Installing and Running Ansible

Welcome to this, our first chapter in the second edition of *Learn Ansible*. In this chapter, we will look at a few topics to introduce you to **Ansible**; these topics will familiarize you with the basics of what Ansible is and give you a few different use cases.

By the end of the chapter, you will have gotten hands-on with Ansible and covered the following:

* Who is behind Ansible?
* The differences between Ansible and other tools
* The problem Ansible solves
* How to install Ansible on macOS and Linux
* Running Ansible on Windows 11 using the Windows Subsystem for Linux
* Launching a test virtual machine
* An introduction to playbooks

Before we start talking about Ansible, let’s quickly discuss my background, how I came to be writing a book about Ansible, and what you will need to install and run Ansible on your system.

Technical requirements

Later in this chapter, we will install Ansible, so you will need a machine capable of running it. I will go into more detail about these requirements in the second half of the chapter. We will also use **Multipass** to launch a virtual machine locally. A section walks through installing Multipass and downloading an Ubuntu image to use the virtual machine’s base, which is a download of a few hundred MBs. You can find all of the code used in this chapter at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter01.

My story: part one

I have been working with servers, primarily ones that serve web pages, since the late 90s, and the landscape is unrecognizable. Here is a quick overview of my first few years running servers to give you an idea of how I used to operate my early servers.

Like most people at the time, I started with a shared hosting account where I had very little control over anything on the server side when the website I was running outgrew shared hosting due to the forum, which made up part of the site’s popularity. I moved to a dedicated server, where I thought I could flex my future system administrator muscles, but I was wrong.

The server I got was a Cobalt RaQ 3; this was a 1U server appliance that was ahead of its time. However, I did not have root-level access to the machine, and I had to use the web-based control panel for everything I needed to do. Eventually, I got a level of access where I could access the server using Telnet; I now know this isn’t good, but it was the early days, and SSH was considered cutting-edge. I started to teach myself how to be a system administrator by making changes in the web control panel and looking at the changes to the configuration files on the server.

After a while, I changed servers and, this time, opted to forego any web-based control panel and use what I had learned with the Cobalt RaQ to configure my first proper **Linux**, **Apache**, **MySQL**, **PHP** (or **LAMP** for short) server by using the pages of notes I had made. I had created runbooks of one-liners to install and configure the software I needed and numerous scribbles to help me investigate problems and keep the lights on.

After I got my second server for another project, I realized that was probably a good time to type out my notes so that I could copy and paste them when I needed to deploy a server; the timing of this couldn’t have been better as it was a few months after making these notes that my first server failed—my host apologized and replaced it with a higher-specification but completely fresh machine with an updated operating system.

So, I grabbed my Microsoft Word file containing my notes and copied and pasted each instruction, making tweaks based on what I needed to install on the upgraded operating system. Several hours later, I had my server up and running and my data restored.

One of the critical lessons I learned, other than that there is no such thing as too many backups, was not to use Microsoft Word to store these types of notes; the Linux command line doesn’t care if your notes are all nicely formatted with headings and courier font for the bits you need to paste in. It does care about proper syntax, and Word had very kindly autocorrected and formatted all of my notes for print, meaning that not only did I have the pressure of having to deploy a new server and restore the backups I had thankfully been taking each day but also to try and debug my notes as I was doing so.

Because of this, I made a copy of the history file on the server and transcribed my notes in plaintext. These notes provided the base for the next few years as I started to script parts of them, mainly the bits that didn’t require any user input.

These scraps of commands, one-liners, and scripts were all adapted through Red Hat Linux 6; note the lack of the word Enterprise appended to the operating system’s name there, all the way through to CentOS 3 and 4.

Things got complicated when I changed roles; I stopped consuming services from a web hosting company and started working for one. Suddenly, I was building servers for customers who may have different requirements than my projects—each server was different.

From here, I started working with Kickstart scripts, PXE boot servers, gold masters on imaging servers, virtual machines, and bash scripts that started prompting information on the system being built. I had also moved from only needing to worry about maintaining my servers to having to log in to hundreds of different physical and virtual servers, from ones that belonged to the company I was working for to customer machines.

Over the next few years, my single text file quickly morphed into a complex collection of notes, scripts, precompiled binaries, and spreadsheets of information that only made sense to me; if I am being honest, I ended up making myself quite a significant single point of failure.

While I had moved to automate quite a few parts of my day-to-day work using bash scripts and stringing commands together, I found that my days were still very much filled with running all these tasks manually and working a service desk dealing with customer-reported problems and queries.

My story is typical of many people, while the operating systems used will probably be considered ancient. The entry point of using a GUI and moving to the command line while keeping a scratch pad of common commands is quite a common scenario I have heard when working with other system administrators and even modern-day DevOps practitioners.

So now that you know a little about my background, let’s talk about Ansible.

Ansible’s story

Let’s take a quick look at who developed Ansible and what it actually is.

What is Ansible?

Before we discuss how Ansible started, we should quickly discuss the origin of the name. The term “Ansible” was penned by science fiction novelist Ursula K. Le Guin; it was first used in her novel *Rocannon’s World*, which was first published in 1966. In the story’s context, an Ansible is a fictional device that sends and receives messages faster than light.

Note

In 1974, Ursula K. Le Guin’s novel *The Dispossessed: An Ambiguous Utopia* was published. This book features the development of the Ansible technology by exploring the (fictional) details of the mathematical theory that would make such a device possible.

The term has since been used by several other notable authors within the genre to describe communication devices that are capable of relaying messages over interstellar distances, which, as you will discover throughout the course of the book, is quite an apt description of the software itself.

Ansible, the software

Ansible was initially developed by Michael DeHaan, who was also the author of **Cobbler**, which was developed while DeHaan worked for Red Hat.

Note

Cobbler is a Linux installation server that allows you to deploy servers within your network quickly; it can help with DNS, DHCP, package updates and distribution, virtual machine deployment, power management of physical hosts, and also the handoff of a newly deployed server, be it physical or virtual, to a configuration management system.

DeHaan left Red Hat and worked for companies such as **Puppet**, which was a good fit since many users of Cobbler used it to hand off to a Puppet server to manage the servers once they had been provisioned, myself included.

A few years after leaving Puppet, DeHaan made the first public commit to the Ansible project on February 23, 2012. The original README file gave quite a simple description that laid the foundation for what Ansible would eventually become:

Ansible is an extra-simple Python API for doing ‘remote things’ over SSH. As Func, which I co-wrote, aspired to avoid using SSH and have its own daemon infrastructure, Ansible aspires to be quite different and more minimal, but still able to grow more modularly over time.

Since that first commit, and at the time of writing, there have been over 53,000 commits by 5,000 contributors, and the project has over 58,000 stars on GitHub.

In 2013, the project had grown. Ansible, Inc. was founded to offer commercial support to users who had relied on the project to manage their infrastructure and server configuration, whether physical, virtual, or hosted on public clouds.

Out of the formation of Ansible, Inc., which received $6 million in series A funding, came the commercial Ansible Tower, which acted as a web-based frontend where end users could consume role-based access to Ansible services.

Then, in October 2015, Red Hat announced they would acquire Ansible for $150 million.

In the announcement, Joe Fitzgerald, who was vice president, Management, at Red Hat at the time of the acquisition, stated, *“Ansible is a clear leader in IT automation and DevOps, and helps Red Hat take a significant step forward in our goal of creating frictionless IT.”*

During this book, you will find that the statements in the original README file and Red Hat’s statement when acquiring Ansible still ring true.

Before we look at rolling our sleeves up and installing Ansible, which we will do later in this chapter, we should look at some of its core concepts.

Ansible versus other tools

If you look at the design principles in the first commit compared to the current version, you will notice that while there have been some additions and tweaks, the core principles remain pretty much intact:

* **Agentless**: Everything should be managed by the SSH daemon using the WinRM protocol in the case of Windows machines or API calls—there should be no reliance on custom agents or additional ports that need to be opened or interacted with on the target host. The machine running Ansible should need line of sight of the target resource network-wise.
* **Minimal**: You should be able to manage new remote machines without installing any new software on the target host; each Linux target host will typically have at least SSH and Python installed as part of a minimal installation, which is all needed to run Ansible.
* **Descriptive**: You should be able to describe your infrastructure, stack, or task in a language readable by machines and humans.
* **Simple**: The setup processes and the learning curve should be simple and intuitive.
* **Easy to use**: It should be the most accessible IT automation system ever.

A few of these principles make Ansible quite different from other tools. Let’s examine the fundamental difference between Ansible and other tools, such as Puppet and **Chef**.

Declarative versus imperative

When I started using Ansible, I had already implemented Puppet to help manage the stacks on the machines I was managing. As the configuration became increasingly complex, the Puppet code became highly complicated. This was when I started looking at alternatives, and some fixed some of the issues I was facing.

Puppet uses a custom declarative language to describe the configuration. Puppet then packages this configuration as a manifest that the agent running on each server then applies.

Using declarative language means that Puppet, Chef, and other configuration tools, such as **CFEngine**, all operate using the principle of eventual consistency, meaning that eventually, after a few runs of the agent, your desired configuration would be in place.

On the other hand, Ansible is an imperative language that, rather than just defining the end state of your desired outcome and letting the tool decide how it should get there, you also define the order in which tasks are executed to reach the state you have defined.

The example I use is as follows. We have a configuration where the following states need to be applied to a server:

1. Create a group called Team.
2. Create a user Alice and add her to the group Team.
3. Create a user Bob, and add him to the group Team.
4. Give the user Alice escalated privileges.

This may seem simple; however, when you execute these tasks using declarative language, you may, for example, find that the following happens:

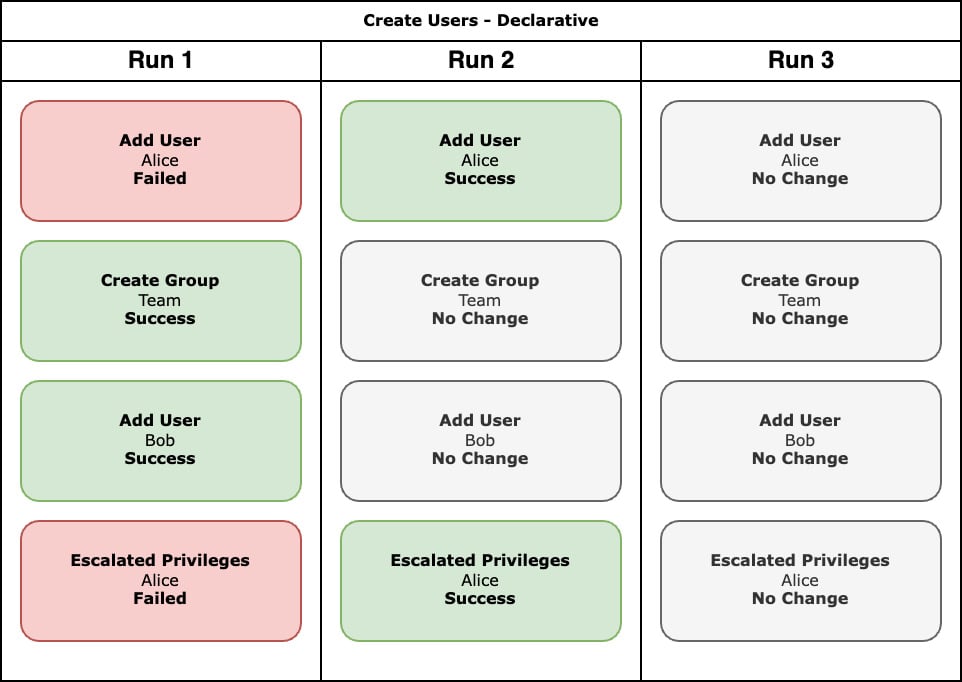


Figure 1.1 – An overview of what happens with the declarative run

So, what has happened here? Our tool has executed the tasks during **run 1** in the order of *2*, *1*, *3*, and *4*, meaning that the user Alice could not be created when the first task ran because the group Team did not exist.

However, as the group Team was created before the user Bob was created, Bob’s user was created without any errors, and the final task, adding escalated privileges to the user Alice, failed because no user called Alice existed on the system for the escalated privileges to be applied to.

During **run 2**, the tasks were executed in the same order as **run 1**, but this time as a group called Team existed, the user Alice was created, and because Alice was present, that user was given escalated privileges.

No changes were needed during **run 3** as everything was as expected; that is, consistent.

Each subsequent run would continue until there was either a change to the configuration or on the host itself, for example, if Bob had annoyed Alice and she used her escalated privileges to remove the user Bob from the host. When the agent subsequently runs, Bob will be recreated as that is still our desired configuration, no matter what access Alice thinks Bob should have.

If we were to run the same tasks using an imperative language, then the following should happen:

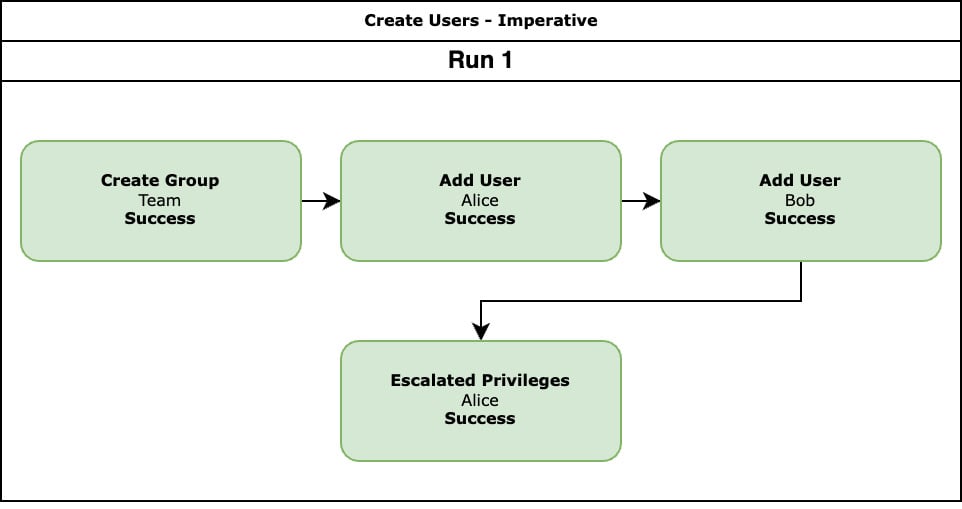


Figure 1.2 – An overview of what happens with the imperative run

The tasks are executed in the order we defined them, meaning that the Team group is created, the Alice and Bob users are added, and the escalated privileges to the Alice user are applied.

As you can see, both ways get to our final configuration and enforce our desired state. With the tools that use declarative language, it is possible to declare dependencies, meaning we can engineer the issue we encountered when running the tasks.

However, this example only has four steps; what happens when you have a few hundred steps that are launching servers in public cloud platforms and then installing software that needs several prerequisites?

This is the position I found myself in before I started to use Ansible. Puppet was great at enforcing my desired end configuration; however, to get there, I had to worry about building so much logic into my manifests to arrive at my desired state. In Puppet, this logic was using a function that allowed me as the end user to define my dependencies.

In the example we used, I would have had to define that users could only be created once the block of code that created the group had been run and the resource was present.

The more complex my code got, the more I fought the way the declarative tools wanted to run and the longer each execution would take because the tool had to consider my logic, which was a little hit and miss.

This became more annoying, as each successful run was getting close to taking about 40 minutes to complete. If I had dependency issues, I had to start from scratch with each failure and change to ensure that I was fixing the problem and not because things were starting to become consistent, so that typically meant having to redeploy a resource rather than running subsequent runs of my code. This made development very time-consuming, especially when it came to debugging the code, which sometimes involved trial and error.

This is not a great position to find yourself in when you are on the clock and must meet customer deadlines.

Configuration versus orchestration

Another critical difference between Ansible and the other tools it is commonly compared to is that most of these tools have their origins as systems designed to deploy and police a configuration state.

They typically require an agent to be installed on each host; that agent discovers some information about the host it is installed on, then calls back to a central server saying, “*Hi, I am server XYZ. Could I please have my configuration?*” The server then decides what the configuration for the server looks like and sends it across to the agent, which then applies it. Typically, this exchange takes place every 15 to 30 minutes—this is great if you need to enforce a configuration on a server.

However, the way that Ansible has been designed to run allows it to act as an orchestration tool; for example, you can run it to launch a server in your VMware environment, and once the server has been launched, it can then connect to your newly launched machine and install a LAMP stack. Then, it never has to connect to that host again, meaning that all we are left with is the server, the LAMP stack, and nothing else, other than maybe a few comments in files to say that Ansible added some lines of configuration, which should be the only sign that Ansible was used to configure the host.

Looking at some code

Before we finish this part of the chapter and move on to installing Ansible, let’s quickly look at examples of some actual code. The following bash script installs several RPMs using the yum package manager:

#!/bin/sh

LIST\_OF\_APPS="dstat lsof mailx rsync tree vim-enhanced git whois"

yum install -y $LIST\_OF\_APPS

CopyExplain

The following is a Puppet class that does the same task as the previous bash script:

class common::apps {

  package {

    [

      'dstat',

      'lsof',

      'mailx',

      'rsync',

      'tree',

      'vim-enhanced',

      'git',

      'whois',

    ]:

    ensure => installed,

  }

}

CopyExplain

Next up, we have the same task using **SaltStack**:

common.packages:

  pkg.installed:

    - pkgs:

      - dstat

      - lsof

      - mailx

      - rsync

      - tree

      - vim-enhanced

      - git

      - whois

CopyExplain

Finally, we have the same task again, this time using Ansible:

- name: "Install packages we need"

  ansible.builtin.yum:

    name:

      - "dstat"

      - "lsof"

      - "mailx"

      - "rsync"

      - "tree"

      - "vim-enhanced"

      - "git"

      - "whois"

      - "iptables-services"

    state: "present"

CopyExplain

Even without going into any detail, you should be able to get the general gist of what each of the three examples is doing. While not strictly infrastructure, all three are valid examples of infrastructure as code.

This is where you manage the code that manages your infrastructure in precisely the same way as a developer would manage the source code for their application. You use source control, store it in a centrally available repository where you can collaborate with your peers, branch and use pull requests to check in your changes, and, where possible, write and execute unit tests to ensure that changes to your infrastructure are successful and error-free before deploying to production. This should be as automated as possible. Any manual intervention in the tasks mentioned could be a point of failure, and you should work to automate the task.

This approach to infrastructure management has a few advantages, one being that you, as system administrators, are using the same processes and tooling as your developer colleagues, meaning that any procedures that apply to them also apply to you. This makes for a more consistent working experience and exposes you to tools you may have yet to be exposed to or use.

Secondly, and more importantly, it allows you to share your work. Before this approach, this type of work seemed to others a dark art performed only by system administrators. Doing this work in the open allows you to have your peers review and comment on your configuration and do the same yourself to theirs. Also, you can share your work so that others can incorporate elements into their projects.

My story: part two

Before we finish this part of the chapter, I would like to finish the story of my journey. As mentioned earlier in the chapter, I moved from my collection of scripts and runbooks to Puppet, which was great until my requirements moved away from managing just server configuration and maintaining the servers’ state.

I needed to start managing infrastructure in public clouds. This requirement quickly started to frustrate me when using Puppet. At the time, Puppet’s coverage of the APIs I needed to use for my infrastructure needed to be improved. I am assured it is a lot better now, but also, I found myself having to build too much logic into my manifests about the order in which each task was executed.

It was around this time, which was December 2014, that I decided to look at Ansible. I know the date because I wrote a blog post entitled *First Steps With Ansible*; I don’t think I have looked back since. I have since introduced several of my work colleagues and customers to Ansible and have written books on the subject, including the first edition of the title you are reading now.

So far in this chapter, we have looked at my personal history with both Ansible and some of the other tools that Ansible is compared to, and we have discussed the differences between these tools and where Ansible originated.

Now we are going to start your journey with Ansible by looking at installing it and running our first Ansible playbook against a local virtual machine.

Installing and running Ansible

Let’s dive straight in and install Ansible. Throughout this book, I will assume you are running a macOS host machine or a Linux machine with an Ubuntu LTS release. While we will cover running Ansible on Windows 11 using the Linux subsystem for Windows, this book will not support using Windows as a host machine.

Installing on macOS

You can install Ansible on your macOS host machine in a few different ways. I will cover both here. As we are discussing two different ways of installing Ansible, I recommend reading through this section and the *Pros and cons* subsection before choosing which installation method to use on your local machine.

Installing with Homebrew

The first installation method is to use a package manager called **Homebrew**.

Note

Homebrew is a package manager of macOS. It can be used to install command-line tools and desktop packages. It describes itself as *“The missing package manager for macOS”*, and it usually is one of the first tools I install after a clean installation or when getting a new computer.

To install Ansible using Homebrew, you first need to install Homebrew. To do this, run the following command:

$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

CopyExplain

At each step of the installation process, the installer will tell you exactly what it is going to do and also prompt you for any additional information it needs from you to complete the installation.

Once installed, or if you already have Homebrew installed, run the following commands to update your list of packages; if there are any updates, then you can also upgrade them:

$ brew update

$ brew upgrade

CopyExplain

Finally, depending on how new your installation is or when you last used it, you might want to run the following command to check that your Homebrew installation is optimal:

$ brew doctor

CopyExplain

Now that we have Homebrew installed, updated, and ready to brew, we can run the following to check what Ansible packages Homebrew has by running:

$ brew search ansible

CopyExplain

As you can see from the results in the following screenshot, there are several packages returned in the search:

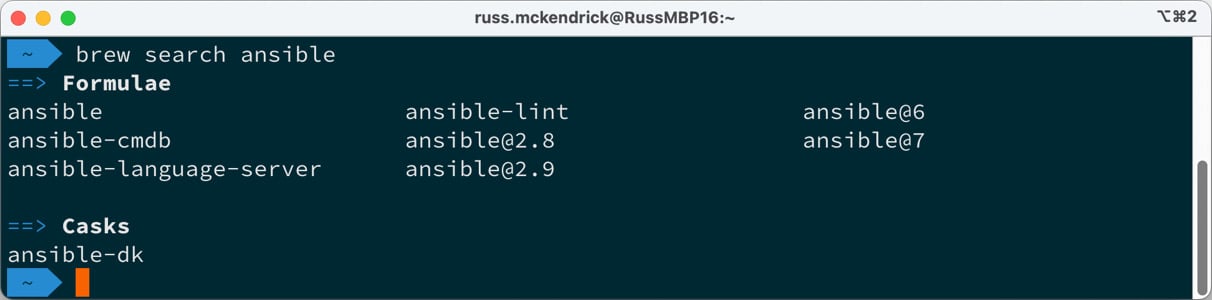


Figure 1.3 – Searching for Ansible using the brew command

We want the Ansible package; you can find out more about the package by running the following command:

$ brew info ansible

CopyExplain

You can see the results of the command in the following screenshot:

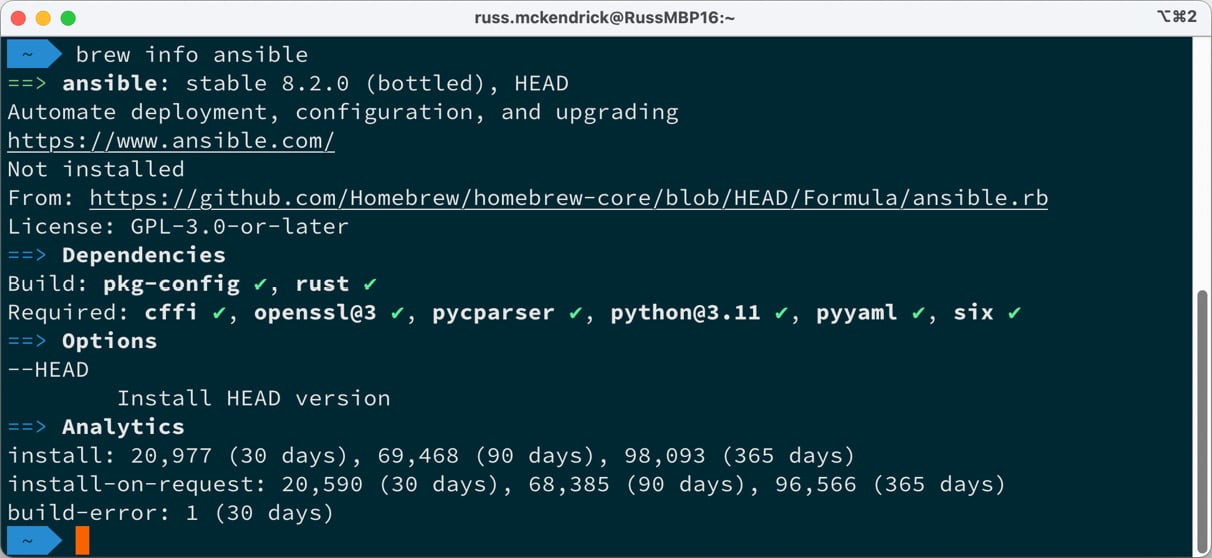


Figure 1.4 – Viewing information on the Ansible package we will install

As you can see, the command returns information on the version of the package that will be installed along with a complete list of its dependencies; in the preceding screen, all of the dependencies have green ticks next to them because I already have them installed—yours may look different.

It also gives the URL to the Homebrew formula, which will be used to install the package. In our case, you can view formula details at https://github.com/Homebrew/homebrew-core/blob/master/Formula/ansible.rb.

To install Ansible using Homebrew, we must run the following command:

$ brew install ansible

CopyExplain

This will download and install all dependencies and then the Ansible package itself.

Depending on how many dependencies are installed on your machine, this may take a few minutes.

Once installed, you should see something like the following screenshot:



Figure 1.5 – Installing Ansible using Homebrew

As you can see from the preceding screenshot, Homebrew is quite verbose in its output, giving you both feedback on what it is doing and details on how to use the packages it installs.

The second of two installation methods we will look at for macOS is a more traditional one.

Installing using pip

The second method, pip, is a more traditional approach to installing and configuring a Python package.

Note

pip is a package manager for Python software. It is a recursive acronym for **pip install packages**. It is a good frontend for installing packages from the **Python Package Index** (**PyPI**).

Most modern macOS installations come with pip installed by default; depending on what you have installed on your machine, you may have to check which pip binary you have installed.

To do this, run the following commands:

$ pip --version

$ pip3 --version

CopyExplain

One or both should return a version number and give you the path to the pip binary.

Depending on the version of pip you have installed, you may need to amend the following pip command, which is what we need to run to install Ansible:

$ pip install ansible

CopyExplain

This command will download and install all the prerequisites to run Ansible on your system. While it is as verbose as Homebrew, its output contains information on what it has done rather than hints on what to do next:

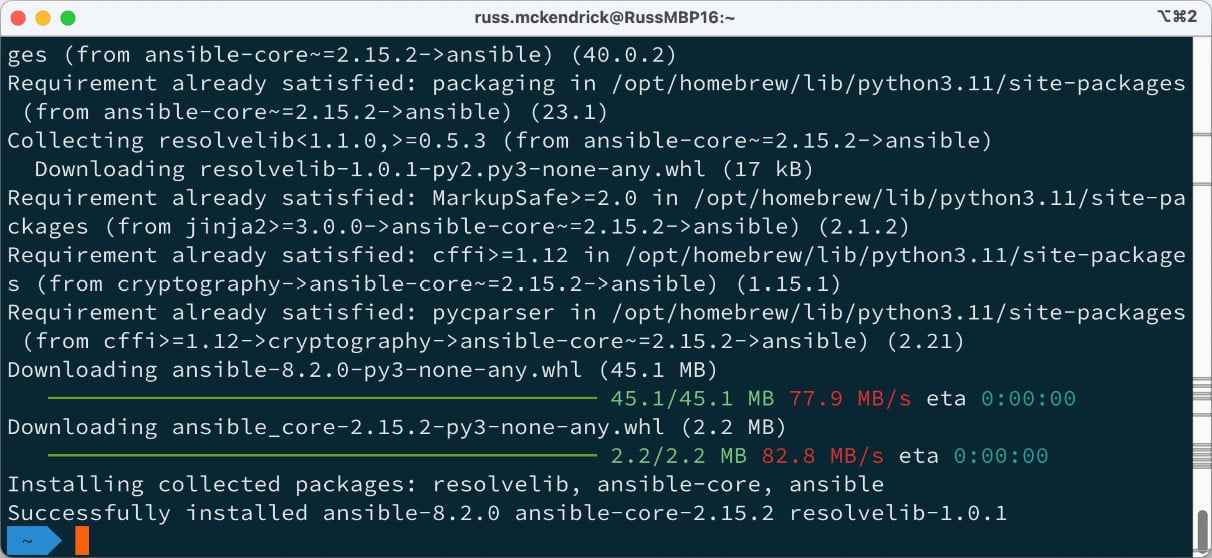


Figure 1.6 – Installing Ansible using Pip

As you can see from the small amount of output, many of the requirements were already satisfied.

Pros and cons

So, now that we have covered some of the different ways of installing Ansible on macOS, which is best? Well, there is no real answer to this as it comes down to personal preference. Both methods will install the latest versions of Ansible. However, Homebrew tends to be a week or two behind the current release.

If you have a lot of packages already installed using Homebrew, then you will be used to running the following commands:

$ brew update

$ brew upgrade

CopyExplain

Just run these occasionally to update your installed packages to the latest versions. If you already do this, it makes sense to use Homebrew to manage your Ansible installation.

If you are not a Homebrew user and want to ensure that you immediately have the latest version installed, use the pip command to install Ansible. Upgrading to the latest version of Ansible is as simple as running the following command:

$ pip install ansible --upgrade

CopyExplain

Should you need to, you can install older versions of Ansible using Homebrew and pip.

To do this using Homebrew, you need to remove the current version by running the following command:

$ brew uninstall ansible

CopyExplain

Then, you can install an earlier version of the package by running the following command:

$ brew install ansible@2.0

CopyExplain

While this will install an earlier version, you do not have much choice in which version you get. If you really need an exact version, you can use the pip command to install it. For example, to install Ansible 2.3.1.0, you would need to run:

$ pip install ansible==2.3.1.0

CopyExplain

It is essential to note that you should never need to do this, and I do not recommend it.

However, I have found that on rare occasions, I have had to downgrade to help debug quirks in my playbooks introduced by upgrading to a later version of Ansible for playbooks that I last touched a few years ago.

As mentioned, I spend the bulk of my time in front of a macOS machine both during my day job and at home, so which of the two methods do I use?

Primarily, I use Homebrew as I have several other tools installed using Homebrew. However, if I need to roll back to a previous version, I use pip and then return to Homebrew once the issue is resolved.

Installing on Linux

There are a few different ways of installing Ansible on Ubuntu. However, I am only going to cover one of them here. While there are packages available for Ubuntu that you can install with apt, they tend to become out of date quickly and are typically behind the current release.

If you wish to install using the apt package manager, then you can run the following command:

$ apt install ansible

CopyExplain

Note

**Advanced Packaging Tool** (**APT**) is the package manager that ships with Debian-based systems, including Ubuntu. It is used to manage .deb files.

Because of this, we will be using pip. The first thing to do is install pip, and this can be done by running the following commands:

$ sudo -H apt-get update

$ sudo -H apt-get install python3-pip

CopyExplain

The first of the two apt-get commands downloads all the update files, ensuring that the package list is up to date on your Ubuntu installation, and the second command installs the python3-pip package and its dependencies.

Once pip is installed, the instruction for installing Ansible is similar to installing on macOS. Run the following command:

$ sudo -H pip install ansible

CopyExplain

This will download and then install Ansible and its requirements, as shown in the following screenshot:

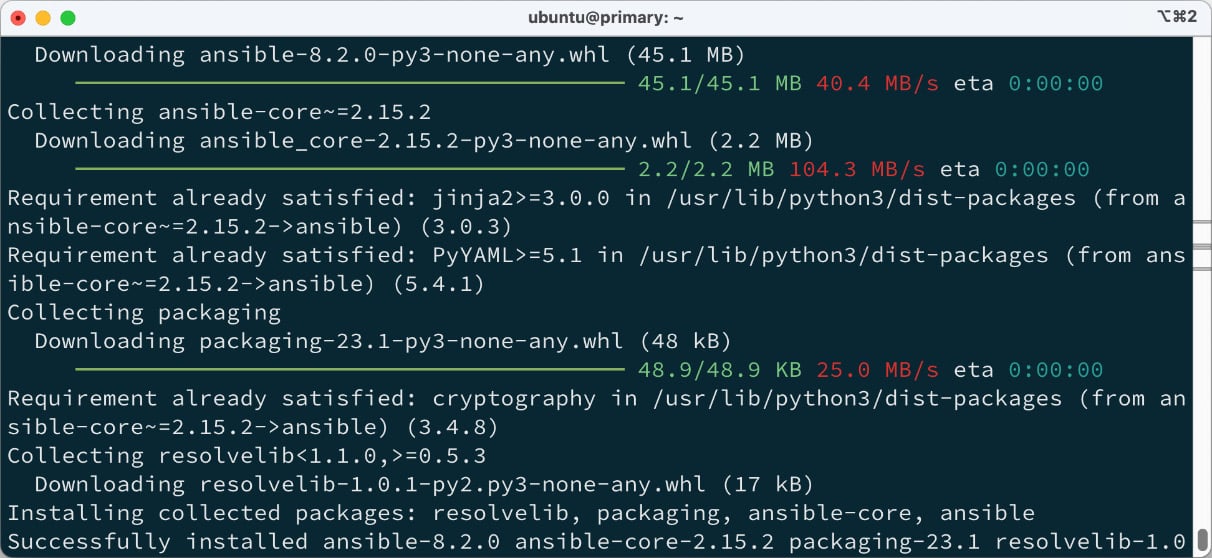


Figure 1.7 – Installing Ansible using pip on Ubuntu

Once installed, you can upgrade it by using the following command:

$ sudo -H pip install ansible --upgrade

CopyExplain

Also, downgrading Ansible uses the same command:

$ sudo -H pip install ansible==2.3.1.0

CopyExplain

The preceding commands should work on most Linux distributions, such as Rocky Linux, Red Hat Enterprise Linux, Debian, and Linux Mint, to name a few.

A lot of these distributions have their own package managers you can also use to install Ansible; for example, on Red Hat-based distributions such as Red Hat Enterprise Linux or Rocky Linux, you could also run:

$ dnf install ansible-core

CopyExplain

Please consult the document for more details on installing whatever your Linux distribution of choice is.

Installing on Windows 11

The last platform we will cover is Windows 11; well, sort of. While it is technically possible to run Ansible on a Windows 11 natively, it is not something I would recommend attempting as it is one of those tasks where the phrase “*just because you can doesn’t mean you should*” applies, as getting all of the dependencies installed and configured just right can be very troublesome, and maintaining them is more so.

Luckily, Microsoft—and as a long-term Linux System administrator typing this, this still feels strange—has excellent native support for running Linux systems seamlessly within Windows 11.

Open the Microsoft Store and search for Ubuntu; you should see something like the following screen:

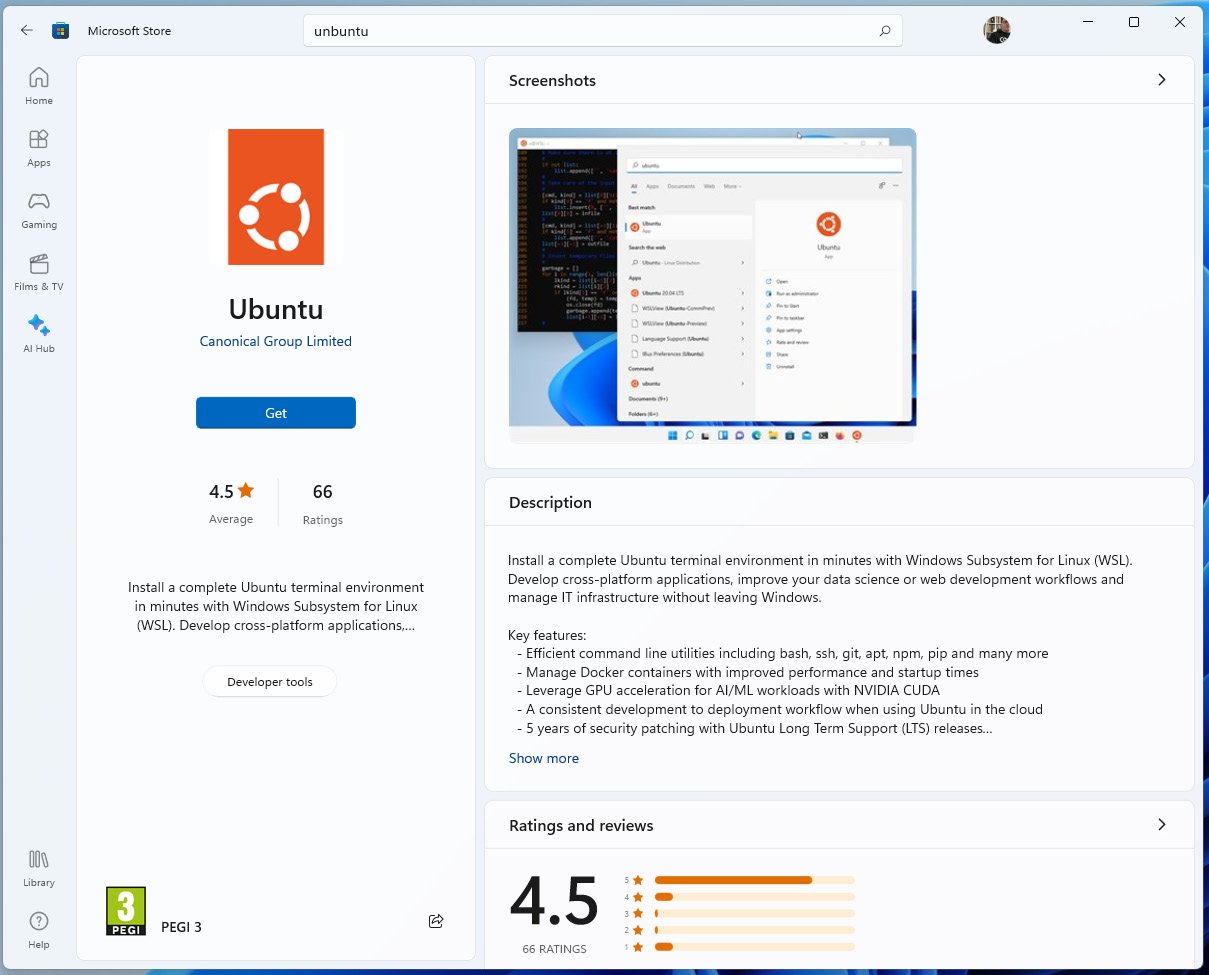


Figure 1.8 – Finding Ubuntu in the Microsoft Store

Click the **Get** button to download Ubuntu. Once downloaded, we have Ubuntu on our Windows 11 host, but we still need something to run it on. To run it, we need to enable Windows Subsystem for Linux.

To enable this, open a PowerShell window by typing **PowerShell** into your Windows Search bar and opening the **Windows PowerShell** application; once you are at a terminal prompt, run the following command:

$ wsl --install

CopyExplain

Follow the onscreen prompts, and once everything is installed, restart your Windows 11 host.

Once rebooted, you should have something like the following prompt pop-up once you log back in:

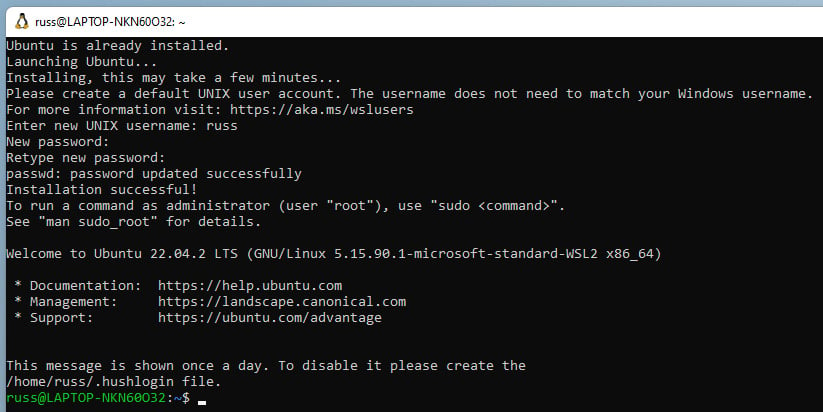


Figure 1.9 – Completing the Ubuntu installation on Windows 11

Once the installation is completed, I like to switch out of the default Windows Subsystem for the Linux terminal and use the Microsoft Terminal, which you can grab for free in the Microsoft Store.

Once you have your preferred terminal emulator open and you are sitting at a prompt in your Windows Subsystem for Linux Ubuntu installation, you can run the same commands we ran to install Ansible on Linux, which are as follows:

$ sudo -H apt-get update

$ sudo -H apt-get install python3-pip

$ sudo -H pip install ansible

CopyExplain

Once you have run these commands, you should see an output that looks like the following screenshot:

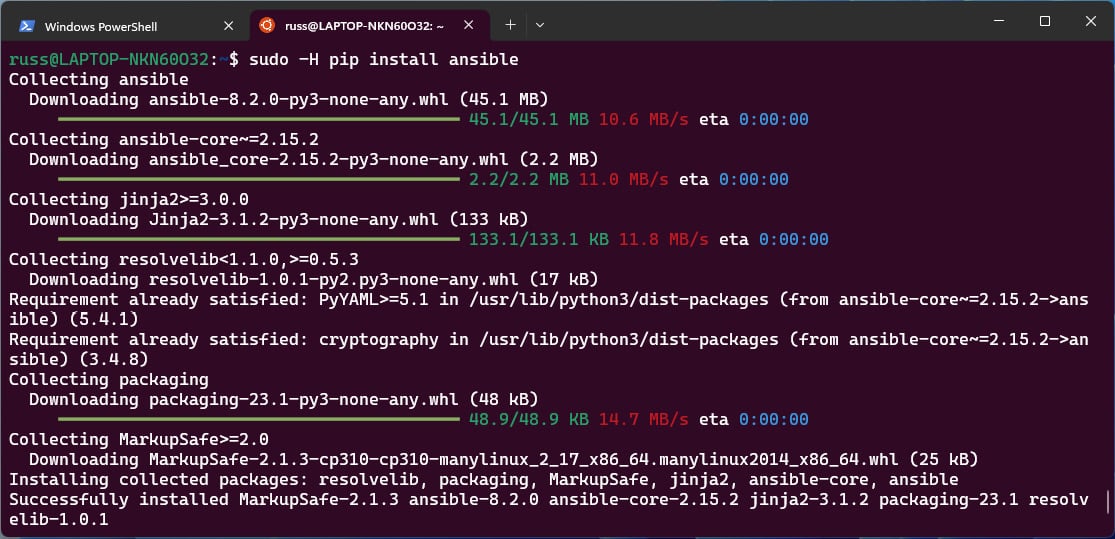


Figure 1.10 – Installing Ansible in Ubuntu on Windows 11

As you can see, everything works as if you were running an Ubuntu machine, allowing you to run and maintain your Ansible installation in precisely the same way.

Note

The **Windows Subsystem for Linux** (**WSL**) is not running on a virtual machine. It is a full native Linux experience baked right into Windows 11. It targets developers who must run Linux tools as part of their toolchain. While the overall support for Linux commands is excellent, I recommend reading through the FAQs written and maintained by Microsoft to get an idea of the limits and quirks of the subsystem. The FAQ can be found at https://learn.microsoft.com/en-us/windows/wsl/faq.

As mentioned, while this is a viable way of running an Ansible control node on a Windows-based machine, some of the other tools we will cover in future chapters may not work with Windows. So, while you may follow along using the Ubuntu instructions, some parts may not work.

Launching a virtual machine

To launch a virtual machine to run our first set of Ansible commands against, we will use a tool called **Multipass**. This tool allows you to run Ubuntu virtual machines on your local host. It works with macOS, Linux, and Windows.

To install Multipass on macOS, we can use Homebrew and run the following command:

$ brew install multipass

CopyExplain

To install on Ubuntu, you can run the following command:

$ snap install multipass

CopyExplain

Finally, for Windows 11 users, you will have to first download and install the VirtualBox Windows executable from https://www.virtualbox.org/wiki/Downloads and then download and install Multipass from https://multipass.run/install. I recommend reading through the installation notes for Windows, which can be found at the following URL https://multipass.run/docs/installing-on-windows before installing.

Note

While you can run the same commands in Ubuntu running under Windows Subsystem for Linux, you will need to replace all references of the multipass command with multipass.exe so that the Windows version of Multipass is called.

Next, check out the GitHub repo accompanying this title and open your terminal in the Chapter01 folder—if you are running Windows 11, you must open an Ubuntu terminal, not a Windows one.

Important

Before we start, a quick word of warning: the folder Chapter01 contains an OpenSSH key-pair, which will be used to access the local machine. It is important that you do not use this key-pair anywhere other than this example on your local machine as the key-pair is publicly accessible, which is not considered secure.

You will see several files in the Chapter01 folder. The one we are going to use when launching the virtual machine is called **cloud-init.yaml**. This file contains some back configuration to add a user called vmadmin and attach the public portion of an OpenSSH key to the user, meaning that when executing Ansible, we can use the private part of the OpenSSH key to authenticate as the vmadmin user.

The command we are going to run to launch the virtual machine, which will be called **ansiblevm**, is as follows:

$ multipass launch -n ansiblevm --cloud-init cloud-init.yaml

CopyExplain

Once the virtual machine has been launched, which may take a short while when you first run the command as it will download a virtual machine image, you need to run the following command to get some information on the newly created ansiblevm virtual machine:

$ multipass info ansiblevm

CopyExplain

The following screen shows me starting and viewing the information on the virtual machine:

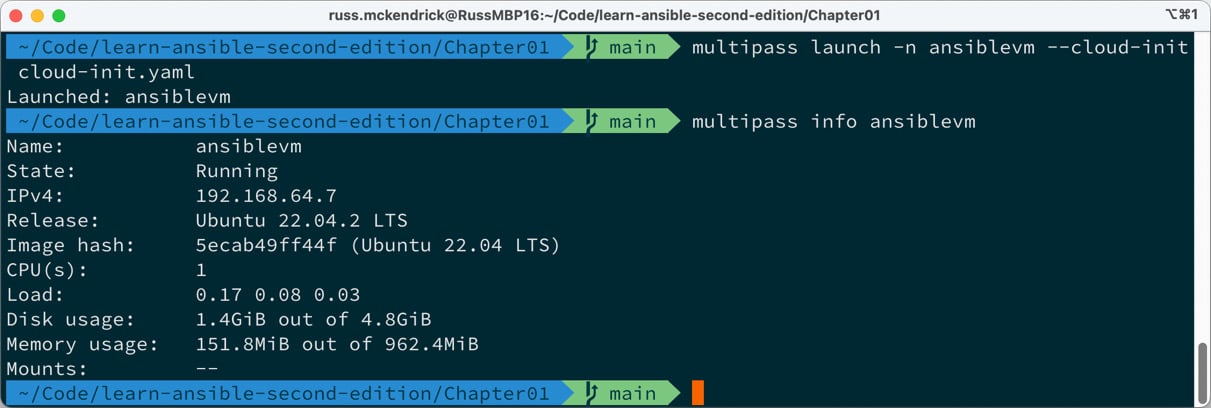


Figure 1.11 – Launching our virtual machine

Now that we have our virtual machine running and we have checked out the basic information, you will need to note the IP address, which in my case is 192.168.64.7. The IP address will be different when you launch the virtual machine on your host.

Before we run our first Ansible playbook, you must make a copy of the hosts-simple.example and hosts.example files and remove the .example in the filename by running the following commands:

$ cp -pr hosts-simple.example hosts-simple

$ cp -pr hosts.example hosts

CopyExplain

Once you have made a copy of the file, open the newly created files and replace just the text that says paste\_your\_ip\_here with the IP address of the ansiblevm virtual machine; in my case, the hosts-simple file went from:

paste\_your\_ip\_here.nip.io ansible\_user=vmadmin ansible\_private\_key\_file=./example\_key

CopyExplain

to reading:

192.168.64.7.nip.io ansible\_user=vmadmin ansible\_private\_key\_file=./example\_key

CopyExplain

Once you have changed both the hosts-simple and hosts files, you are ready to run your first Ansible Playbook.

An introduction to playbooks

Typically, in IT, a **playbook** is a set of instructions run by someone when something happens; a little vague, I know, but stay with me. These range from building and configuring new server instances to deploying code updates and dealing with problems when they occur.

In the traditional sense, a playbook is typically a collection of scripts or instructions for a user to follow, and while they are meant to introduce consistency and conformity across systems, even with the best intentions, this is seldom the case.

This is where Ansible comes in. Using an Ansible playbook, you are telling it to apply these changes and commands against these sets of hosts rather than having to log in and start working your way through the playbook manually.

Before we run a playbook, let’s discuss how we provide Ansible with a list of hosts to target. To do this, we will be using the ansible.builtin.setup module. This connects to a host and then fetches as much information on the host as possible.

Host inventories

To provide a list of hosts, we need to provide an inventory list. This is in the form of a host’s file.

In its simplest form, our host’s file could contain a single line just like our hosts-simple file:

192.168.64.7.nip.io ansible\_user=vmadmin ansible\_private\_key\_file=./example\_key

CopyExplain

This tells Ansible that the host we want to contact is 192.168.64.7.nip.io (please remember your IP address will be different) using the username vmadmin. If we didn’t provide the username, it would fall back to the user you are logged into your Ansible control host as, which in my case is the user russ, which does not exist on the ansiblevm we launched. The final part tells Ansible to use the private OpenSSH key file called example\_key, which we installed the public portion of to the vmadmin user when we launched the virtual machine.

Note

We are using https://nip.io, a free service that provides free wildcard DNS entries for any hostname containing an IP address. This means that our domain 192.168.64.7.nip.io will resolve to 192.168.64.7 when a DNS lookup is made against the domain.

To run the ansible.builtin.setup module, we need to run the following command from within the Chapter01 folder where your updated hosts-simple and example\_key files are stored, making sure to update the IP address to your own:

$ ansible -i hosts-simple 192.168.64.7.nip.io -m ansible.builtin.setup

CopyExplain

If everything works as expected, you should see a lot of output, which specifies some quite detailed and low-level information about your host. You should see something like the following:

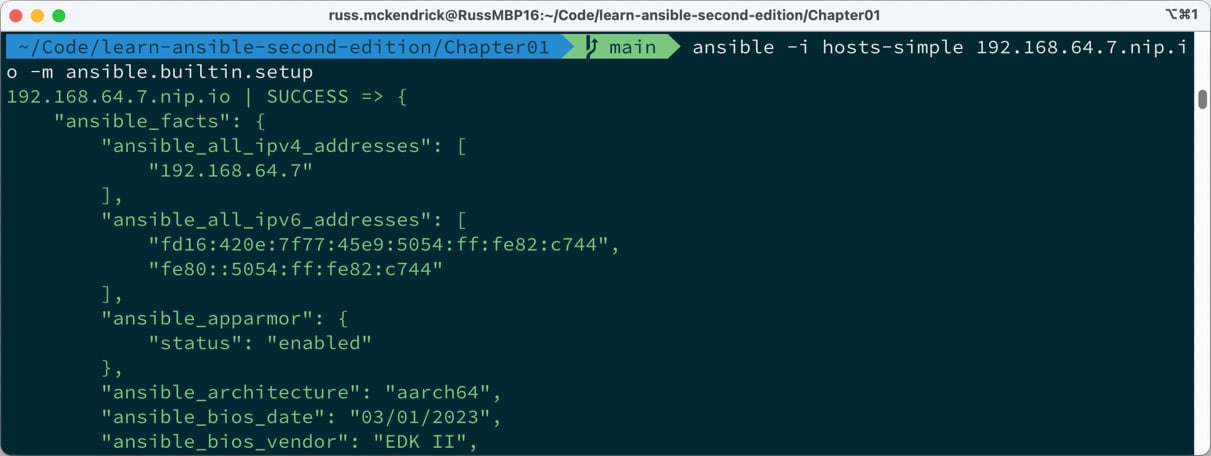


Figure 1.12 – The start of the output of me running the ansible.builtin.setup module

As you can see from the preceding screenshot, Ansible has quickly found out a lot of information on our Vagrant box. The screenshot shows the IP addresses configured on the machine, along with the IPv6 addresses. It has recorded the time and date, and if you scroll through your output, you will see a lot of information returned detailing the host.

Let’s go back to the command we ran:

$ ansible -i hosts-simple 192.168.64.7.nip.io -m ansible.builtin.setup

CopyExplain

As you can see, we are loading the hosts-simple file using the -i flag. We could have also used --inventory=hosts-simple, which loads our inventory file. The next part of the command is the host to the target. In our case, this is 192.168.50.4.nip.io. The final part of the command, -m, tells Ansible to use the setup module. We could have also used --module-name= ansible.builtin.setup.

This means that the full command if we didn’t use shorthand would be:

$ ansible --inventory=hosts-simple simple 192.168.64.7.nip.io --module-name=ansible.builtin.setup

CopyExplain

As already mentioned, the hosts-simple file is as basic as we can get it. The following is a more common host inventory file:

ansiblevm ansible\_host=192.168.64.7.nip.io

[ansible\_hosts]

ansiblevm

[ansible\_hosts:vars]

ansible\_connection=ssh

ansible\_user=vmadmin

ansible\_private\_key\_file=./example\_key

host\_key\_checking=False

CopyExplain

This is the content of the file called just hosts; as you can see, there is a lot more going on, so let’s quickly work through it from top to bottom.

The first line defines our single host. Unlike the simple example, we will be calling our target host ansiblevm and grouping it together in a group called ansible\_hosts, so we are giving Ansible details of where it can SSH to. This means we can now use the name ansiblevm when referring to 192.168.64.7.nip.io. This means our command would now look something like this:

$ ansible -i hosts ansiblevm -m ansible.builtin.setup

CopyExplain

Next up in the file, we are creating a group of hosts called ansible\_hosts and, in that group, we are adding our single host ansiblevm. This means that we can also run:

$ ansible -i hosts ansible\_hosts -m ansible.builtin.setup

CopyExplain

If we had more than just a single host in the group, the preceding command would have looped through all of them. The final section of the hosts file sets up some common configuration options for all of the hosts in the boxes group. In this case, we are telling Ansible that all of the hosts in the group are using SSH, the user is vmadmin, the private key at ./example\_key should be used, and it should not check the host key when connecting.

We will be revisiting the inventory host files in later chapters. From now on, we will use the hosts file to target the ansible\_hosts group.

Playbooks

In the previous section, running the ansible command allowed us to call a single module.

In this section, we are going to look at calling several modules. The following playbook is called **playbook01.yml**. It calls the ansible.builtin.setup module we called in the previous section and then uses the ansible.builtin.debug module to print a message to the screen:

---

- name: "A simple playbook"

  hosts: ansible\_hosts

  gather\_facts: true

  become: true

  become\_method: "ansible.builtin.sudo"

  tasks:

    - name: "Output some information on our host"

      ansible.builtin.debug:

        msg: "I am connecting to {{ ansible\_nodename }} which is running {{ ansible\_distribution }} {{ ansible\_distribution\_version }}"

CopyExplain

Before we break the configuration down, let’s look at the results of running the playbook. To do this, use the following command:

$ ansible-playbook -i hosts playbook01.yml

CopyExplain

This will connect to our host, gather information on the system, and then return just the information we want in a message:

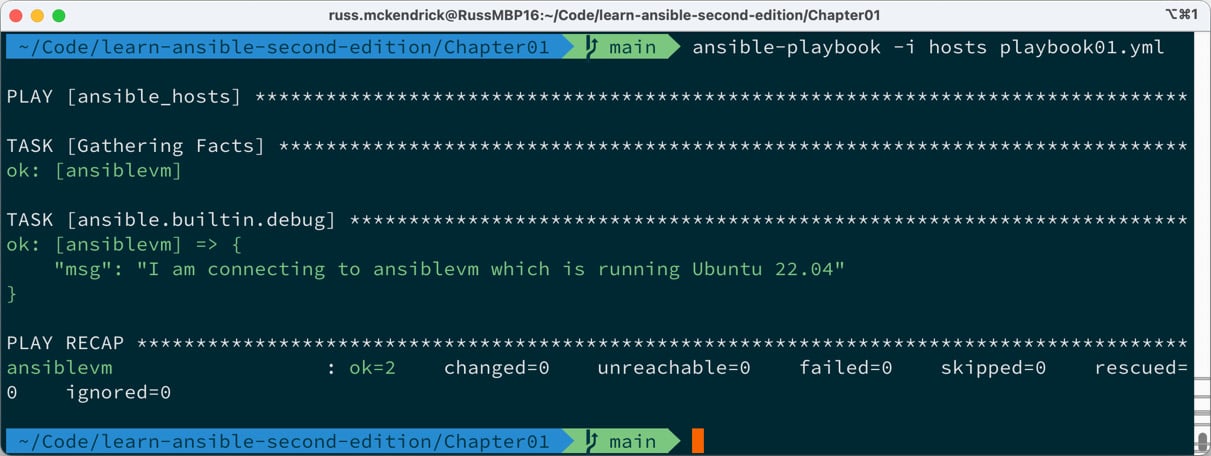


Figure 1.13 – The output of running ansible-playbook01.yml

The first thing you will notice about the playbook is that it is written in **YAML**, a recursive acronym for **YAML Ain’t Markup Language**. YAML was designed to be a human-readable data serialization standard that all programming languages can use. It is commonly used to help define configurations.

The indentation is very important in YAML as it is used to nest and define areas of the file. Let’s look at our playbook in more detail:

---

CopyExplain

While these lines might not seem like much, they are used as document separators, as Ansible compiles all the YAML files into a single file. It is essential for Ansible to know where one document ends and another begins.

Next up, we have the configuration for the playbook. As you can see, this is where the indentation starts to come into play:

- name: "A simple playbook"

  hosts: ansible\_hosts

  gather\_facts: true

  become: true

  become\_method: "ansible.builtin.sudo"

CopyExplain

The - tells Ansible that this is the start of a section. From there, key-value pairs are used. These are as follows:

* name: This gives a name to the playbook run.
* hosts: This tells Ansible the host or host group to target in the playbook. This must be defined in a host inventory like the ones we covered in the previous section.
* gather\_facts: This tells Ansible to run the ansible.builtin.setup module when it first connects to the host. This information is then available to the playbook during the run.
* become: This is present because we are connecting to our host as a basic user, in this case, the **vmadmin** user. Ansible may not have enough access privileges to execute some of the commands we are telling it to, so this instructs Ansible to execute all of its commands as the root user.
* become\_method: This tells Ansible how to become the root user; in our case, we have a passwordless sudo configured by the cloud-init script we ran when launching the virtual machine, so we are using ansible.builtin.sudo.
* tasks: These are the tasks we can tell Ansible to run when connected to the target host.

You will notice that from here, we move the indentation across again. This defines another section of the configuration. This time it is for the tasks:

    - name: "Output some information on our host"

      ansible.builtin.debug:

        msg: "I am connecting to {{ ansible\_nodename }} which is running {{ ansible\_distribution }} {{ ansible\_distribution\_version }}"

CopyExplain

As we have already seen, the only task we run is the ansible.builtin.debug module. This module allows us to display output in the Ansible playbook run stream you saw when we ran the playbook.

You may have already noticed that the information between the curly brackets is made up of the keys from the ansible.builtin.setup module. Here, we are telling Ansible to substitute the value of each key wherever we use the key. We will be using this a lot in our playbooks. We will also be defining our own key values to use as part of our playbook runs.

Let’s extend our playbook by adding another task. The following can be found as **playbook02.yml**:

---

- name: "Update all packages"

  hosts: "ansible\_hosts"

  gather\_facts: true

  become: true

  become\_method: "ansible.builtin.sudo"

  tasks:

    - name: "Output some information on our host"

      ansible.builtin.debug:

        msg: "I am connecting to {{ ansible\_nodename }} which is running {{ ansible\_distribution }} {{ ansible\_distribution\_version }}"

    - name: "Update all packages to the latest version"

      ansible.builtin.apt:

        name: "\*"

        state: "latest"

        update\_cache: true

CopyExplain

As you can see, we have added a second task that calls the ansible.builtin.apt module. This module is designed to help us interact with the package manager used by Ubuntu and other Debian-based operating systems called apt. We are setting three key values here:

* name: This is a wildcard. It tells Ansible to use all of the installed packages rather than just a single named package. For example, we could have used something such as apache2 here to target Apache.
* state: Here, we are telling Ansible to ensure the package we have defined in the name key is the latest version. As we have named all of the installed packages, this will update everything we have installed.
* update\_cache: As the virtual machine image we downloaded was optimized for being small, it does not contain any information on the available package; by setting update\_cache to true, this will download a list of all package and version information.

Run the playbook using the following command:

$ ansible-playbook -i hosts playbook02.yml

CopyExplain

This will give us the following results:

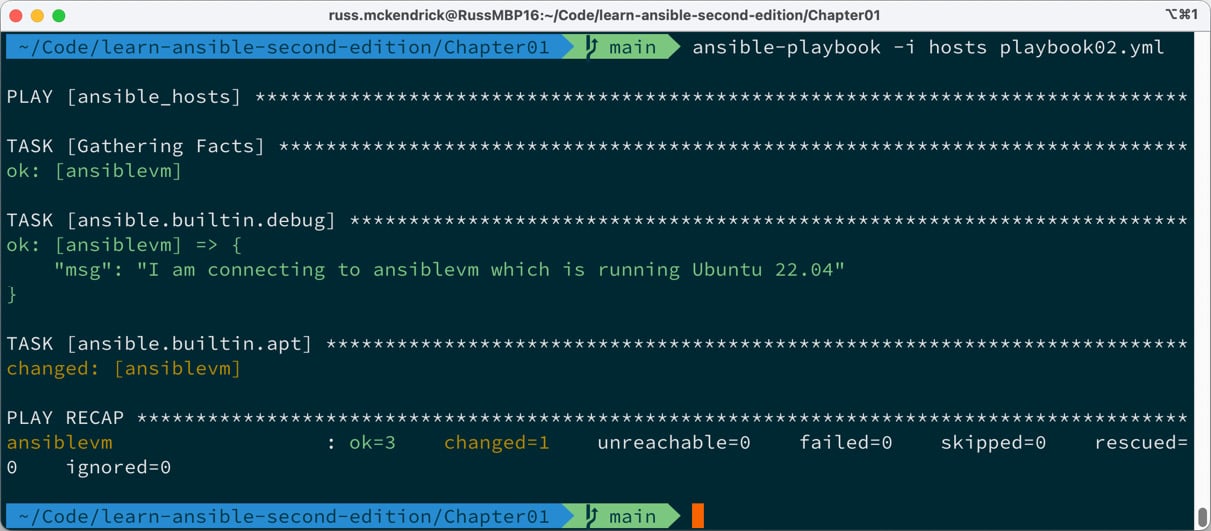


Figure 1.14 – The output of running ansible-playbook02.yml

The ansible.builtin.apt task has been marked as changed on the host box. This means that packages were updated.

Rerunning the same command shows the following results:

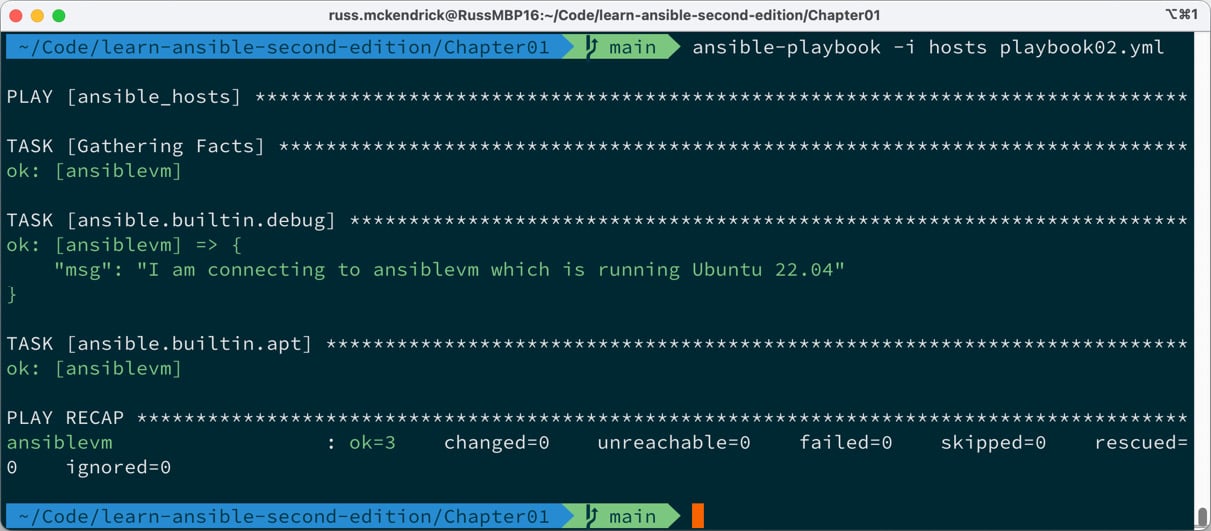


Figure 1.15 – The output of rerunning ansible-playbook02.yml

As you can see, the ansible.builtin.apt task is now showing as ok on our host. This is because there are currently no longer any packages requiring updates.

Before we finish this quick look at playbooks, let’s do something more interesting.

The playbook, playbook03.yml, adds NTP installing, configuring, and starting capabilities to our virtual machine. It also uses a template to add a custom NTP config file to our virtual machine.

The vars section allows us to configure our own key-value pairs. In this case, we are providing a list of NTP servers, which we will be using later in the playbook:

  vars:

    ntp\_servers:

      - "0.uk.pool.ntp.org"

      - "1.uk.pool.ntp.org"

      - "2.uk.pool.ntp.org"

      - "3.uk.pool.ntp.org"

CopyExplain

We are actually providing four different values for the same key. These will be used in the template task. We could have also written this as follows:

  vars:

    ntp\_servers: [ "0.uk.pool.ntp.org", "1. uk.pool.ntp.org", "2. uk.pool.ntp.org", "3. uk.pool.ntp.org" ]

CopyExplain

However, this is a little more difficult to read. The next new section is handlers. A **handler** is a task that is assigned a name and called at the end of a playbook run depending on what tasks have changed:

  handlers:

    - name: "Restart ntp"

      ansible.builtin.service:

        name: "ntp"

        state: "restarted"

CopyExplain

In our case, the restart ntp handler uses the ansible.builtin.service module to restart ntp. Next up, we have two new tasks, starting with installing the NTP service and the sntp and ntp-doc packages using ansible.builtin.apt:

    - name: "Install packages"

      ansible.builtin.apt:

        state: "present"

        pkg:

          - "ntp"

          - "sntp"

          - "ntp-doc"

CopyExplain

As we need to install three packages, we need a way to provide three different package names to the ansible.builtin.apt module so that we don’t have to have three different tasks for each of the package installations. To achieve this, we use the pkg option rather than the name option, where you can only define a single package to install. Rather than using latest, we are using present; this will mean that our packages don’t get updated if they are already installed.

The final addition to the playbook is the following task:

    - name: "Configure NTP"

      ansible.builtin.template:

        src: "./ntp.conf.j2"

        dest: "/etc/ntp.conf"

        mode: "0644"

      notify: "Restart ntp"

CopyExplain

This task uses the ansible.builtin.template module. To read a template file from our Ansible controller, process it and upload the processed template to the host machine. Once uploaded, we are telling Ansible to notify the restart ntp handler if there have been any changes to the configuration file we are uploading.

In this case, the template file is the ntp.conf.j2 file in the same folder as the playbooks, as defined in the src option. This file looks like this:

# {{ ansible\_managed }}

driftfile /var/lib/ntp/drift

restrict default nomodify notrap nopeer noquery

restrict 127.0.0.1

restrict ::1

{% for item in ntp\_servers %}

server {{ item }} iburst

{% endfor %}

includefile /etc/ntp/crypto/pw

keys /etc/ntp/keys

disable monitor

CopyExplain

The bulk of the file is the standard NTP configuration file, with the addition of a few Ansible parts. The first addition is the very first line:

# {{ ansible\_managed }}

CopyExplain

If this line wasn’t there every time we ran Ansible, the file would be uploaded, which would count as a change and the restart ntp handler would be called, meaning that even if there were no changes, NTP would be restarted.

The next part loops through the ntp\_servers values we defined in the vars section of the playbook:

{% for item in ntp\_servers %}

server {{ item }} iburst

{% endfor %}

CopyExplain

For each of the values, add a line that contains the word server, the value or {{ item }}, and then iburst.

Now that we know what we have added to the playbook and have an idea of the additional tasks that will be performed, let’s run it using the following command:

$ ansible-playbook -i hosts playbook03.yml

CopyExplain

The following screen just shows the additional tasks and not the full output as we know it will be just marked as ok:



Figure 1.16 – The output of running ansible-playbook03.yml

This time, we have three changed tasks. Running the playbook again shows the following:

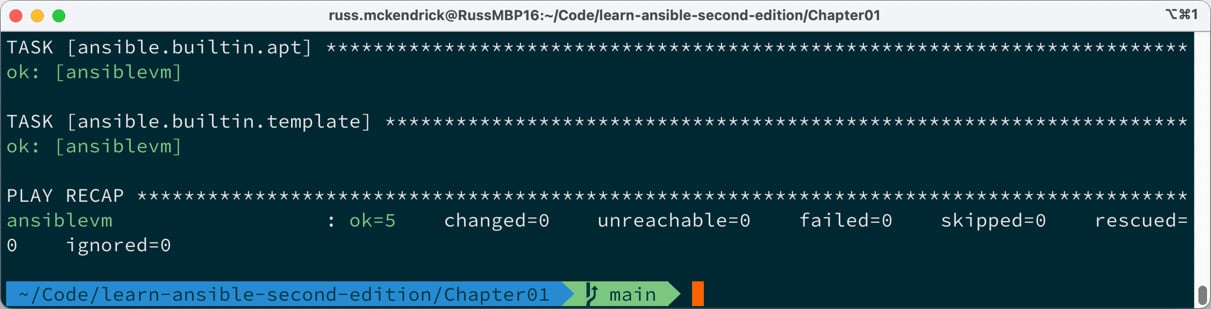


Figure 1.17 – The output of rerunning ansible-playbook03.yml

As expected, there are no changes because we haven’t changed the playbook or anything on the virtual machine, and Ansible is reporting everything as ok. Also, because no changes were detected to the NTP config file, the Handler to restart NTP did not need to be called, and therefore it doesn’t appear in the output.

Before we finish, let’s launch a second virtual machine by running the following command:

$ multipass launch -n ansiblevm2 --cloud-init cloud-init.yaml

CopyExplain

Once the second virtual machine has started, run the following command to get some information on the new virtual machine:

$ multipass info ansiblevm2

CopyExplain

Now that we know the IP address, we can add two new lines to our hosts file. First of all, to define the new host, add the following code (updating it so it uses the correct IP address) underneath where our original host is defined:

ansiblevm2 ansible\_host=192.168.64.8.nip.io

CopyExplain

Then, add ansiblevm2 to the ansible\_hosts group:

[ansible\_hosts]

ansiblevm

ansiblevm2

CopyExplain

Then, rerun the playbook using the following:

$ ansible-playbook -i hosts playbook03.yml

CopyExplain

As you can see, the same commands run, but now we are targeting both virtual machines, the original virtual machine has no changes, and all the changes are applied to the newly deployed host:

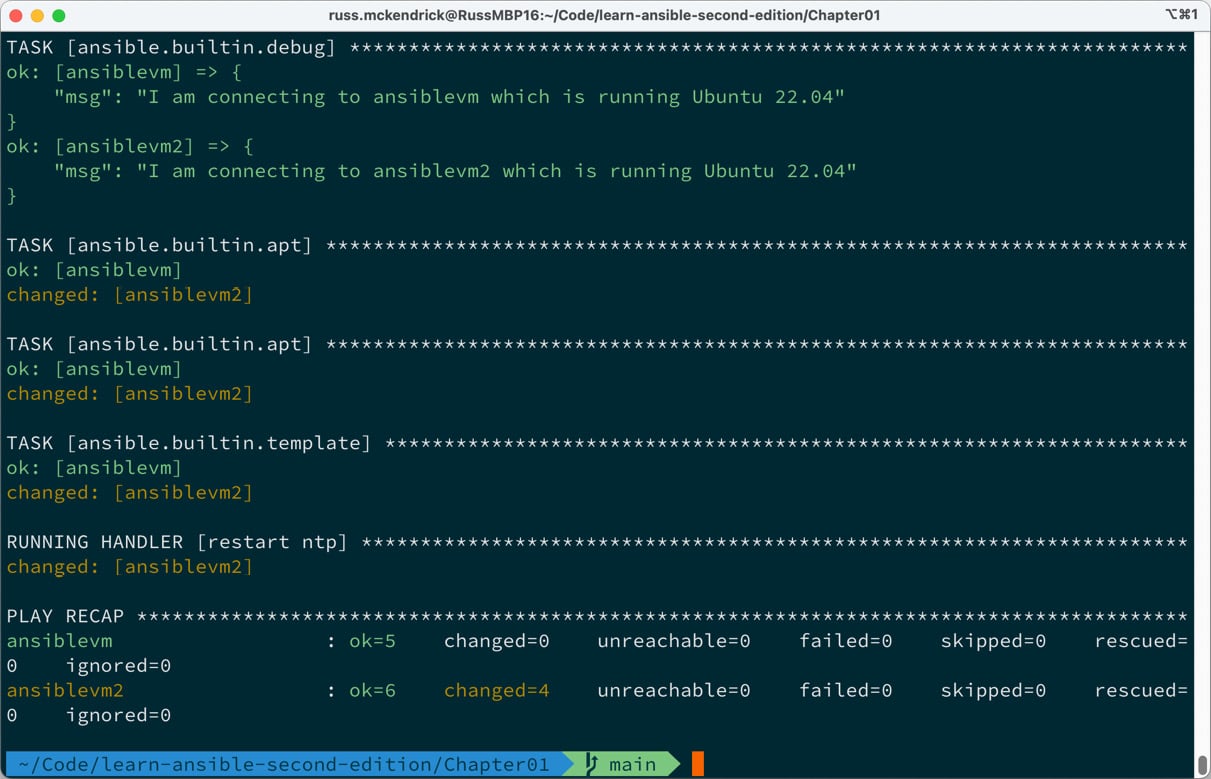


Figure 1.18 – The output of rerunning ansible-playbook03.yml against two virtual machines

If you rerun the command, you will see that everything is now shown as ok as there are no further changes.

Before we move on to the summary, let’s tidy up our two virtual machines and remove them as we won’t need them again. To do this, run the following command:

$ multipass delete --purge ansiblevm ansiblevm2

CopyExplain

As I am sure you would have already guessed, this deletes the virtual machines and then purges the configuration and files.

Summary

In this chapter, we have taken our first steps with Ansible by installing it locally and then, using Vagrant, launching a virtual machine to interact with. We learned about basic host inventory files and used the Ansible command to execute a single task against our virtual machine.

We then looked at playbooks, starting with a basic playbook that returned some information on our target before progressing to a playbook that updates all the installed operating system packages before installing and configuring the NTP service.

By the end of the chapter, we had launched a second virtual machine and quickly brought it up to the same configuration level as our first virtual machine.

In the next chapter, we will look at Ansible Galaxy and discuss how Ansible packages up and maintains its community modules.