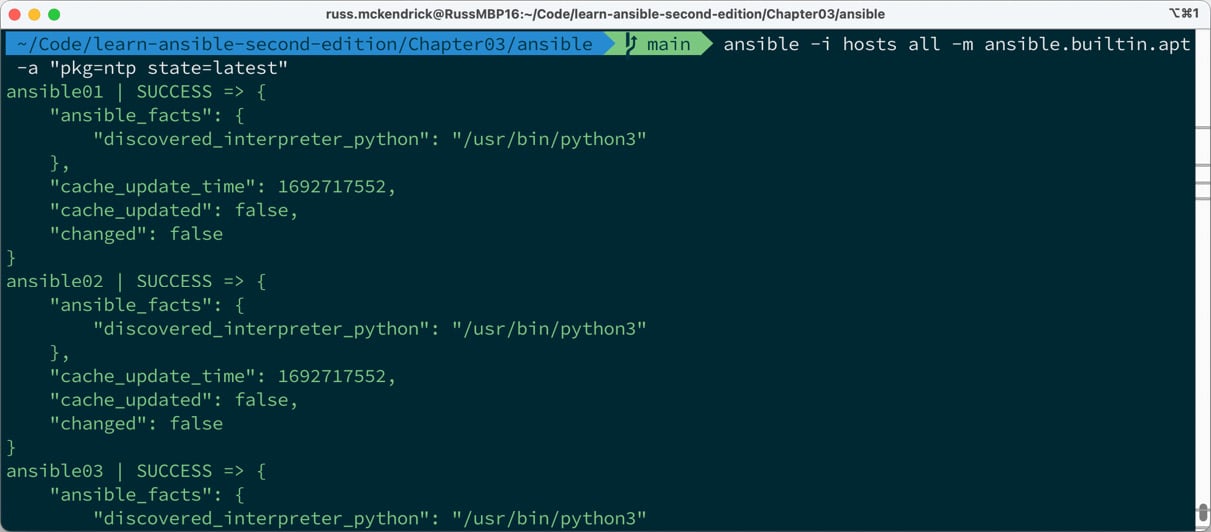


Figure 3.6 – Rerunning the ansible.builtin.apt module to install the ntp package

As you can see, the hosts are returning a SUCCESS status and are showing no changes, which is what we would expect to see.

So, why would you want to do this, and what is the difference between the two commands we ran?

First, let’s take a look at two of the commands we initially ran once we confirmed our hosts were available using an Ansible ping:

$ ansible -i hosts london -a "ping -c 3 google.com"

$ ansible -i hosts all -m ansible.builtin.apt -a "name=\* state=latest update\_cache=true"

CopyExplain

While it appears that the first command isn’t running a module, it is. The default module for the ansible command is called raw and runs raw commands on each of the targeted hosts. The -a part of the command passes arguments to the module. The raw module happens to accept raw commands, which is precisely what we are doing with the second command.

As mentioned previously, you will have noticed that the syntax is slightly different when we pass commands to the ansible command and when using it as part of a YAML playbook. All we are doing here is passing the key-value pairs directly to the module.

So, why would you want to use Ansible like this? Well, it’s excellent for running commands directly against non-Ansible managed hosts in an extremely controlled way.

Ansible uses SSH to connect to the hosts, runs the command, and lets you know the results. Just be careful – it is easy to get overconfident and run something like the following:

$ ansible -I hosts all -a "reboot now"

CopyExplain

If the user Ansible is using to connect to the host has permission to execute the command, it will just run the command you give it. Running the previous command will reboot all the servers in the host inventory file:

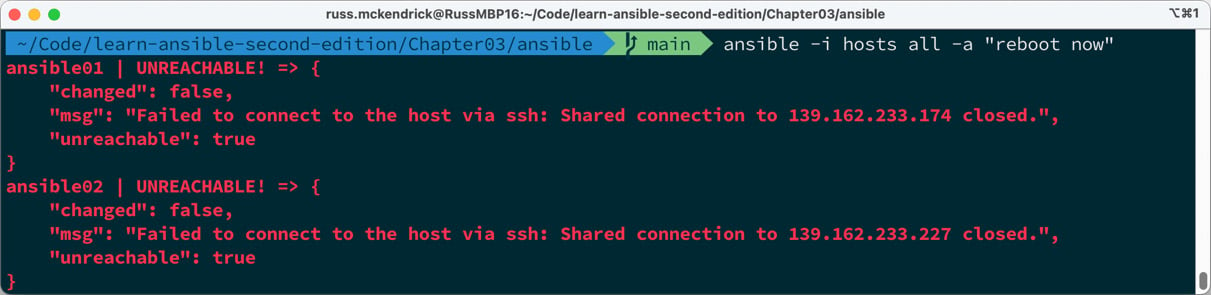


Figure 3.7 – Rebooting all four of the hosts with a single command

All hosts have an UNREACHABLE status because the reboot command kicked our SSH session before the SUCCESS status could be returned. You can, however, see that each of the hosts has been rebooted by running the uptime command:

$ ansible -i hosts all -a "uptime"

CopyExplain

The following screenshot shows the output for the preceding command:



Figure 3.8 – Checking the uptime of the four hosts

Important

As mentioned previously, plus speaking from experience (it’s a long story), please be extremely careful when using Ansible to manage hosts using ad hoc commands – it’s a powerful but dumb tool, and it will assume you know the consequences of running the commands against your hosts.

That concludes our look at the ansible command; let’s move on to our next command, ansible-config.

The ansible-config command

The ansible-config command is used to manage Ansible configuration files. Ansible ships with sensible defaults, so there is little to configure outside of these. You can view the current configuration by running the following:

$ ansible-config dump

CopyExplain

As shown from the following output, all the text in green is the default config, and any configuration in orange is a changed value:

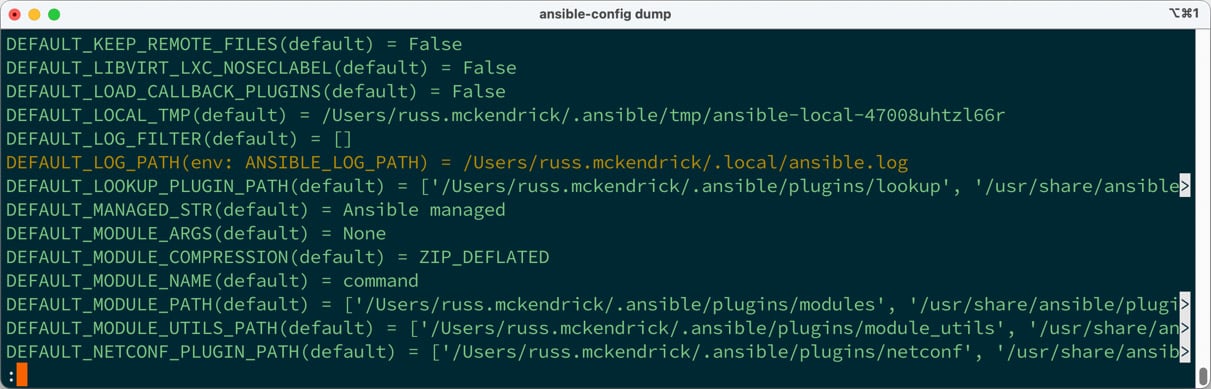


Figure 3.9 – Dumping our complete Ansible configuration to screen

Running the following command will list details of every configuration option there is within Ansible, including what the option does, its current state, when it was introduced, the type, and much more:

$ ansible-config list

CopyExplain

The following screenshot shows the output for the preceding command:

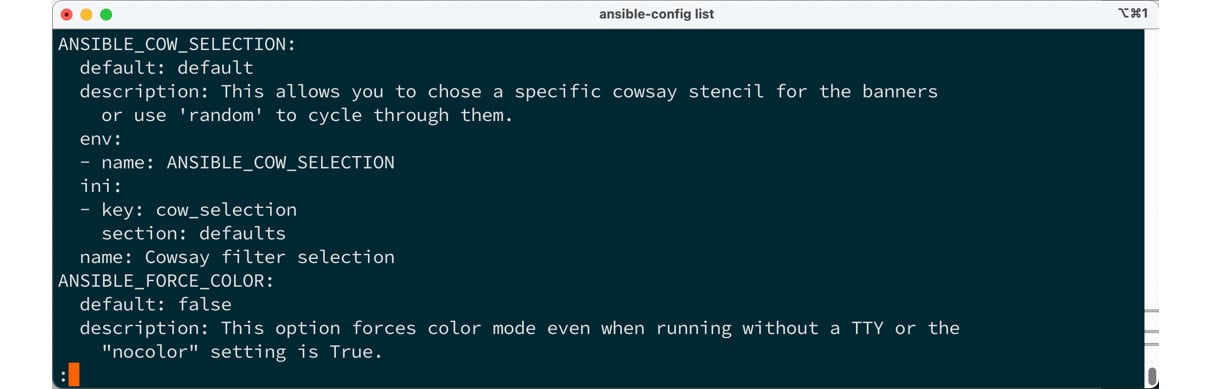


Figure 3.10 – Viewing details on an Ansible configuration option

If you had a configuration file, say at ~/.ansible.cfg, then you can load it using the -c or–-config flags:

$ ansible-config view –-config "~/.ansible.cfg

CopyExplain

The previous command will give you an overview of the custom configuration file and display the Ansible default values not defined in your custom configuration file.

The ansible-console command

Ansible has a built-in console. It is not something I have used much in my day-to-day running of Ansible. To start the console, we need to run one of the following commands:

$ ansible-console -i hosts

$ ansible-console -i hosts london

$ ansible-console -i hosts nyc

CopyExplain

The first of the three commands targets all of the hosts, while the next two just target the named groups:

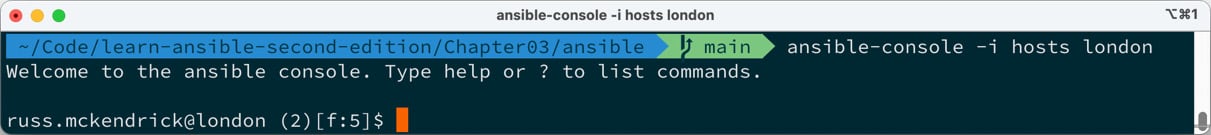


Figure 3.11 – Establishing the console connection

Once connected, you will see that I am connected to the london group of hosts, in which there are two hosts. From here, you can type a module name, such as ping:

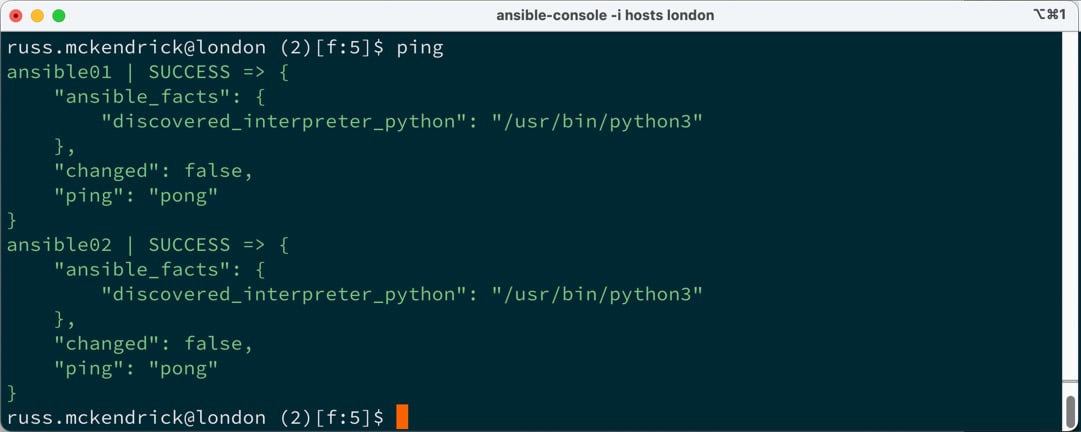


Figure 3.12 – Running ping from Ansible

Alternatively, you can use the raw module; for example, you can check the uptime command by typing ansible.builtin.raw uptime:



Figure 3.13 – Using the raw module to run the uptime command

You can also use the same syntax as we did when running the ansible command to pass key-value pairs – for example, running the following at the console prompt:

ansible.builtin.apt pkg=ntp state=latest update\_cache=true

CopyExplain

It should give you something like the following output:

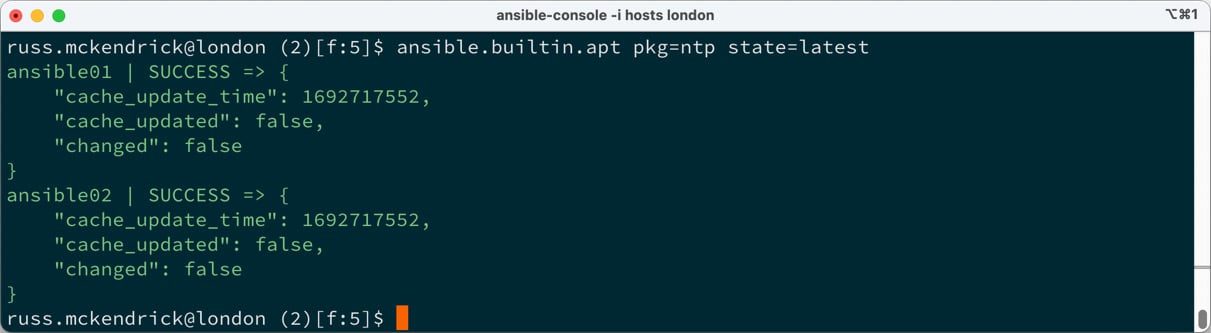


Figure 3.14 – Checking that the ntp package is installed using the ansible.builtin.apt module

You may have noticed that the syntax of the command we are running this time is slightly different from when we ran the same module using the ansible command earlier in this chapter.

That command was as follows:

$ ansible -i hosts london -m ansible.builtin.apt -a"pkg=ntp state=latest update\_cache=true"

CopyExplain

Whereas this time, we just ran the following:

ansible.builtin.apt pkg=ntp state=latest update\_cache=true

CopyExplain

The reason for this is that when we called the module using the ansible command, we were working on the command line of our local machine, so we needed to pass in the module name using the -m flag and then define the attributes by using the -a flag. After, we had to pass in our key-value pairs within quotation marks so as not to break the flow of the command as spaces are used as a delimiter when it comes to the command line.

When we ran the Ansible console, we had effectively already run the ansible -i hosts london part of the command, left our local command line altogether, and were interacting with Ansible itself directly.

To leave the console, type exit to return to your regular command-line shell.

As mentioned at the start of this section, the ansible-console command is something I do not use – mainly for the warning I gave when we looked at the ansible command at the start of this chapter.

When connecting to several hosts using the ansible-console command, you must be 100% confident that the commands you are typing are correct. For example, while I was only connected to two hosts, my hosts file could have contained 200 hosts. Now, imagine I typed the wrong command – executing it across 200 hosts at once could potentially do some unwanted things, such as rebooting them all simultaneously.

To quit the ansible-console session, simply type exit and hit *Enter*.

As you have probably guessed, this happened to me. It wasn’t 200 hosts, but it could have easily been – *so please be careful.*

The ansible-inventory command

Using the ansible-inventory command provides you with details of your host inventory files. It can be helpful to understand how your hosts are grouped. For example, let’s say I run the following command:

$ ansible-inventory -i hosts–-graph

CopyExplain

In the same folder as the hosts inventory file that I have been using throughout this section, the following is returned:

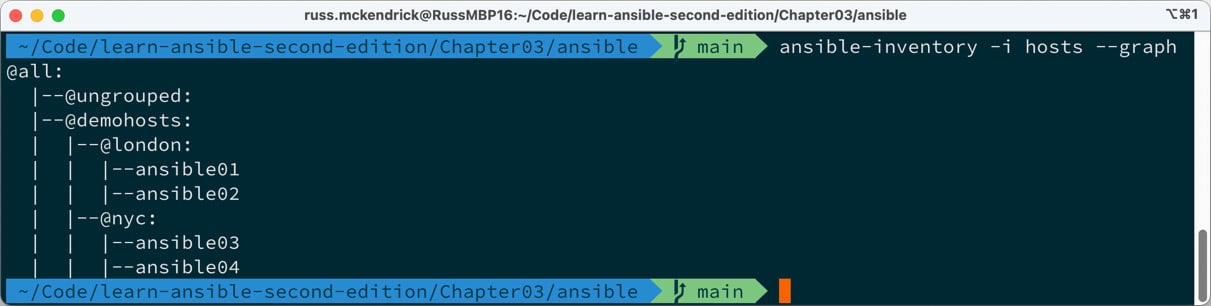


Figure 3.15 – Getting an overview of the inventory hosts file

As you can see, it displays the groups, starting with all, then the main host group (demohosts), then the child groups (london and nyc), and finally the hosts themselves (ansible01 > ansible04).

If you want to view the configuration for a single host, you can use this command:

$ ansible-inventory -i hosts –-host=ansible01

CopyExplain

The following screenshot shows the output of the preceding command:

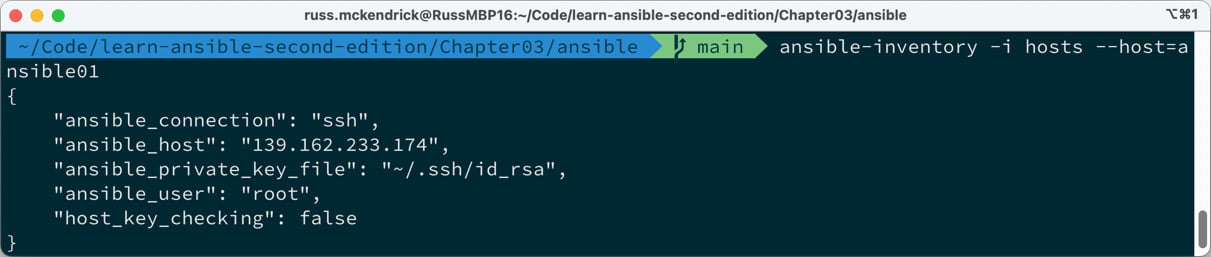


Figure 3.16 – Viewing a single host

You may have noticed that it displays the configuration information that the host inherited from the configuration we set for all the hosts in the demohost host group in the inventory file. You can view all the information on each of your hosts and groups by running the following command:

$ ansible-inventory -i hosts –-list

CopyExplain

This command is helpful if you have a large or complicated host inventory file and want information on just a single host or if you have taken on a host inventory and want a better idea of how the inventory is structured. We will look at a third-party tool later in this chapter that gives more display options.

What is ansible-pull?

Like the ansible-console command, ansible-pull is not a command I use very often; I can count on one hand how often I have used it in the past several years.

ansible-pull is a command that allows a target machine to pull its configuration from a given source, such as a Git repository, and apply it locally. This reverses the typical Ansible push model, where a central control node pushes configuration to managed nodes.

The ansible-pull command works as follows:

1. The target machine, the one running ansible-pull, fetches a specified repository.
2. Once the repository has been fetched, the target machine looks for a playbook. By default, it looks for one called localhost.yml, but you can specify a different playbook file if you need to – please note that this is not included in the example files.
3. The target machine then runs the playbook against itself.

There are a few use cases for ansible-pull:

* **Decentralized configuration management**: In environments where a centralized Ansible server might be a single point of failure or isn’t feasible, ansible-pull allows nodes to self-configure by pulling their configurations
* **Edge locations and remote sites**: For edge locations or remote sites with limited connectivity, ansible-pull can be scheduled to run at specific intervals via a cron job, ensuring that hosts can self-update when they have connectivity
* **Development and testing**: Developers can use ansible-pull to pull down and apply configurations to their local development environments, ensuring consistency with production configurations

There are a few prerequisites to running ansible-pull – the most prominent being that the host running ansible-pull must have an active and valid Ansible installation and any other dependencies needed to execute the playbook.

In summary, ansible-pull provides a way to invert the traditional Ansible model, allowing hosts to pull their configurations as needed rather than having a central host push the configurations to them, as we did in [*Chapter 1*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/1), *Installing and Running Ansible*, and [*Chapter 2*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/2), *Exploring Ansible Galaxy*. For the remainder of this book, we will be taking the more traditional approach to Ansible deployments and pushing our configuration to our target hosts.

However, it is always good to know that if, for whatever reason, you are not able to take this approach, then you do have an alternative option in ansible-pull.

Using the ansible-vault command

In Ansible, it is possible to load variables from files or within a playbook itself; we will look at this in the next chapter in more detail. These files can contain sensitive information such as passwords and API keys. Here’s an example:

secret:"mypassword"

secret-api-key:"myprivateapikey"

CopyExplain

As you can see, we have two sensitive bits of information visible as plaintext. This is OK while the file is on our local machine – well, just about OK. But what if we want to check the file into source control to share it with our colleagues?

*We shouldn’t store this information in plaintext, even if the repository is private.*

Ansible introduced Ansible Vault to help solve this very problem. Using the ansible-vault command, we can encrypt a file or just variables, and then when Ansible is executed, it can be decrypted in memory, and the content can be read as part of the execution.

Note

For the rest of the chapter, I will set a Vault password of password, should you wish to run the ansible-vault command against the files in the Chapter03/vault folder.

To encrypt a file, we need to run the following command, providing a password that will be used to decrypt the file when prompted:

$ ansible-vault encrypt secrets.yml

CopyExplain

The following screenshot shows the output of the preceding command:



Figure 3.17 – Using ansible-vault to encrypt an entire file

As you can see from the output, you will be asked to confirm the password. Once encrypted, your file will look something like this:

$ANSIBLE\_VAULT;1.1;AES256

62373138643038636664363166646637333131386431366137643630326433303231336331303262

3661383061313436653464663039626338376233646630310a306437666462313439636634646633

39653435333433326361306531393832613038353665333866383161313239343134376632316263

3736633665303161630a393265633066663631336239613938363130306262613633333030336430

66343833376532313866363838653464383065633737613735323739303232383031326262376366

61663136623431306363666330373831336230323132336263626237366539326162373564353937

303465306233633633303533633232623233

CopyExplain

As you can see, the details are encoded using text. This ensures that our secrets.yml file will still work without problems when it’s checked into source control such as Git.

You can view the content of a file by running the following command:

$ ansible-vault view secrets.yml

CopyExplain

This will ask you for the password and print the content of the file to the screen:



Figure 3.18 – Using ansible-vault to encrypt an entire file

You can decrypt the file on disk by running the following command:

$ ansible-vault decrypt secrets.yml

CopyExplain

This will restore the file to its unencrypted original state.

Important

When using the ansible-vault decrypt command, please do not commit or check the decrypted file into your source control system!

Since early in the release of Ansible 2, encrypting a single variable in a file is now possible. Let’s add some more variables to our file:

username:"russmckendrick"

password:"mypassword"

secretapikey:"myprivateapikey"

packages:

  - apache2

  - ntp

  - git

CopyExplain

It would be good if we didn’t have to keep viewing or decrypting our file to check its variable name and overall content.

Let’s encrypt the password content by running the following command:

$ ansible-vault encrypt\_string'mypassword'–-name'password'

CopyExplain

This will encrypt the mypassword string and give it a variable name of password:

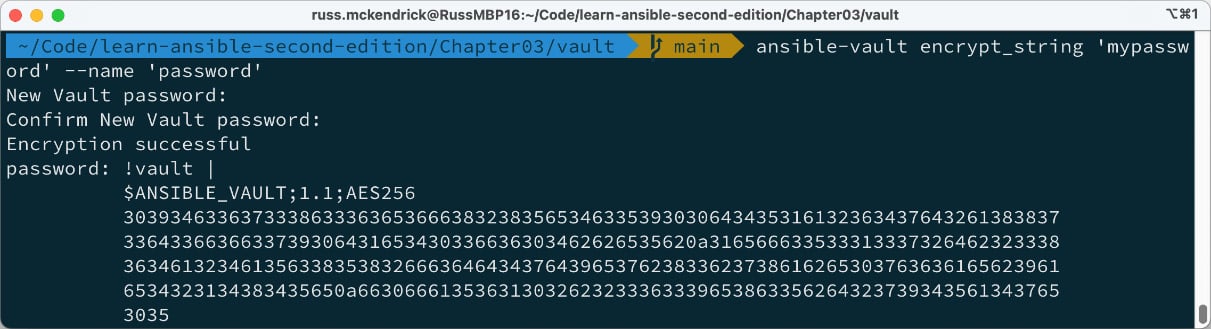


Figure 3.19 – Using ansible-vault to encrypt a single string

We can then copy and paste the output into our file and repeat this process for secretapikey:

$ ansible-vault encrypt\_string 'myprivateapikey' –-name 'secretapikey'

CopyExplain

With that, we have generated two secret variables and replaced the unencrypted ones in our variables file.

Note

For ease of reading, I have truncated the output a little – the entire file can be found in the Chapter03/vault folder in this book’s GitHub repository.

Our variables file should end up looking something like this:

username:"russmckendrick"

password: !vault |

          $ANSIBLE\_VAULT;1.1;AES256

30393463363733386333636536663832383565346335393030643435316132363437643261383837

          3035

secretapikey: !vault |

          $ANSIBLE\_VAULT;1.1;AES256

38663133393834646638663632353634343638626237333438336131653862373761666539326263

          3934

packages:

  - apache2

  - ntp

  - git

CopyExplain

As you can see, that is much easier to read and is just as secure as encrypting the file.

So far, so good, but how do you use Ansible Vault encrypted data in an Ansible playbook?

Before we look at how to do this, let’s see what happens when you don’t tell the ansible-playbook command you are using Ansible Vault by running the following playbook. As you can see, it is loading in the myvars.yml file and then printing the contents of our variables to the screen using the ansible.builtin.debug module:

---

- name: "Print some secrets"

  hosts: "localhost"

  vars\_files:

    - "myvars.yml"

  tasks:

    - name: "Print the vault content"

      ansible.builtin.debug:

        msg:

          - "The username is {{ username }} and password is {{ password }}, also the API key is {{ secretapikey }}"

          - "I am going to install {{ packages }}"

CopyExplain

We can run the playbook using the following command; note that since it’s just running locally, we are not passing an inventory file. This is something it will give you a warning about:

$ ansible-playbook playbook01.yml

CopyExplain

This results in an error message being shown in the Terminal output:

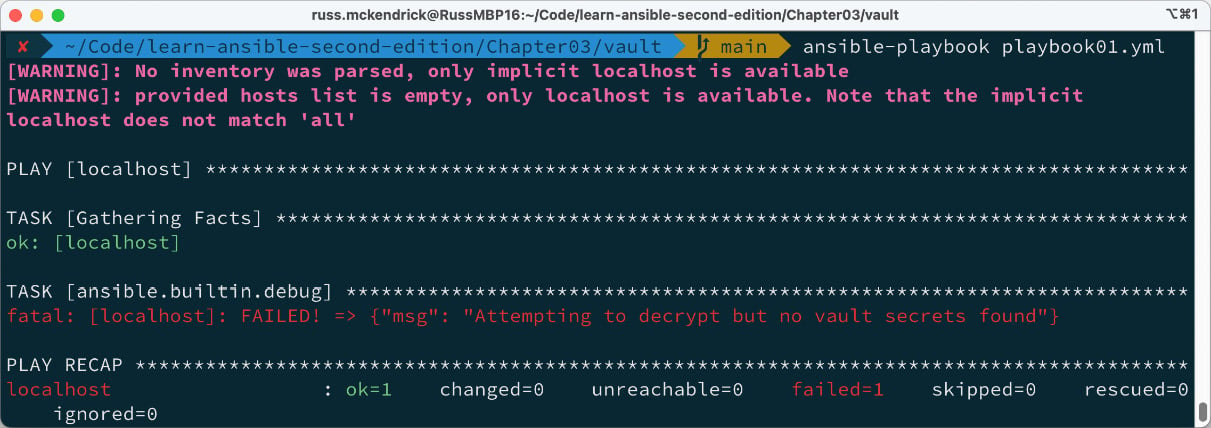


Figure 3.20 – Getting an error when running the ansible-playbook command

As you can see, it’s complaining that it found Vault-encrypted data in one of the files, but we haven’t provided the secret to unlock it.

The first way we can pass the Vault password during the ansible-playbook run is to put the password in a text file and have the ansible-playbook command read the file’s contents.

As mentioned at the start of this section, I have been encoding my Vaults using a password of password. Let’s put that in a file and then use it to unlock our Vault:

$ echo "password" > /tmp/vault-file

CopyExplain

Running the following command will read the content of /tmp/vault-file and decrypt the data:

$ ansible-playbook --vault-id /tmp/vault-file playbook01.yml

CopyExplain

As you can see from the following playbook run, the output is now as we expect:

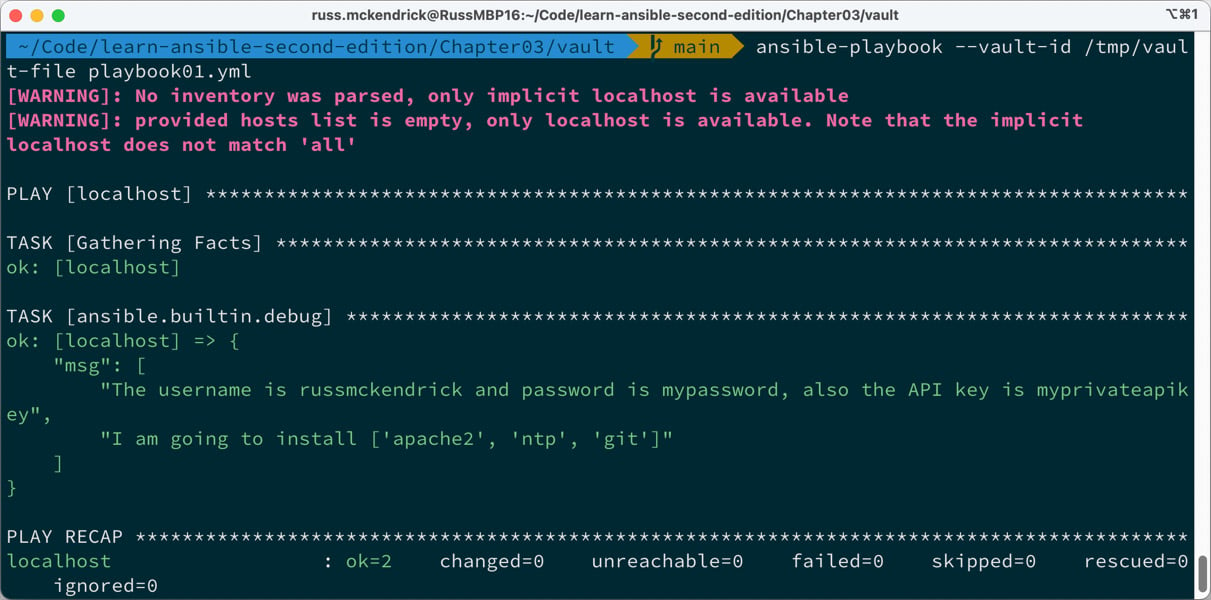


Figure 3.21 – Running ansible-playbook and passing the Vault password in via a file

If you prefer to be prompted for the password, you can use the following command:

$ ansible-playbook --vault-id @prompt playbook01.yml

CopyExplain

The following output shows the prompt:



Figure 3.22 – Running ansible-playbook and entering the password via a prompt

You might be asking yourself, why are there two different options? When prompted, just running the command and entering the password might seem enough.

However, when it comes to using services such as the ones we will cover in [*Chapter 15*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/15), *Using Ansible with GitHub Actions and Azure DevOps*, the commands need to run utterly unattended as there will not be an active terminal for you to enter the password.

Another advantage that will be looked at in both [*Chapter 15*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/15), *Using Ansible with GitHub Actions and Azure DevOps*, and [*Chapter 16*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/16), *Introducing Ansible AWX and Red Hat Ansible Automation Platform*, is that by abstracting away the need for an end user to enter credentials at runtime, it is entirely possible for someone to run a pipeline and never need to know or have access to any of the secrets stored in your playbooks or the credentials to unlock them.

Third-party commands

Before we finish looking at the various Ansible commands, let’s look at a command that isn’t shipped as part of Ansible itself but is, in fact, a third-party open source project.

The ansible-inventory-grapher command

The ansible-inventory-grapher command, by Will Thames, uses the Graphviz library to visualize your host inventories. The first thing we need to do is install Graphviz. To install this on macOS using Homebrew, run the following command:

$ brew install graphviz

CopyExplain

To install Graphviz on Ubuntu, use the following command:

$ sudo apt-get install graphviz

CopyExplain

Once installed, you can install ansible-inventory-grapher using pip:

$ pip install ansible-inventory-grapher

CopyExplain

Now that we have everything installed, we can generate the graph using the hosts file we used earlier in this chapter:

$ ansible-inventory-grapher -i hosts demohosts

CopyExplain

This will generate something that looks like this:

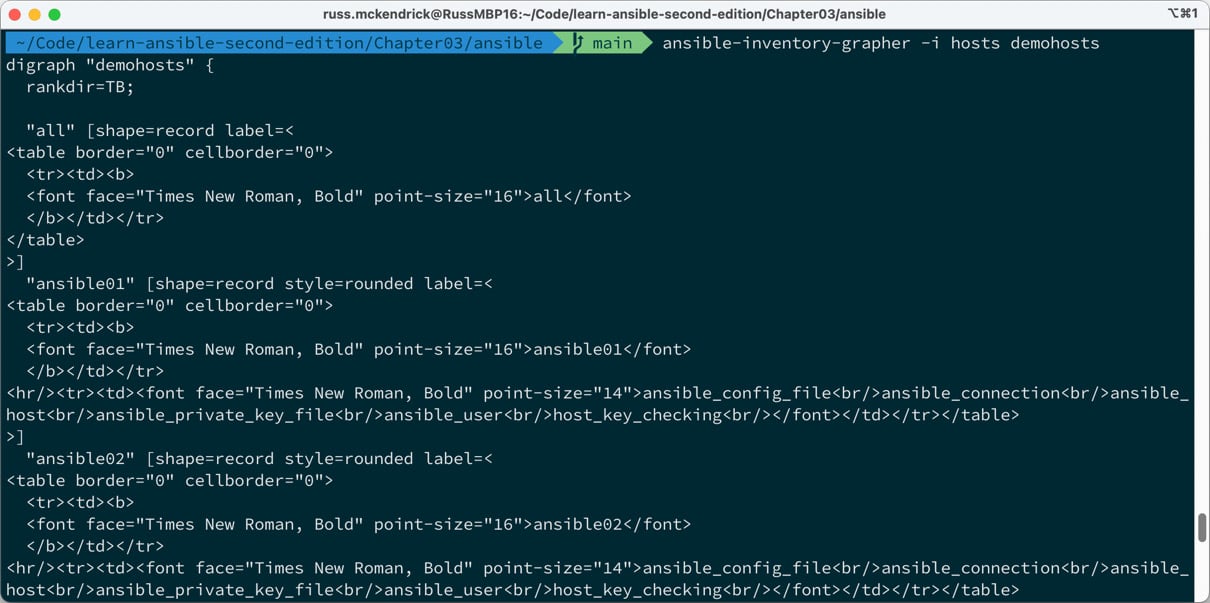


Figure 3.23 – Running ansible-inventory-grapher against our hosts file

This is the raw output of the graph. As you can see, it is like and uses some of the same syntax as HTML. We can render this using the dot command, which ships as part of Graphviz. The dot command creates hierarchical drawings from graphs. To do this, run the following command:

$ ansible-inventory-grapher -i hosts demohosts | dot -Tpng > hosts.png

CopyExplain

This will generate a PNG file called hosts.png that contains the visualization of the host inventory file you can see here:

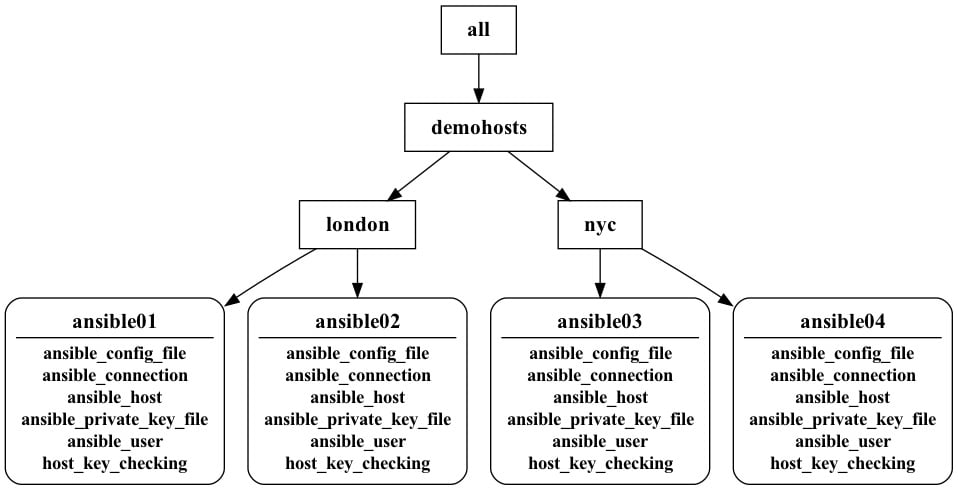


Figure 3.24 – The output of passing our ansible-inventory-grapher output through Graphviz

As you can see, this is an excellent representation of the hosts being targeted by Ansible; it works great for inclusion in your documentation but also gives you an idea of how a complicated inventory file is structured.

Summary

In this chapter, we briefly looked at some of the supporting tools that ship as part of a standard Ansible installation and a useful third-party tool designed to work with Ansible.

We will use these commands and the one we have purposely missed, ansible-playbook, in later chapters.

In the next chapter, we will write a more complex playbook that installs a basic LAMP stack on our local virtual machine.