Practical Ansible 2

Automate infrastructure, manage configuration, and deploy applications with Ansible 2.9

Packt
www.packt.com

Daniel Oh, James Freeman, and Fabio Alessandro Locati

Practical Ansible 2

Commissioning Editor: Vijin Boricha
Acquisition Editor: Meeta Rajani
Content Development Editor: Ronn Kurien
Senior Editor: Rahul Dsouza
Technical Editor: Sarvesh Jaywant
Copy Editor: Safis Editing
Project Coordinator: Neil Dmello
Proofreader: Safis Editing
Indexer: Rekha Nair
Production Designer: Jyoti Chauhan

First published: June 2020

Production reference: 1050620

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78980-746-2

www.packt.com

Table of Contents

Preface

Welcome to *Practical Ansible 2*, your guide to going from beginner to proficient Ansible automation engineer in a matter of a few chapters. This book will provide you with the knowledge and skills required to perform your very first installation and automation tasks with Ansible, and take you on a journey from simple one-line automation commands that perform single tasks, all the way through to writing your own complex custom code to extend the functionality of Ansible, and even automate cloud and container infrastructures. Throughout the book, practical examples will be given for you to not just read about Ansible automation, but actually try it out for yourself and understand how the code works. You will then be well placed to automate your infrastructure with Ansible in a manner that is scalable, repeatable, and reliable.

Who this book is for

This book is for anyone who has IT tasks they want to automate, from mundane day-to-day housekeeping tasks to complex infrastructure-as-code-based deployments. It is intended to appeal to anyone with prior experience of Linux-based environments who wants to get up to speed quickly with Ansible automation, and to appeal to a wide range of individuals, from system administrators to DevOps engineers to architects looking at overall automation strategy. It will even serve hobbyists well. Basic proficiency in Linux system administration and maintenance tasks is assumed; however, no previous Ansible or automation experience is required.

What this book covers

`Chapter 1`, *Getting Started with Ansible*, provides the steps you need for your very first installation of Ansible, and explains how to get up and running with this powerful automation.

`Chapter 2`, *Understanding the Fundamentals of Ansible*, explores the Ansible framework, gives you a sound understanding of the fundamentals of the Ansible language, and explains how to work with the various command-line tools that it comprises.

`Chapter 3`, *Defining Your Inventory*, gives you details about the Ansible inventory, its purpose, and how to create your own inventories and work with them. It also explores the differences between static and dynamic inventories, and when to leverage each type.

`Chapter 4`, *Playbooks and Roles*, provides you with an in-depth look at creating your own automation code in Ansible in the form of playbooks, and how to enable effective reuse of that code through roles.

`Chapter 5`, *Consuming and Creating Modules*, teaches you about Ansible modules and their purpose, and then provides you with the steps required to write your own module, and even to submit it to the Ansible project for inclusion.

`Chapter 6`, *Consuming and Creating Plugins*, explains the purpose of Ansible plugins, and covers the various types of plugin that Ansible uses. It then explains how to write your own plugins, and explains how to submit your code to the Ansible project.

`Chapter 7`, *Coding Best Practices*, provides an in-depth look at the best practices that you should adhere to while writing Ansible automation code to ensure that your solutions are manageable, easy to maintain, and easy to scale.

`Chapter 8`, *Advanced Ansible Topics*, explores some of the more advanced Ansible options and language directives, which are valuable in scenarios such as performing a roll-out to a highly available cluster. It also explains how to work with jump hosts to automate tasks on secure networks, and how to encrypt your variable data at rest.

`Chapter 9`, *Network Automation with Ansible*, provides a detailed look at the importance of network automation, explains why Ansible is especially well suited to this task, and takes you through practical examples of how to connect to a variety of network devices with Ansible.

`Chapter 10`, *Container and Cloud Management*, explores the manner in which Ansible supports working with both cloud and container platforms, and teaches you how to build containers with Ansible, along with methods to deploy infrastructure as code in a cloud environment using Ansible.

`Chapter 11`, *Troubleshooting and Testing Strategies*, teaches you how to test and debug your Ansible code, and gives you robust strategies to handle errors and unexpected failures both with playbooks and the agentless connections on which Ansible relies.

`Chapter 12`, *Getting Started with Ansible Tower*, provides an introduction to Ansible Tower and its upstream open source counterpart, AWX, demonstrating how this powerful tool provides a valuable complement to Ansible, especially in large, multi-user environments such as enterprises.

To get the most out of this book

All the chapters of this book assume you have access to at least one Linux machine running a relatively recent Linux distribution. All examples in this book were tested on CentOS 7 and Ubuntu Server 18.04, but should work on just about any other mainstream distribution. You will require Ansible 2.9 installed on at least one test machine too – installation steps will be covered in the very first chapter. Later versions of Ansible should also work, though there may be some subtle differences, and you should refer to the release notes and porting guide for newer Ansible versions. The final chapter also takes you through the installation of AWX, but this assumes a Linux server with Ansible installed. Most examples demonstrate automation across more than one host, and if you have more Linux hosts available you will be able to get more out of the examples; however, they can be scaled up or down as you require. Having more hosts is not mandatory, but enables you to get more out of the book.

Software/hardware covered in the book	**OS requirements**
At least one Linux server (virtual machine or physical)	CentOS 7 or Ubuntu Server 18.04, though other mainstream distributions (including newer versions of these operating systems) should work.
Ansible 2.9	As above
AWX release 10.0.0 or later	As above

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Practical-Ansible-2`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/9781789807462_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
html, body, #map {
 height: 100%;
 margin: 0;
 padding: 0
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

Any command-line input or output is written as follows:

```
$ mkdir css
$ cd css
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Section 1: Learning the Fundamentals of Ansible

In this section, we will take a look at the very fundamentals of Ansible. We will start with the process of installing Ansible and then we will get to grips with the fundamentals, including the basics of the language and ad hoc commands. We will then explore Ansible inventories, before looking at writing our very first playbooks and roles to complete multi-stage automation tasks.

This section contains the following chapters:

1
Getting Started with Ansible

Ansible enables you to easily deploy applications and systems consistently and repeatably using native communication protocols such as SSH and WinRM. Perhaps most importantly, Ansible is agentless and so requires nothing to be installed on the managed systems (except for Python, which, these days, is present on most systems). As a result, it enables you to build a simple yet robust automation platform for your environment.

Ansible is simple and straightforward to install and comes packaged for most modern systems. Its architecture is serverless as well as agentless, and so it has a minimal footprint. You can choose to run it from a central server or your own laptop—the choice is entirely yours. You can manage anything from a single host to hundreds of thousands of remote hosts from one Ansible control machine. All remote machines can be (with sufficient playbooks being written) managed by Ansible, and with everything created correctly, you may never have to log in to any of these machines individually again.

In this chapter, we will begin to teach you the practical skills to cover the very fundamentals of Ansible, starting with how to install Ansible on a wide variety of operating systems. We will then look at how to configure Windows hosts to enable them to be managed with Ansible automation, before delving into greater depth on the topic of how Ansible connects to its target hosts. We'll then look at node requirements and how to validate your Ansible installation, before finally looking at how to obtain and run the very latest Ansible source code if you wish to either contribute to its development or gain access to the very latest of features.

In this chapter, we will cover the following topics:

- Installing and configuring Ansible
- Understanding your Ansible installation
- Running from source versus pre-built RPMs

Technical requirements

Ansible has a fairly minimal set of system requirements—as such, you should find that if you have a machine (either a laptop, a server, or a virtual machine) that is capable of running Python, then you will be able to run Ansible on it. Later in this chapter, we will demonstrate the installation methods for Ansible on a variety of operating systems—it is hence left to you to decide which operating systems are right for you.

The one exception to the preceding statement is Microsoft Windows—although there are Python environments available for Windows, there is as yet no native build of Ansible for Windows. Readers running more recent versions of Windows will be able to install Ansible using Windows Subsystem for Linux (henceforth, WSL) and by following the procedures outlined later for their chosen WSL environment (for example, if you install Ubuntu on WSL, you should simply follow the instructions given in this chapter for installing Ansible on Ubuntu).

Installing and configuring Ansible

Ansible is written in Python and, as such, can be run on a wide range of systems. This includes most popular flavors of Linux, FreeBSD, and macOS. The one exception to this is Windows, where though native Python distributions exist, there is as yet no native Ansible build. As a result, your best option at the time of writing is to install Ansible under WSL proceeding as if you were running on a native Linux host.

Once you have established the system on which you wish to run Ansible, the installation process is normally simple and straightforward. In the following sections, we will discuss how to install Ansible on a wide range of different systems, so that most readers should be able to get up and running with Ansible in a matter of minutes.

Installing Ansible on Linux and FreeBSD

The release cycle for Ansible is usually about four months, and during this short release cycle, there are normally many changes, from minor bug fixes to major ones, to new features and even sometimes fundamental changes to the language. The simplest way to not only get up and running with Ansible but to keep yourself up to date is to use the native packages built for your operating system where they are available.

For example, if you wish to run the latest version of Ansible on top of Linux distribution such as CentOS, Fedora, **Red Hat Enterprise Linux** (**RHEL**), Debian, and Ubuntu, I strongly recommend that you use an operating system package manager such as `yum` on Red Hat-based distributions or `apt` on Debian-based ones. In this manner, whenever you update your operating system, you will update Ansible simultaneously.

Of course, it might be that you need to retain a specific version of Ansible for certain purposes—perhaps because your playbooks have been tested with this. In this instance, you would almost certainly choose an alternative installation method, but this is beyond the scope of this book. Also, it is recommended that, where possible, you create and maintain your playbooks in line with documented best practices, which should mean that they survive most Ansible upgrades.

The following are some examples showing how you might install Ansible on several Linux distributions:

- **Installing Ansible on Ubuntu**: To install the latest version of the Ansible control machine on Ubuntu, the `apt` packaging tool makes it easy using the following commands:

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

  If you are running an older version of Ubuntu, you might need to replace `software-properties-common` with `python-software-properties` instead.

- **Installing Ansible on Debian:** You should add the following line into your `/etc/apt/sources.list` file:

  ```
  deb http://ppa.launchpad.net/ansible/ansible/ubuntu trusty main
  ```

  You will note that the word `ubuntu` appears in the preceding line of configuration along with `trusty`, which is an Ubuntu version. Debian builds of Ansible are, at the time of writing, taken from the Ansible repositories for Ubuntu and work without issue. You might need to change the version string in the preceding configuration according to your Debian build, but for most common use cases, the line quoted here will suffice.

Once this is done, you can install Ansible on Debian as follows:

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
93C4A3FD7BB9C367
$ sudo apt-get update
$ sudo apt-get install ansible
```

- **Installing Ansible on Gentoo**: To install the latest version of the Ansible control machine on Gentoo, the `portage` package manager makes it easy with the following commands:

```
$ echo 'app-admin/ansible' >> /etc/portage/package.accept_keywords
$ emerge -av app-admin/ansible
```

- **Installing Ansible on FreeBSD**: To install the latest version of the Ansible control machine on FreeBSD, the PKG manager makes it easy with the following commands:

```
$ sudo pkg install py36-ansible
$ sudo make -C /usr/ports/sysutils/ansible install
```

- **Installing Ansible on Fedora**: To install the latest version of the Ansible control machine on Fedora, the `dnf` package manager makes it easy with the following commands:

```
$ sudo dnf -y install ansible
```

- **Installing Ansible on CentOS**: To install the latest version of the Ansible control machine on CentOS or RHEL, the `yum` package manager makes it easy with the following commands:

```
$ sudo yum install epel-release
$ sudo yum -y install ansible
```

If you execute the preceding commands on RHEL, you have to make sure that the Ansible repository is enabled. If it's not, you need to enable the relevant repository with the following commands:

```
$ sudo subscription-manager repos --enable rhel-7-server-
ansible-2.9-rpms
```

- **Installing Ansible on Arch Linux**: To install the latest version of the Ansible control machine on Arch Linux, the `pacman` package manager makes it easy with the following commands:

  ```
  $ pacman -S ansible
  ```

Once you have installed Ansible on the specific Linux distribution that you use, you can begin to explore. Let's start with a simple example—when you run the `ansible` command, you will see output similar to the following:

```
$ ansible --version
ansible 2.9.6
 config file = /etc/ansible/ansible.cfg
 configured module search path =
[u'/home/jamesf_local/.ansible/plugins/modules',
u'/usr/share/ansible/plugins/modules']
 ansible python module location = /usr/lib/python2.7/dist-packages/ansible
 executable location = /usr/bin/ansible
 python version = 2.7.17 (default, Nov 7 2019, 10:07:09) [GCC 9.2.1
20191008]
```

Those who wish to test the very latest versions of Ansible, fresh from GitHub itself, might be interested in building an RPM package for installing to control machines. This method is, of course, only suitable for Red Hat-based distributions such as Fedora, CentOS, and RHEL. To do this, you will need to clone source code from the GitHub repository and build the RPM package as follows:

```
$ git clone https://github.com/ansible/ansible.git
$ cd ./ansible
$ make rpm
$ sudo rpm -Uvh ./rpm-build/ansible-*.noarch.rpm
```

Now that you have seen how to install Ansible on Linux, we'll take a brief look at how to install Ansible on macOS.

Installing Ansible on macOS

In this section, you will learn how to install Ansible on macOS. The easiest installation method is to use Homebrew, but you could also use the Python package manager. Let's get started by installing Homebrew, which is a fast and convenient package management solution for macOS.

If you don't already have Homebrew installed on macOS, you can easily install it as detailed here:

- **Installing Homebrew**: Normally the two commands shown here are all that is required to install Homebrew on macOS:

  ```
  $ xcode-select --install
  $ ruby -e "$(curl -fsSL
  https://raw.githubusercontent.com/Homebrew/install/master/install)"
  ```

  If you have already installed the Xcode command-line tools for another purpose, you might see the following error message:

  ```
  xcode-select: error: command line tools are already installed, use
  "Software Update" to update
  ```

  You may want to open the App Store on macOS and check whether updates to Xcode are required, but as long as the command-line tools are installed, your Homebrew installation should proceed smoothly.

  If you wish to confirm that your installation of Homebrew was successful, you can run the following command, which will warn you about any potential issues with your install—for example, the following output is warning us that, although Homebrew is installed successfully, it is not in our `PATH` and so we may not be able to run any executables without specifying their absolute path:

  ```
  $ brew doctor
  Please note that these warnings are just used to help the Homebrew
  maintainers
  with debugging if you file an issue. If everything you use Homebrew
  for is
  working fine: please don't worry or file an issue; just ignore
  this. Thanks!

  Warning: Homebrew's sbin was not found in your PATH but you have
  installed
  formulae that put executables in /usr/local/sbin.
  Consider setting the PATH for example like so
   echo 'export PATH="/usr/local/sbin:$PATH"' >> ~/.bash_profile
  ```

- **Installing the Python package manager (pip)**: If you don't wish to use Homebrew to install Ansible, you can instead install `pip` using with the following simple commands:

  ```
  $ sudo easy_install pip
  ```

Also check that your Python version is at least 2.7, as Ansible won't run on anything older (this should be the case with almost all modern installations of macOS):

```
$ python --version
Python 2.7.16
```

You can use either Homebrew or the Python package manager to install the latest version of Ansible on macOS as follows:

- **Installing Ansible via Homebrew**: To install Ansible via Homebrew, run the following command:

```
$ brew install ansible
```

- **Installing Ansible via the Python package manager (pip)**: To install Ansible via `pip`, use the following command:

```
$ sudo pip install ansible
```

You might be interested in running the latest development version of Ansible direct from GitHub, and if so, you can achieve this by running the following command:

```
$ pip install git+https://github.com/ansible/ansible.git@devel
```

Now that you have installed Ansible using your preferred method, you can run the `ansible` command as before, and if all has gone according to plan, you will see output similar to the following:

```
$ ansible --version
ansible 2.9.6
  config file = None
  configured module search path = ['/Users/james/.ansible/plugins/modules',
'/usr/share/ansible/plugins/modules']
  ansible python module location =
/usr/local/Cellar/ansible/2.9.4_1/libexec/lib/python3.8/site-
packages/ansible
  executable location = /usr/local/bin/ansible
  python version = 3.8.1 (default, Dec 27 2019, 18:05:45) [Clang 11.0.0
(clang-1100.0.33.16)]
```

If you are running macOS 10.9, you may experience issues when installing Ansible using `pip`. The following is a workaround that should resolve the issue:

```
$ sudo CFLAGS=-Qunused-arguments CPPFLAGS=-Qunused-arguments pip install
ansible
```

If you want to update your Ansible version, `pip` makes it easy via the following command:

```
$ sudo pip install ansible --upgrade
```

Similarly, you can upgrade it using the `brew` command if that was your install method:

```
$ brew upgrade ansible
```

Now that you have learned the steps to install Ansible on macOS, let's see how to configure a Windows host for automation with Ansible.

Configuring Windows hosts for Ansible

As discussed earlier, there is no direct installation method for Ansible on Windows—simply, it is recommended that, where available, you install WSL and install Ansible as if you were running Linux natively, using the processes outlined earlier in this chapter.

Despite this limitation, however, Ansible is not limited to managing just Linux- and BSD-based systems—it is capable of the agentless management of Windows hosts using the native WinRM protocol, with modules and raw commands making use of PowerShell, which is available in every modern Windows installation. In this section, you will learn how to configure Windows to enable task automation with Ansible.

Let's look at what Ansible is capable of when automating Windows hosts:

- Gather facts about remote hosts.
- Install and uninstall Windows features.
- Manage and query Windows services.
- Manage user accounts and a list of users.
- Manage packages using Chocolatey (a software repository and accompanying management tool for Windows).
- Perform Windows updates.
- Fetch multiple files from a remote machine to the Windows host.
- Execute raw PowerShell commands and scripts on target hosts.

Ansible allows you to automate tasks on Windows machines by connecting with either a local user or a domain user. You can run actions as an administrator using the Windows `runas` support, just as with the `sudo` command on Linux distributions.

Also, as Ansible is open source software, it is easy to extend its functionality by creating your own modules in PowerShell or even sending raw PowerShell commands. For example, an InfoSec team could manage filesystem ACLs, configure Windows Firewall, and manage hostnames and domain membership with ease, using a mix of native Ansible modules and, where necessary, raw commands.

The Windows host must meet the following requirements for the Ansible control machine to communicate with it:

- Ansible attempts to support all Windows versions that are under either current or extended support from Microsoft, including desktop platforms such as Windows 7, 8.1, and 10, along with server operating systems including Windows Server 2008 (and R2), 2012 (and R2), 2016, and 2019.
- You will also need to install PowerShell 3.0 or later and at least .NET 4.0 on your Windows host.
- You will need to create and activate a WinRM listener, which is described in detail later. For security reasons, this is not enabled by default.

Let's look in more detail at how to prepare a Windows host to be automated by Ansible:

1. With regard to prerequisites, you have to make sure PowerShell 3.0 and .NET Framework 4.0 are installed on Windows machines. If you're still using the older version of PowerShell or .NET Framework, you need to upgrade them. You are free to perform this manually, or the following PowerShell script can handle it automatically for you:

```
$url =
"https://raw.githubusercontent.com/jborean93/ansible-windows/master
/scripts/Upgrade-PowerShell.ps1"
$file = "$env:temp\Upgrade-PowerShell.ps1" (New-Object -TypeName
System.Net.WebClient).DownloadFile($url, $file)

Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Force &$file -
Verbose Set-ExecutionPolicy -ExecutionPolicy Restricted -Force
```

   This script works by examining the programs that need to be installed (such as .NET Framework 4.5.2) and the required PowerShell version, rebooting if required, and setting the username and password parameters. The script will automatically restart and log on at reboot so that no more action is required and the script will continue until the PowerShell version matches the target version.

If the username and password parameters aren't set, the script will ask the user to reboot and log in manually if necessary, and the next time the user logs in, the script will continue at the point where it was interrupted. The process continues until the host meets the requirements for Ansible automation.

2. When PowerShell has been upgraded to at least version 3.0, the next step will be to configure the WinRM service so that Ansible can connect to it. WinRM service configuration defines how Ansible can interface with the Windows hosts, including the listener port and protocol.

If you have never set up a WinRM listener before, you have three options to do this:

- Firstly, you can use `winrm quickconfig` for HTTP and `winrm quickconfig -transport:https` for HTTPS. This is the simplest method to use when you need to run outside of the domain environment and just create a simple listener. This process has the advantage of opening the required port in the Windows firewall and automatically starting the WinRM service.
- If you are running in a domain environment, I strongly recommend using **Group Policy Objects** (**GPOs**) because if the host is the domain member, then the configuration is done automatically without user input. There are many documented procedures for doing this available, and as this is a very Windows domain-centric task, it is beyond the scope of this book.
- Finally, you can create a listener with a specific configuration by running the following PowerShell commands:

```
$selector_set = @{
    Address = "*"
    Transport = "HTTPS"
}
$value_set = @{
    CertificateThumbprint =
"E6CDAA82EEAF2ECE8546E05DB7F3E01AA47D76CE"
}

New-WSManInstance -ResourceURI "winrm/config/Listener" -SelectorSet
$selector_set -ValueSet $value_set
```

The preceding `CertificateThumbprint` should match the thumbprint of a valid SSL certificate that you previously created or imported into the Windows Certificate Store.

If you are running in PowerShell v3.0, you might face an issue with the WinRM service that limits the amount of memory available. This is a known bug and a hotfix is available to resolve it. An example process (written in PowerShell) to apply this hotfix is given here:

```
$url =
"https://raw.githubusercontent.com/jborean93/ansible-windows/master/scripts
/Install-WMF3Hotfix.ps1"
$file = "$env:temp\Install-WMF3Hotfix.ps1"

(New-Object -TypeName System.Net.WebClient).DownloadFile($url, $file)
powershell.exe -ExecutionPolicy ByPass -File $file -Verbose
```

Configuring the WinRM listeners can be a complex task, so it is important to be able to check the results of your configuration process. The following command (which can be run from Command Prompt) will display the current WinRM listener configuration:

```
winrm enumerate winrm/config/Listener
```

If all goes well, you should have output similar to this:

```
Listener
    Address = *
    Transport = HTTP
    Port = 5985
    Hostname
    Enabled = true
    URLPrefix = wsman
    CertificateThumbprint
    ListeningOn = 10.0.2.15, 127.0.0.1, 192.168.56.155, ::1,
fe80::5efe:10.0.2.15%6, fe80::5efe:192.168.56.155%8, fe80::
ffff:ffff:fffe%2, fe80::203d:7d97:c2ed:ec78%3, fe80::e8ea:d765:2c69:7756%7

Listener
    Address = *
    Transport = HTTPS
    Port = 5986
    Hostname = SERVER2016
    Enabled = true
    URLPrefix = wsman
    CertificateThumbprint = E6CDAA82EEAF2ECE8546E05DB7F3E01AA47D76CE
    ListeningOn = 10.0.2.15, 127.0.0.1, 192.168.56.155, ::1,
fe80::5efe:10.0.2.15%6, fe80::5efe:192.168.56.155%8, fe80::
ffff:ffff:fffe%2, fe80::203d:7d97:c2ed:ec78%3, fe80::e8ea:d765:2c69:7756%7
```

According to the preceding output, two listeners are active—one to listen on port `5985` over HTTP and the other to listen on port `5986` over HTTPS providing greater security. By way of additional explanation, the following parameters are also displayed in the preceding output:

- `Transport`: This should be set to either HTTPS or HTTPS, though it is strongly recommended that you use the HTTPS listener to ensure your automation commands are not subject to snooping or manipulation.
- `Port`: This is the port on which the listener operates, by default `5985` for HTTP or `5986` for HTTPS.
- `URLPrefix`: This is the URL prefix to communicate with, by default, `wsman`. If you change it, you must set the `ansible_winrm_path` host on your Ansible control host to the same value.
- `CertificateThumbprint`: If running on an HTTPS listener, this is the certificate thumbprint of the Windows Certificate Store used by the connection.

If you need to debug any connection issues after setting up your WinRM listener, you may find the following commands valuable as they perform WinRM-based connections between Windows hosts without Ansible—hence, you can use them to distinguish whether an issue you might be experiencing is related to your Ansible host or whether there is an issue with the WinRM listener itself:

```
# test out HTTP
winrs -r:http://<server address>:5985/wsman -u:Username -p:Password
ipconfig

# test out HTTPS (will fail if the cert is not verifiable)
winrs -r:https://<server address>:5986/wsman -u:Username -p:Password -ssl
ipconfig

# test out HTTPS, ignoring certificate verification
$username = "Username"
$password = ConvertTo-SecureString -String "Password" -AsPlainText -Force
$cred = New-Object -TypeName System.Management.Automation.PSCredential -
ArgumentList $username, $password

$session_option = New-PSSessionOption -SkipCACheck -SkipCNCheck -
SkipRevocationCheck
Invoke-Command -ComputerName server -UseSSL -ScriptBlock { ipconfig } -
Credential $cred -SessionOption $session_option
```

If one of the preceding commands fails, you should investigate your WinRM listener setup before attempting to set up or configure your Ansible control host.

At this stage, Windows should be ready to receive communication from Ansible over WinRM. To complete this process, you will need to also perform some additional configuration on your Ansible control host. First of all, you will need to install the `winrm` Python module, which, depending on your control hosts' configuration, may or may not have been installed before. The installation method will vary from one operating system to another, but it can generally be installed on most platforms with `pip` as follows:

```
$ pip install winrm
```

Once this is complete, you will need to define some additional inventory variables for your Windows hosts—don't worry too much about inventories for now as we will cover these later in this book. The following example is just for reference:

```
[windows]
192.168.1.52

[windows:vars]
ansible_user=administrator
ansible_password=password
ansible_connection=winrm
ansible_winrm_server_cert_validation=ignore
```

Finally, you should be able to run the Ansible `ping` module to perform an end-to-end connectivity test with a command like the following (adjust for your inventory):

```
$ ansible -i inventory -m ping windows
192.168.1.52 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Now that you have learned the necessary steps to configure Windows hosts for Ansible, let's see how to connect multiple hosts via Ansible in the next section.

Understanding your Ansible installation

By this stage in this chapter, regardless of your operating system choice for your Ansible control machine, you should have a working installation of Ansible with which to begin exploring the world of automation. In this section, we will carry out a practical exploration of the fundamentals of Ansible to help you to understand how to work with it. Once you have mastered these basic skills, you will then have the knowledge required to get the most out of the remainder of this book. Let's get started with an overview of how Ansible connects to non-Windows hosts.

Understanding how Ansible connects to hosts

With the exception of Windows hosts (as discussed at the end of the previous section), Ansible uses the SSH protocol to communicate with hosts. The reasons for this choice in the Ansible design are many, not least that just about every Linux/FreeBSD/macOS host has it built in, as do many network devices such as switches and routers. This SSH service is normally integrated with the operating system authentication stack, enabling you to take advantage of things such as Kerberos to improve authentication security. Also, features of OpenSSH such as `ControlPersist` are used to increase the performance of the automation tasks and SSH jump hosts for network isolation and security.

TIP

`ControlPersist` is enabled by default on most modern Linux distributions as part of the OpenSSH server installation. However, on some older operating systems such as Red Hat Enterprise Linux 6 (and CentOS 6), it is not supported, and so you will not be able to use it. Ansible automation is still perfectly possible, but longer playbooks might run slower.

Ansible makes use of the same authentication methods that you will already be familiar with, and SSH keys are normally the easiest way to proceed as they remove the need for users to input the authentication password every time a playbook is run. However, this is by no means mandatory, and Ansible supports password authentication through the use of the `--ask-pass` switch. If you are connecting to an unprivileged account on the hosts, and need to perform the Ansible equivalent of running commands under `sudo`, you can also add `--ask-become-pass` when you run your playbooks to allow this to be specified at runtime as well.

TIP

The goal of automation is to be able to run tasks securely but with the minimum of user intervention. As a result, it is highly recommended that you use SSH keys for authentication, and if you have several keys to manage, then be sure to make use of `ssh-agent`.

Every Ansible task, whether it is run singly or as part of a complex playbook, is run against an inventory. An inventory is, quite simply, a list of the hosts that you wish to run the automation commands against. Ansible supports a wide range of inventory formats, including the use of dynamic inventories, which can populate themselves automatically from an orchestration provider (for example, you can generate an Ansible inventory dynamically from your Amazon EC2 instances, meaning you don't have to keep up with all of the changes in your cloud infrastructure).

Dynamic inventory plugins have been written for most major cloud providers (for example, Amazon EC2, Google Cloud Platform, and Microsoft Azure), as well as on-premises systems such as OpenShift and OpenStack. There are even plugins for Docker. The beauty of open source software is that, for most of the major use cases you can dream of, someone has already contributed the code and so you don't need to figure it out or write it for yourself.

TIP

Ansible's agentless architecture and the fact that it doesn't rely on SSL means that you don't need to worry about DNS not being set up or even time skew problems as a result of NTP not working—these can, in fact, be tasks performed by an Ansible playbook! Ansible really was designed to get your infrastructure running from a virtually bare operating system image.

For now, let's focus on the INI formatted inventory. An example is shown here with four servers, each split into two groups. Ansible commands and playbooks can be run against an entire inventory (that is, all four servers), one or more groups (for example, `webservers`), or even down to a single server:

```
[webservers]
web1.example.com
web2.example.com

[apservers]
ap1.example.com
ap2.example.com
```

Let's use this inventory file along with the Ansible `ping` module, which is used to test whether Ansible can successfully perform automation tasks on the inventory host in question. The following example assumes you have installed the inventory in the default location, which is normally `/etc/ansible/hosts`. When you run the following `ansible` command, you see a similar output to this:

```
$ ansible webservers -m ping
web1.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
web2.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
$
```

Notice that the `ping` module was only run on the two hosts in the `webservers` group and not the entire inventory—this was by virtue of us specifying this in the command-line parameters.

The `ping` module is one of many thousands of modules for Ansible, all of which perform a given set of tasks (from copying files between hosts, to text substitution, to complex network device configuration). Again, as Ansible is open source software, there is a veritable army of coders out there who are writing and contributing modules, which means if you can dream of a task, there's probably already an Ansible module for it. Even in the instance that no module exists, Ansible supports sending raw shell commands (or PowerShell commands for Windows hosts) and so even in this instance, you can complete your desired tasks without having to move away from Ansible.

As long as the Ansible control host can communicate with the hosts in your inventory, you can automate your tasks. However, it is worth giving some consideration to where you place your control host. For example, if you are working exclusively with a set of Amazon EC2 machines, it arguably would make more sense for your Ansible control machine to be an EC2 instance—in this way, you are not sending all of your automation commands over the internet. It also means that you don't need to expose the SSH port of your EC2 hosts to the internet, hence keeping them more secure.

We have so far covered a brief explanation of how Ansible communicates with its target hosts, including what inventories are and the importance of SSH communication to all except Windows hosts. In the next section, we will build on this by looking in greater detail at how to verify your Ansible installation.

Verifying the Ansible installation

In this section, you will learn how you can verify your Ansible installation with simple ad hoc commands.

As discussed previously, Ansible can authenticate with your target hosts several ways. In this section, we will assume you want to make use of SSH keys, and that you have already generated your public and private key pair and applied your public key to all of your target hosts that you will be automating tasks on.

The `ssh-copy-id` utility is incredibly useful for distributing your public SSH key to your target hosts before you proceed any further. An example command might be `ssh-copy-id -i ~/.ssh/id_rsa ansibleuser@web1.example.com`.

To ensure Ansible can authenticate with your private key, you could make use of `ssh-agent`—the commands show a simple example of how to start `ssh-agent` and add your private key to it. Naturally, you should replace the path with that to your own private key:

```
$ ssh-agent bash
$ ssh-add ~/.ssh/id_rsa
```

As we discussed in the previous section, we must also define an inventory for Ansible to run against. Another simple example is shown here:

```
[frontends]
frt01.example.com
frt02.example.com
```

The `ansible` command that we used in the previous section has two important switches that you will almost always use: `-m <MODULE_NAME>` to run a module on the hosts from your inventory that you specify and, optionally, the module arguments passed using the `-a OPT_ARGS` switch. Commands run using the `ansible` binary are known as ad hoc commands.

Following are three simple examples that demonstrate ad hoc commands—they are also valuable for verifying both the installation of Ansible on your control machine and the configuration of your target hosts, and they will return an error if there is an issue with any part of the configuration:

- **Ping hosts**: You can perform an Ansible "ping" on your inventory hosts using the following command:

  ```
  $ ansible frontends -i hosts -m ping
  ```

- **Display gathered facts**: You can display gathered facts about your inventory hosts using the following command:

  ```
  $ ansible frontends -i hosts -m setup | less
  ```

- **Filter gathered facts**: You can filter gathered facts using the following command:

  ```
  $ ansible frontends -i hosts -m setup -a
  "filter=ansible_distribution*"
  ```

For every ad hoc command you run, you will get a response in JSON format—the following example output results from running the `ping` module successfully:

```
$ ansible frontends -m ping
frontend01.example.com | SUCCESS => {
    "changed": false,
```

```
        "ping": "pong"
    }
    frontend02.example.com | SUCCESS => {
        "changed": false,
        "ping": "pong"
    }
```

Ansible can also gather and return "facts" about your target hosts—facts are all manner of useful information about your hosts, from CPU and memory configuration to network parameters, to disk geometry. These facts are intended to enable you to write intelligent playbooks that perform conditional actions—for example, you might only want to install a given software package on hosts with more than 4 GB of RAM or perhaps perform a specific configuration only on macOS hosts. The following is an example of the filtered facts from a macOS-based host:

```
$ ansible frontend01.example.com -m setup -a "filter=ansible_distribution*"
frontend01.example.com | SUCCESS => {
 ansible_facts": {
 "ansible_distribution": "macOS",
 "ansible_distribution_major_version": "10",
 "ansible_distribution_release": "18.5.0",
 "ansible_distribution_version": "10.14.4"
 },
 "changed": false
```

Ad hoc commands are incredibly powerful, both for verifying your Ansible installation and for learning Ansible and how to work with modules as you don't need to write a whole playbook—you can just run a module with an ad hoc command and learn how it responds. Here are some more ad hoc examples for you to consider:

- Copy a file from the Ansible control host to all hosts in the `frontends` group with the following command:

```
$ ansible frontends -m copy -a "src=/etc/yum.conf
dest=/tmp/yum.conf"
```

- Create a new directory on all hosts in the `frontends` inventory group, and create it with specific ownership and permissions:

```
$ ansible frontends -m file -a "dest=/path/user1/new mode=777
owner=user1 group=user1 state=directory"
```

- Delete a specific directory from all hosts in the `frontends` group with the following command:

```
$ ansible frontends -m file -a "dest=/path/user1/new state=absent"
```

- Install the `httpd` package with `yum` if it is not already present—if it is present, do not update it. Again, this applies to all hosts in the `frontends` inventory group:

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

- The following command is similar to the previous one, except that changing `state=present` to `state=latest` causes Ansible to install the (latest version of the) package if it is not present, and update it to the latest version if it is present:

```
$ ansible frontends -m yum -a "name=demo-tomcat-1 state=latest"
```

- Display all facts about all the hosts in your inventory (warning—this will produce a lot of JSON!):

```
$ ansible all -m setup
```

Now that you have learned more about verifying your Ansible installation and about how to run ad hoc commands, let's proceed to look in a bit more detail at the requirements of the nodes that are to be managed by Ansible.

Managed node requirements

So far, we have focused almost exclusively on the requirements for the Ansible control host and have assumed that (except for the distribution of the SSH keys) the target hosts will just work. This, of course, is not always the case, and for example, while a modern installation of Linux installed from an ISO will often just work, cloud operating system images are often stripped down to keep them small, and so might lack important packages such as Python, without which Ansible cannot operate.

If your target hosts are lacking Python, it is usually easy to install it through your operating system's package management system. Ansible requires you to install either Python version 2.7 or 3.5 (and above) on both the Ansible control machine (as we covered earlier in this chapter) and on every managed node. Again, the exception here is Windows, which relies on PowerShell instead.

If you are working with operating system images that lack Python, the following commands provide a quick guide to getting Python installed:

- To install Python using `yum` (on older releases of Fedora and CentOS/RHEL 7 and below), use the following:

  ```
  $ sudo yum -y install python
  ```

- On RHEL and CentOS version 8 and newer versions of Fedora, you would use the `dnf` package manager instead:

  ```
  $ sudo dnf install python
  ```

  You might also elect to install a specific version to suit your needs, as in this example:

  ```
  $ sudo dnf install python37
  ```

- On Debian and Ubuntu systems, you would use the `apt` package manager to install Python, again specifying a version if required (the example given here is to install Python 3.6 and would work on Ubuntu 18.04):

  ```
  $ sudo apt-get update
  $ sudo apt-get install python3.6
  ```

The `ping` module we discussed earlier in this chapter for Ansible not only checks connectivity and authentication with your managed hosts, but it uses the managed hosts' Python environment to perform some basic host checks. As a result, it is a fantastic end-to-end test to give you confidence that your managed hosts are configured correctly as hosts, with the connectivity and authentication set up perfectly, but where Python is missing, it would return a `failed` result.

Of course, a perfect question at this stage would be: how can Ansible help if you roll out 100 cloud servers using a stripped-down base image without Python? Does that mean you have to manually go through all 100 nodes and install Python by hand before you can start automating?

Thankfully, Ansible has you covered even in this case, thanks to the `raw` module. This module is used to send raw shell commands to the managed nodes—and it works both with SSH-managed hosts and Windows PowerShell-managed hosts. As a result, you can use Ansible to install Python on a whole set of systems from which it is missing, or even run an entire shell script to bootstrap a managed node. Most importantly, the raw module is one of very few that does not require Python to be installed on the managed node, so it is perfect for our use case where we must roll out Python to enable further automation.

The following are some examples of tasks in an Ansible playbook that you might use to bootstrap a managed node and prepare it for Ansible management:

```
- name: Bootstrap a host without python2 installed
  raw: dnf install -y python2 python2-dnf libselinux-python

- name: Run a command that uses non-posix shell-isms (in this example
/bin/sh doesn't handle redirection and wildcards together but bash does)
  raw: cat < /tmp/*txt
  args:
    executable: /bin/bash

- name: safely use templated variables. Always use quote filter to avoid
injection issues.
  raw: "{{package_mgr|quote}} {{pkg_flags|quote}} install {{python|quote}}"
```

We have now covered the basics of setting up Ansible both on the control host and on the managed nodes, and we have given you a brief primer on configuring your first connections. Before we wrap up this chapter, we will look in more detail at how you might run the latest development version of Ansible, direct from GitHub.

Running from source versus pre-built RPMs

Ansible is always rapidly evolving, and there may be times, either for early access to a new feature (or module) or as part of your own development efforts, that you wish to run the latest, bleeding-edge version of Ansible from GitHub. In this section, we will look at how you can quickly get up and running with the source code. The method outlined in this chapter has the advantage that, unlike package-manager-based installs that must be performed as root, the end result is a working installation of Ansible without the need for any root privileges.

Let's get started by checking out the very latest version of the source code from GitHub:

1. You must clone the sources from the `git` repository first, and then change to the directory containing the checked-out code:

   ```
   $ git clone https://github.com/ansible/ansible.git --recursive
   $ cd ./ansible
   ```

2. Before you can proceed with any development work, or indeed to run Ansible from the source code, you must set up your shell environment. Several scripts are provided for just that purpose, each being suitable for different shell environments. For example, if you are running the venerable Bash shell, you would set up your environment with the following command:

   ```
   $ source ./hacking/env-setup
   ```

   Conversely, if you are running the Fish shell, you would set up your environment as follows:

   ```
   $ source ./hacking/env-setup.fish
   ```

3. Once you have set up your environment, you must install the `pip` Python package manager, and then use this to install all of the required Python packages (note: you can skip the first command if you already have `pip` on your system):

   ```
   $ sudo easy_install pip
   $ sudo pip install -r ./requirements.txt
   ```

   Note that, when you have run the `env-setup` script, you'll be running from your source code checkout, and the default inventory file will be `/etc/ansible/hosts`. You can optionally specify an inventory file other than `/etc/ansible/hosts`.

4. When you run the `env-setup` script, Ansible runs from the source code checkout, and the default inventory file is `/etc/ansible/hosts`; however, you can optionally specify an inventory file wherever you want on your machine (see *Working with Inventory*, `https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#inventory`, for more details). The following command provides an example of how you might do this, but obviously, your filename and contents are almost certainly going to vary:

   ```
   $ echo "ap1.example.com" > ~/my_ansible_inventory
   $ export ANSIBLE_INVENTORY=~/my_ansible_inventory
   ```

`ANSIBLE_INVENTORY` applies to Ansible version 1.9 and above and replaces the deprecated `ANSIBLE_HOSTS` environment variable.

Once you have completed these steps, you can run Ansible exactly as we have discussed throughout this chapter, with the exception that you must specify the absolute path to it. For example, if you set up your inventory as in the preceding code and clone the Ansible source into your home directory, you could run the ad hoc `ping` command that we are now familiar with, as follows:

```
$ ~/ansible/bin/ansible all -m ping
ap1.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Of course, the Ansible source tree is constantly changing and it is unlikely you would just want to stick with the copy you cloned. When the time comes to update it, you don't need to clone a new copy; you can simply update your existing working copy using the following commands (again, assuming that you initially cloned the source tree into your home directory):

```
$ git pull --rebase
$ git submodule update --init --recursive
```

That concludes our introduction to setting up both your Ansible control machine and managed nodes. It is hoped that the knowledge you have gained in this chapter will help you to get your own Ansible installation up and running and set the groundwork for the rest of this book.

Summary

Ansible is a powerful and versatile yet simple automation tool, of which the key benefits are its agentless architecture and its simple installation process. Ansible was designed to get you from zero to automation rapidly and with minimal effort, and we have demonstrated the simplicity with which you can get up and running with Ansible in this chapter.

In this chapter, you learned the basics of setting up Ansible—how to install it to control other hosts and the requirements for nodes being managed by Ansible. You learned about the fundamentals required to set up SSH and WinRM for Ansible automation, as well as how to bootstrap managed nodes to ensure they are suitable for Ansible automation. You also learned about ad hoc commands and their benefits. Finally, you learned how to run the latest version of the code directly from GitHub, which both enables you to contribute directly to the development of Ansible and gives you access to the very latest features should you wish to make use of them on your infrastructure.

In the next chapter, we will learn Ansible language fundamentals to enable you to write your first playbooks and to help you to create templated configurations and start to build up complex automation workflows.

Questions

1. On which operating systems can you install Ansible? (Multiple correct answers)

   A) Ubuntu

   B) Fedora

   C) Windows 2019 server

   D) HP-UX

   E) Mainframe

2. Which protocol does Ansible use to connect the remote machine for running tasks?

   A) HTTP

   B) HTTPS

   C) SSH

   D) TCP

   E) UDP

3. To execute a specific module in the Ansible ad hoc command line, you need to use the `-m` option.

   A) True

   B) False

Further reading

- For any questions about installation via Ansible Mailing Liston Google Groups, see the following:

  ```
  https://groups.google.com/forum/#!forum/ansible-project
  ```

- How to install the latest version of `pip` can be found here:

  ```
  https://pip.pypa.io/en/stable/installing/#installation
  ```

- Specific Windows modules using PowerShell can be found here:

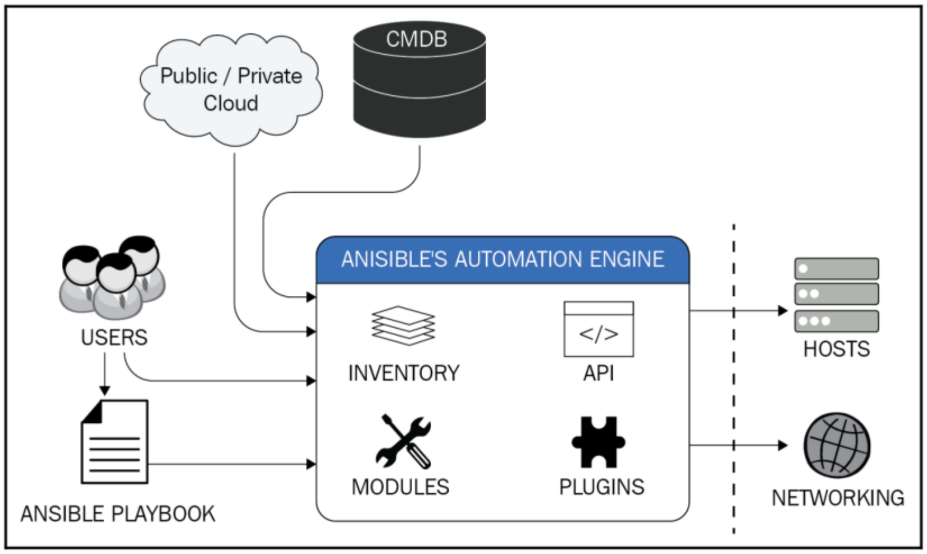
  ```
  https://github.com/ansible/ansible-modules-core/tree/devel/windows
  ```

- If you have a GitHub account and want to follow the GitHub project, you can keep tracking issues, bugs, and ideas for Ansible:

  ```
  https://github.com/ansible/ansible
  ```

2
Understanding the Fundamentals of Ansible

Ansible is, at its heart, a simple framework that pushes a small program called an **Ansible module** to target nodes. Modules are at the heart of Ansible and are responsible for performing all of the automation's hard work. The Ansible framework goes beyond this, however, and also includes plugins and dynamic inventory management, as well as tying all of this together with playbooks to automate infrastructure provisioning, configuration management, application deployment, network automation, and much more, as shown:



Ansible only needs to be installed on the management node; from there, it distributes the required modules over the network's transport layer (usually SSH or WinRM) to perform tasks and deletes them once the tasks are complete. In this way, Ansible retains its agentless architecture and does not clutter up your target nodes with code that might be required for a one-off automation task.

In this chapter, you will learn more about the composition of the Ansible framework and its various components, as well as how to use them together in playbooks written in YAML syntax. So, you will learn how to create automation code for your IT operations tasks and learn how to apply this using both ad hoc tasks and more complex playbooks. Finally, you will learn how Jinja2 templating allows you to repeatably build dynamic configuration files using variables and dynamic expressions.

In this chapter, we will cover the following topics:

- Getting familiar with the Ansible framework
- Exploring the configuration file
- Command-line arguments
- Defining variables
- Understanding Jinja2 filters

Technical requirements

This chapter assumes that you have successfully installed the latest version of Ansible (2.9, at the time of writing) onto a Linux node, as discussed in `Chapter 1`, *Getting Started with Ansible*. It also assumes that you have at least one other Linux host to test automation code on; the more hosts you have available, the more you will be able to develop the examples in this chapter and learn about Ansible. SSH communication between the Linux hosts is assumed, as is a working knowledge of them.

The code bundle for this chapter is available at `https://github.com/PacktPublishing/Ansible-2-Cookbook/tree/master/Chapter%202`.

Getting familiar with the Ansible framework

In this section, you will understand how the Ansible framework fits into IT operations automation. We will explain how to start Ansible for the first time. Once you understand this framework, you will be ready to start learning more advanced concepts, such as creating and running playbooks with your own inventory.

In order to run Ansible's ad hoc commands via an SSH connection from your Ansible control machine to multiple remote hosts, you need to ensure you have the latest Ansible version installed on the control host. Use the following command to confirm the latest Ansible version:

```
$ ansible --version
ansible 2.9.6
 config file = /etc/ansible/ansible.cfg
 configured module search path =
[u'/home/jamesf_local/.ansible/plugins/modules',
u'/usr/share/ansible/plugins/modules']
 ansible python module location = /usr/lib/python2.7/dist-packages/ansible
 executable location = /usr/bin/ansible
 python version = 2.7.17 (default, Nov 7 2019, 10:07:09) [GCC 9.2.1
20191008]
```

You also need to ensure SSH connectivity with each remote host that you will define in the inventory. You can use a simple, manual SSH connection on each of your remote hosts to test the connectivity, as Ansible will make use of SSH during all remote Linux-based automation tasks:

```
$ ssh <username>@frontend.example.com
The authenticity of host 'frontend.example.com (192.168.1.52)' can't be
established.
ED25519 key fingerprint is SHA256:hU+saFERGFDERW453tasdFPAkpVws.
Are you sure you want to continue connecting (yes/no)? yes
password:<Input_Your_Password>
```

In this section, we will walk you through how Ansible works, starting with some simple connectivity testing. You can learn how the Ansible framework accesses multiple host machines to execute your tasks by following this simple procedure:

1. Create or edit your default inventory file, `/etc/ansible/hosts` (you can also specify the path with your own inventory file by passing options such as `--inventory=/path/inventory_file`). Add some example hosts to your inventory—these must be the IP addresses or hostnames of real machines for Ansible to test against. The following are examples from my network, but you need to substitute these for your own devices. Add one hostname (or IP address) per line:

   ```
   frontend.example.com
   backend1.example.com
   backend2.example.com
   ```

All hosts should be specified with a resolvable address—that is, a **Fully Qualified Domain Name** (**FQDN**)—if your hosts have DNS entries (or are in `/etc/hosts` on your Ansible control node). This can be IP addresses if you do not have DNS or host entries set up. Whatever format you choose for your inventory addresses, you should be able to successfully ping each host. See the following output as an example:

```
$ ping frontend.example.com
PING frontend.example.com (192.168.1.52): 56 data bytes
64 bytes from 192.168.1.52: icmp_seq=0 ttl=64 time=0.040 ms
64 bytes from 192.168.1.52: icmp_seq=1 ttl=64 time=0.115 ms
64 bytes from 192.168.1.52: icmp_seq=2 ttl=64 time=0.097 ms
64 bytes from 192.168.1.52: icmp_seq=3 ttl=64 time=0.130 ms
```

2. To make the automation process seamless, we'll generate an SSH authentication key pair so that we don't have to type in a password every time we want to run a playbook. If you do not already have an SSH key pair, you can generate one using the following command:

```
$ ssh-keygen
```

When you run the `ssh-keygen` tool, you will see an output similar to the following. Note that you should leave the `passphrase` variable blank when prompted; otherwise, you will need to enter a passphrase every time you want to run an Ansible task, which removes the convenience of authenticating with SSH keys:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/doh/.ssh/id_rsa):
<Enter>
Enter passphrase (empty for no passphrase): <Press Enter>
Enter same passphrase again: <Press Enter>
Your identification has been saved in /Users/doh/.ssh/id_rsa.
Your public key has been saved in /Users/doh/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:1IF0KMMTVAMEQF62kTwcG59okGZLiMmi4Ae/BGBT+24 doh@danieloh.com
The key's randomart image is:
+---[RSA 2048]----+
|=*=*BB==+oo |
|B=*+*B=.o+ . |
|=+=o=.o+. . |
|...=. . |
| o .. S |
| .. |
| E |
| . |
| |
+----[SHA256]-----+
```

3. Although there are conditions that your SSH keys are automatically picked up with, it is recommended that you make use of `ssh-agent` as this allows you to load multiple keys to authenticate against a variety of targets. This will be very useful to you in the future, even if it isn't right now. Start `ssh-agent` and add your new authentication key, as follows (note that you will need to do this for every shell that you open):

```
$ ssh-agent bash
$ ssh-add ~/.ssh/id_rsa
```

4. Before you can perform key-based authentication with your target hosts, you need to apply the public key from the key pair you just generated to each host. You can copy the key to each host, in turn, using the following command:

```
$  ssh-copy-id -i ~/.ssh/id_rsa.pub frontend.example.com
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed:
"~/.ssh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new
key(s), to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if
you are prompted now it is to install the new keys
doh@frontend.example.com's password:

Number of key(s) added: 1

Now try logging into the machine, with: "ssh
'frontend.example.com'"
and check to make sure that only the key(s) you wanted were added.
```

5. With this complete, you should now be able to perform an Ansible `ping` command on the hosts you put in your inventory file. You will find that you are not prompted for a password at any point as the SSH connections to all the hosts in your inventory are authenticated with your SSH key pair. So, you should see an output similar to the following:

```
$ ansible all -i hosts -m ping
frontend.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
backend1.example.com | SUCCESS => {
        "changed": false,
        "ping": "pong"
    }
    backend2.example.com | SUCCESS => {
        "changed": false,
        "ping": "pong"
    }
```

This example output is generated with Ansible's default level of verbosity. If you run into problems during this process, you can increase Ansible's level of verbosity by passing one or more `-v` switches to the `ansible` command when you run it. For most issues, it is recommended that you use `-vvvv`, which gives you ample debugging information, including the raw SSH commands and the output from them. For example, assume that a certain host (such as `backend2.example.com`) can't be connected to and you receive an error similar to the following:

```
backend2.example.com | FAILED => SSH encountered an unknown error during
the connection. We recommend you re-run the command using -vvvv, which will
enable SSH debugging output to help diagnose the issue
```

Note that even Ansible recommends the use of the `-vvvv` switch for debugging. This could potentially produce pages of output but will include many useful details, such as the raw SSH command that was used to generate the connection to the target host in the inventory, along with any error messages that may have resulted from that call. This can be incredibly useful when debugging connectivity or code issues, although the output might be a little overwhelming at first. However, with some practice, you will quickly learn to interpret it.

By now, you should have a good idea of how Ansible communicates with its clients over SSH. Let's proceed to the next section, where we will look in more detail at the various components that make up Ansible, as this will help us understand how to work with it better.

Breaking down the Ansible components

Ansible allows you to define policies, configurations, task sequences, and orchestration steps in playbooks—the limit is really only your imagination. A playbook can be executed to manage your tasks either synchronously or asynchronously on a remote machine, although you will find that most examples are synchronous. In this section, you will learn about the main components of Ansible and understand how Ansible employs those components to communicate with remote hosts.

In order to understand the various components, we first need an inventory to work from. Let's create an example one, ideally with multiple hosts in it—this could be the same as the one you created in the previous section. As discussed in that section, you should populate the inventory with the hostnames or IP addresses of the hosts that you can reach from the control host itself:

```
remote1.example.com
remote2.example.com
remote3.example.com
```

To really understand how Ansible—as well as its various components—works, we first need to create an Ansible playbook. While the ad hoc commands that we have experimented with so far are just single tasks, playbooks are organized groups of tasks that are (usually) run in sequence. Conditional logic can be applied and in any other programming language, they would be considered your code. At the head of the playbook, you should specify the name of your play—although this is not mandatory, it is good practice to name all your plays and tasks as without this, it would be quite hard for someone else to interpret what the playbook does, or even for you to if you come back to it after a period of time. Let's get started with building our first example playbook:

1. Specify the play name and inventory hosts to run your tasks against at the very top of your playbook. Also, note the use of `---`, which denotes the beginning of a YAML file (Ansible playbooks that are written in YAML):

   ```
   ---
   - name: My first Ansible playbook
     hosts: all
   ```

2. After this, we will tell Ansible that we want to perform all the tasks in this playbook as a superuser (usually `root`). We do this with the following statement (to aid your memory, think of `become` as shorthand for `become superuser`):

   ```
     become: yes
   ```

3. After this header, we will specify a task block that will contain one or more tasks to be run in sequence. For now, we will simply create one task to update the version of Apache using the `yum` module (because of this, this playbook is only suitable for running against RHEL-, CentOS-, or Fedora-based hosts). We will also specify a special element of the play called a handler. Handlers will be covered in greater detail in `Chapter 4`, *Playbooks and Roles*, so don't worry too much about them for now. Simply put, a handler is a special type of task that is called only if something changes. So, in this example, it restarts the web server, but only if it changes, preventing unnecessary restarts if the playbook is run several times and there are no updates for Apache. The following code performs these functions exactly and should form the basis of your first playbook:

```
 tasks:
 - name: Update the latest of an Apache Web Server
   yum:
     name: httpd
     state: latest
   notify:
     - Restart an Apache Web Server

handlers:
- name: Restart an Apache Web Server
  service:
    name: httpd
    state: restarted
```

Congratulations, you now have your very first Ansible playbook! If you run this now, you should see it iterate through all the hosts in your inventory, as well as on each update in the Apache package, and then restart the service where the package was updated. Your output should look something as follows:

```
$ PLAY [My first Ansible playbook]
***********************************************

TASK [Gathering Facts]
*********************************************************
ok: [remote2.example.com]
ok: [remote1.example.com]
ok: [remote3.example.com]

TASK [Update the latest of an Apache Web Server]
*******************************
changed: [remote2.example.com]
changed: [remote3.example.com]
changed: [remote1.example.com]
```

```
RUNNING HANDLER [Restart an Apache Web Server]
*********************************
changed: [remote3.example.com]
changed: [remote1.example.com]
changed: [remote2.example.com]

PLAY RECAP
*********************************************************************
remote1.example.com : ok=3 changed=2 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0
remote2.example.com : ok=3 changed=2 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0
remote3.example.com : ok=3 changed=2 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0
```

If you examine the output from the playbook, you can see the value in naming not only the play but also each task that is executed, as it makes interpreting the output of the run a very simple task. You will also see that there are multiple possible results from running a task; in the preceding example, we can see two of these results—`ok` and `changed`. Most of these results are fairly self-explanatory, with `ok` meaning the task ran successfully and that nothing changed as a result of the run. An example of this in the preceding playbook is the `Gathering Facts` stage, which is a read-only task that gathers information about the target hosts. As a result, it can only ever return `ok` or a failed status, such as `unreachable`, if the host is down. It should never return `changed`.

However, you can see in the preceding output that all three hosts need to upgrade their Apache package and, as a result of this, the results from the `Update the latest of an Apache Web Server` task is `changed` for all the hosts. This `changed` result means that our `handler` variable is notified and the web server service is restarted.

If we run the playbook a second time, we know that it is very unlikely that the Apache package will need upgrading again. Notice how the playbook output differs this time:

```
PLAY [My first Ansible playbook]
***********************************************

TASK [Gathering Facts]
*********************************************************
ok: [remote1.example.com]
ok: [remote2.example.com]
ok: [remote3.example.com]

TASK [Update the latest of an Apache Web Server]
*******************************
ok: [remote2.example.com]
ok: [remote3.example.com]
```

```
ok: [remote1.example.com]

PLAY RECAP
*********************************************************************
remote1.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0
remote2.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0
remote3.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0
```

You can see that this time, the output from the `Update the latest of an Apache Web Server` task is `ok` for all three hosts, meaning no changes were applied (the package was not updated). As a result of this, our handler is not notified and does not run—you can see that it does not even feature in the preceding playbook output. This distinction is important—the goal of an Ansible playbook (and the modules that underpin Ansible) should be to only make changes when they need to be made. If everything is all up to date, then the target host should not be altered. Unnecessary restarts to services should be avoided, as should unnecessary alterations to files. In short, Ansible playbooks are (and should be) designed to be efficient and to achieve a target machine state.

This has very much been a crash course on writing your first playbook, but hopefully, it gives you a taste of what Ansible can do when you move from single ad hoc commands through to more complex playbooks. Before we explore the Ansible language and components any further, let's take a more in-depth look at the YAML language that playbooks are written in.

Learning the YAML syntax

In this section, you will learn how to write a YAML file with the correct syntax and best practices and tips for running a playbook on multiple remote machines. Ansible uses YAML because it is easier for humans to read and write than other common data formats, such as XML or JSON. There are no commas, curly braces, or tags to worry about, and the enforced indentation in the code ensures that it is tidy and easy on the eye. In addition, there are libraries available in most programming languages for working with YAML.

This reflects one of the core goals of Ansible—to produce easy-to-read (and write) code that described the target state of a given host. Ansible playbooks are (ideally) supposed to be self-documenting, as documentation is often an afterthought in busy technology environments—so, what better way to document than through the automation system responsible for deploying code?

Before we dive into YAML structure, a word on the files themselves. Files written in YAML can optionally begin with `---` (as seen in the example playbook in the previous section) and end with `...`. This applies to all files in YAML, regardless of whether it is employed by Ansible or another system, and indicates that the file is in the YAML language. You will find that most examples of Ansible playbooks (as well as roles and other associated YAML files) start with `---` but do not end with `...`—the header is sufficient to clearly denote that the file uses the YAML format.

Let's explore the YAML language through the example playbook we created in the preceding section:

1. Lists are an important construct in the YAML language—in fact, although it might not be obvious, the `tasks:` block of the playbook is actually a YAML list. A list in YAML lists all of its items at the same indentation level, with each line starting with `-`. For example, we updated the `httpd` package from the preceding playbook using the following code:

```
- name: Update the latest of an Apache Web Server
  yum:
    name: httpd
    state: latest
```

   However, we could have specified a list of packages to be upgraded as follows:

```
- name: Update the latest of an Apache Web Server
  yum:
    name:
      - httpd
      - mod_ssl
    state: latest
```

   Now, rather than passing a single value to the `name:` key, we pass a YAML-formatted list containing the names of two packages to be updated.

2. Dictionaries are another important concept in YAML—they are represented by a `key: value` format, as we have already extensively seen, but all of the items in the dictionary are indented by one more level. This is easiest explained by an example, so consider the following code from our example playbook:

```
service:
  name: httpd
  state: restarted
```

In this example (from `handler`), the `service` definition is actually a dictionary and both the `name` and `state` keys are indented with two more spaces than the `service` key. This higher level of indentation means that the `name` and `state` keys are associated with the `service` key, therefore, in this case, telling the `service` module which service to operate on (`httpd`) and what to do with it (restart it).

Already, we have observed in these two examples that you can produce quite complicated data structures by mixing lists and dictionaries.

3. As you become more advanced at playbook design (we will see examples of this later on in this book), you may very well start to produce quite complicated variable structures that you will put into their own separate files to keep your playbook code readable. The following is an example of a `variables` file that provides the details of two employees of a company:

```
---
employees:
  - name: daniel
    fullname: Daniel Oh
    role: DevOps Evangelist
    level: Expert
    skills:
      - Kubernetes
      - Microservices
      - Ansible
      - Linux Container
  - name: michael
    fullname: Michael Smiths
    role: Enterprise Architect
    level: Advanced
    skills:
      - Cloud
      - Middleware
      - Windows
      - Storage
```

In this example, you can see that we have a dictionary containing the details of each employee. The employees themselves are list items (you can spot this because the lines start with -) and equally, the employee skills are denoted as list items. You will notice the `fullname`, `role`, `level`, and `skills` keys are at the same indentation level as `name` but do not feature - before them. This tells you that they are in the dictionary with the list item itself, and so they represent the details of the employee.

4. YAML is very literal when it comes to parsing the language and a new line always represents a new line of code. What if you actually need to add a block of text (for example, to a variable)? In this case, you can use a literal block scalar, `|`, to write multiple lines and YAML will faithfully preserve the new lines, carriage returns, and all the whitespace that follows each line (note, however, that the indentation at the beginning of each line is part of the YAML syntax):

   ```
   Specialty: |
     Agile methodology
     Cloud-native app development practices
     Advanced enterprise DevOps practices
   ```

   So, if we were to get Ansible to print the preceding contents to the screen, it would display as follows (note that the preceding two spaces have gone—they were interpreted correctly as part of the YAML language and not printed):

   ```
   Agile methodology
   Cloud-native app development practices
   Advanced enterprise DevOps practices
   ```

   Similar to the preceding is the folded block scalar, `>`, which does the same as the literal block scalar but does not preserve line endings. This is useful for very long strings that you want to print on a single line, but also want to wrap across multiple lines in your code for the purpose of readability. Take the following variation on our example:

   ```
   Specialty: >
     Agile methodology
     Cloud-native app development practices
     Advanced enterprise DevOps practices
   ```

   Now, if we were to print this, we would see the following:

```
Agile methodologyCloud-native app development practicesAdvanced enterprise
DevOps practices
```

   We could add trailing spaces to the preceding example to stop the words from running into each other, but I have not done this here as I wanted to provide you with an easy-to-interpret example.

As you review playbooks, variable files, and so on, you will see these structures used over and over again. Although simple in definition, they are very important—a missed level of indentation or a missing – instance at the start of a list item can cause your entire playbook to fail to run. As we discovered, you can put all of these various constructs together. One additional example is provided in the following code block of a `variables` file for you to consider, which shows the various examples we have covered all in one place:

```
---
servers:
  - frontend
  - backend
  - database
  - cache
employees:
  - name: daniel
    fullname: Daniel Oh
    role: DevOps Evangelist
    level: Expert
    skills:
      - Kubernetes
      - Microservices
      - Ansible
      - Linux Container
  - name: michael
    fullname: Michael Smiths
    role: Enterprise Architect
    level: Advanced
    skills:
      - Cloud
      - Middleware
      - Windows
      - Storage
    Speciality: |
      Agile methodology
      Cloud-native app development practices
      Advanced enterprise DevOps practices
```

You can also express both dictionaries and lists in an abbreviated form, known as **flow collections**. The following example shows exactly the same data structure as our original `employees` variable file:

```
---
employees: [{"fullname": "Daniel Oh","level": "Expert","name":
"daniel","role": "DevOps Evangelist","skills":
["Kubernetes","Microservices","Ansible","Linux Container"]},{"fullname":
"Michael Smiths","level": "Advanced","name": "michael","role": "Enterprise
Architect","skills":["Cloud","Middleware","Windows","Storage"]}]
```

Although this displays exactly the same data structure, you can see how difficult it is to read with the naked eye. Flow collections are not used extensively in YAML and I would not recommend you to make use of them yourself, but it is important to understand them in case you come across them. You will also notice that although we've started talking about variables in YAML, we haven't expressed any variable types. YAML tries to make assumptions about variable types based on the data they contain, so if you want assign `1.0` to a variable, YAML will assume it is a floating-point number. If you need to express it as a string (perhaps because it is a version number), you need to put quotation marks around it, which causes the YAML parser to interpret it as a string instead, such as in the following example:

```
version: "2.0"
```

This completes our look at the YAML language syntax. Now that's complete, in the next section, let's take a look at ways that you can organize your automation code to keep it manageable and tidy.

Organizing your automation code

As you can imagine, if you were to write all of your required Ansible tasks in one massive playbook, it would quickly become unmanageable—that is to say, it would be difficult to read, difficult for someone else to pick up and understand, and—most of all—difficult to debug when things go wrong. Ansible provides a number of ways for you to divide your code into manageable chunks; perhaps the most important of these is the use of roles. Roles (for the sake of a simple analogy) behave like a library in a conventional high-level programming language. We will go into more detail about roles in `Chapter 4`, *Playbooks and Roles*.

There are, however, other ways that Ansible supports splitting your code into manageable chunks, which we will explore briefly in this section as a precursor to the more in-depth exploration of roles later in this book.

Let's build up a practical example. To start, we know that we need to create an inventory for Ansible to run against. In this instance, we'll create four notional groups of servers, with each group containing two servers. Our hypothetical example will contain a frontend server and application servers for a fictional application, located in two different geographic locations. Our inventory file will be called `production-inventory` and the example contents are as follows:

```
[frontends_na_zone]
frontend1-na.example.com
frontend2-na.example.com
```

```
[frontends_emea_zone]
frontend1-emea.example.com
frontend2-emea.example.com

[appservers_na_zone]
appserver1-na.example.com
appserver2-na.example.com

[appservers_emea_zone]
appserver1-emea.example.com
appserver2-emea.example.com
```

Now, obviously, we could just write one massive playbook to address the required tasks on these different hosts, but as we have already discussed, this would be cumbersome and inefficient. Let's instead break the task of automating these different hosts down into smaller playbooks:

1. Create a playbook to run a connection test on a specific host group, such as `frontends_na_zone`. Put the following contents into the playbook:

   ```
   ---
   - hosts: frontends_na_zone
     remote_user: danieloh
     tasks:
       - name: simple connection test
         ping:
   ```

2. Now, try running this playbook against the hosts (note that we have configured it to connect to a remote user on the inventory system, called `danieloh`, so you will either need to create this user and set up the appropriate SSH keys or change the user in the `remote_user` line of your playbook). When you run the playbook after setting up the authentication, you should see an output similar to the following:

   ```
   $ ansible-playbook -i production-inventory frontends-na.yml

   PLAY [frontends_na_zone]
   *******************************************************

   TASK [Gathering Facts]
   *********************************************************
   ok: [frontend1-na.example.com]
   ok: [frontend2-na.example.com]

   TASK [simple connection test]
   **************************************************
   ok: [frontend1-na.example.com]
   ```

```
ok: [frontend2-na.example.com]

PLAY RECAP
*****************************************************************
**
frontend1-na.example.com : ok=2 changed=0 unreachable=0 failed=0
skipped=0 rescued=0 ignored=0
frontend2-na.example.com : ok=2 changed=0 unreachable=0 failed=0
skipped=0 rescued=0 ignored=0
```

3. Now, let's extend our simple example by creating a playbook that will only run on the application servers. Again, we will use the Ansible `ping` module to perform a connection test, but in a real-world situation, you would perform more complex tasks, such as installing packages or modifying files. Specify that this playbook is run against this host group from the `appservers_emea_zone` inventory. Add the following contents to the playbook:

```
---
- hosts: appservers_emea_zone
  remote_user: danieloh
  tasks:
    - name: simple connection test
      ping:
```

As before, you need to ensure you can access these servers, so either create the `danieloh` user and set up authentication to that account or change the `remote_user` line in the example playbook. Once you have done this, you should be able to run the playbook and you will see an output similar to the following:

```
$ ansible-playbook -i production-inventory appservers-emea.yml

PLAY [appservers_emea_zone]
****************************************************

TASK [Gathering Facts]
*********************************************************
ok: [appserver2-emea.example.com]
ok: [appserver1-emea.example.com]

TASK [simple connection test]
**************************************************
ok: [appserver2-emea.example.com]
ok: [appserver1-emea.example.com]

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

```
**
appserver1-emea.example.com : ok=2 changed=0 unreachable=0 failed=0
skipped=0 rescued=0 ignored=0
appserver2-emea.example.com : ok=2 changed=0 unreachable=0 failed=0
skipped=0 rescued=0 ignored=0
```

4. So far, so good. However, we now have two playbooks that we need to run manually, which only addresses two of our inventory host groups. If we want to address all four groups, we need to create a total of four playbooks, all of which need to be run manually. This is hardly reflective of best automation practices. What if there was a way to take these individual playbooks and run them together from one top-level playbook? This would enable us to divide our code to keep it manageable, but also prevents a lot of manual effort when it comes to running the playbooks. Fortunately, we can do exactly that by taking advantage of the `import_playbook` directive in a top-level playbook that we will call `site.yml`:

```
---
- import_playbook: frontend-na.yml
- import_playbook: appserver-emea.yml
```

Now, when you run this single playbook using the (by now, familiar) `ansible-playbook` command, you will see that the effect is the same as if we had actually run both playbooks back to back. In this way, even before we explore the concept of roles, you can see that Ansible supports splitting up your code into manageable chunks without needing to run each chunk manually:

```
$ ansible-playbook -i production-inventory site.yml

PLAY [frontends_na_zone]
*******************************************************

TASK [Gathering Facts]
*********************************************************
ok: [frontend2-na.example.com]
ok: [frontend1-na.example.com]

TASK [simple connection test]
**************************************************
ok: [frontend1-na.example.com]
ok: [frontend2-na.example.com]

PLAY [appservers_emea_zone]
****************************************************

TASK [Gathering Facts]
```

The first instance of the file is the configuration it will use; all of the others are ignored, even if they are present:

1. `ANSIBLE_CONFIG`: The file location specified by the value of this environment variable, if set
2. `ansible.cfg`: In the current working directory
3. `~/.ansible.cfg`: In the home directory of the user
4. `/etc/ansible/ansible.cfg`: The central configuration that we previously mentioned

If you installed Ansible through a package manager, such as `yum` or `apt`, you will almost always find a default configuration file called `ansible.cfg` in `/etc/ansible`. However, if you built Ansible from the source or installed it via `pip`, the central configuration file will not exist and you will need to create it yourself. A good starting point is to reference the example Ansible configuration file that is included with the source code, a copy of which can be found on GitHub at `https://raw.githubusercontent.com/ansible/ansible/devel/examples/ansible.cfg`.

In this section, we will detail how to locate Ansible's running configuration and how to manipulate it. Most people who install Ansible through a package find that they can get a long way with Ansible before they have to modify the default configuration, as it has been carefully designed to work in a great many scenarios. However, it is important to know a little about configuring Ansible in case you come across an issue in your environment that can only be changed by modifying the configuration.

Obviously, if you don't have Ansible installed, there's little point in exploring its configuration, so let's just check whether you have Ansible installed and working by issuing a command such as the following (the output shown is from the latest version of Ansible at the time of writing, installed on macOS with Homebrew):

```
$ ansible 2.9.6
  config file = None
  configured module search path = ['/Users/james/.ansible/plugins/modules',
'/usr/share/ansible/plugins/modules']
  ansible python module location =
/usr/local/Cellar/ansible/2.9.6_1/libexec/lib/python3.8/site-
packages/ansible
  executable location = /usr/local/bin/ansible
  python version = 3.8.2 (default, Mar 11 2020, 00:28:52) [Clang 11.0.0
(clang-1100.0.33.17)]
```

Let's get started by exploring the default configuration that is provided with Ansible:

1. The command in the following code block lists the current configuration parameters supported by Ansible. It is incredibly useful because it tells you both the environment variable that can be used to change the setting (see the `env` field) as well as the configuration file parameter and section that can be used (see the `ini` field). Other valuable information, including the default configuration values and a description of the configuration, is given (see the `default` and `description` fields, respectively). All of the information is sourced from `lib/constants.py`. Run the following command to explore the output:

```
$ ansible-config list
```

   The following is an example of the kind of output you will see. There are, of course, many pages to it, but a snippet is shown here as an example:

```
$ ansible-config list
ACTION_WARNINGS:
  default: true
  description:
  - By default Ansible will issue a warning when received from a
task action (module
    or action plugin)
  - These warnings can be silenced by adjusting this setting to
False.
  env:
  - name: ANSIBLE_ACTION_WARNINGS
  ini:
  - key: action_warnings
    section: defaults
  name: Toggle action warnings
  type: boolean
  version_added: '2.5'
AGNOSTIC_BECOME_PROMPT:
  default: true
  description: Display an agnostic become prompt instead of
displaying a prompt containing
    the command line supplied become method
  env:
  - name: ANSIBLE_AGNOSTIC_BECOME_PROMPT
  ini:
  - key: agnostic_become_prompt
    section: privilege_escalation
  name: Display an agnostic become prompt
  type: boolean
  version_added: '2.5'
  yaml:
    key: privilege_escalation.agnostic_become_prompt
.....
```

2. If you want to see a straightforward display of all the possible configuration parameters, along with their current values (regardless of whether they are configured from environment variables or a configuration file in one of the previously listed locations), you can run the following command:

```
$ ansible-config dump
```

The output shows all the configuration parameters (in an environment variable format), along with the current settings. If the parameter is configured with its default value, you are told so (see the `(default)` element after each parameter name):

```
$ ansible-config dump
ACTION_WARNINGS(default) = True
AGNOSTIC_BECOME_PROMPT(default) = True
ALLOW_WORLD_READABLE_TMPFILES(default) = False
ANSIBLE_CONNECTION_PATH(default) = None
ANSIBLE_COW_PATH(default) = None
ANSIBLE_COW_SELECTION(default) = default
ANSIBLE_COW_WHITELIST(default) = ['bud-frogs', 'bunny', 'cheese',
'daemon', 'default', 'dragon', 'elephant-in-snake', 'elephant',
'eyes', 'hellokitty', 'kitty', 'luke-koala', 'meow', 'milk',
'moofasa', 'moose', 'ren', 'sheep', 'small', 'stegosaurus',
'stimpy', 'supermilker', 'three-eyes', 'turkey', 'turtle', 'tux',
'udder', 'vader-koala', 'vader', 'www']
ANSIBLE_FORCE_COLOR(default) = False
ANSIBLE_NOCOLOR(default) = False
ANSIBLE_NOCOWS(default) = False
ANSIBLE_PIPELINING(default) = False
ANSIBLE_SSH_ARGS(default) = -C -o ControlMaster=auto -o
ControlPersist=60s
ANSIBLE_SSH_CONTROL_PATH(default) = None
ANSIBLE_SSH_CONTROL_PATH_DIR(default) = ~/.ansible/cp
....
```

3. Let's see the effect on this output by editing one of the configuration parameters. Let's do this by setting an environment variable, as follows (this command has been tested in the `bash` shell, but may differ for other shells):

```
$ export ANSIBLE_FORCE_COLOR=True
```

Now, let's re-run the ansible-config command, but this time get it to tell us only the parameters that have been changed from their default values:

```
$ ansible-config dump --only-change
ANSIBLE_FORCE_COLOR(env: ANSIBLE_FORCE_COLOR) = True
```

Here, you can see that ansible-config tells us that we have only changed ANSIBLE_FORCE_COLOR from the default value, that it is set to True, and that we set it through an env variable. This is incredibly valuable, especially if you have to debug configuration issues.

When working with the Ansible configuration file itself, you will note that it is in INI format, meaning it has sections such as [defaults], parameters in the format key = value, and comments beginning with either # or ;. You only need to place the parameters you wish to change from their defaults in your configuration file, so if you wanted to create a simple configuration to change the location of your default inventory file, it might look as follows:

```
# Set my configuration variables
[defaults]
inventory = /Users/danieloh/ansible/hosts ; Here is the path of the
inventory file
```

As discussed earlier, one of the possible valid locations for the ansible.cfg configuration file is in your current working directory. It is likely that this is within your home directory, so on a multi-user system, we strongly recommend you restrict access to the Ansible configuration file to your user account alone. You should take all the usual precautions when it comes to securing important configuration files on a multi-user system, especially as Ansible is normally used to configure multiple remote systems and so, a lot of damage could be done if a configuration file is inadvertently compromised!

Of course, Ansible's behavior is not just controlled by the configuration files and switches—the command-line arguments that you pass to the various Ansible executables are also of vital importance. In fact, we have already worked with one already—in the preceding example, we showed you how to change where Ansible looks for its inventory file using the inventory parameter in ansible.cfg. However, in many of the examples that we previously covered in this book, we overrode this with the -i switch when running Ansible. So, let's proceed to the next section to look at the use of command-line arguments when running Ansible.

Command-line arguments

In this section, you will learn about the use of command-line arguments for playbook execution and how to employ some of the more commonly used ones to your advantage. We are already very familiar with one of these arguments, the `--version` switch, which we use to confirm that Ansible is installed (and which version is installed):

```
$ ansible 2.9.6
  config file = None
  configured module search path = ['/Users/james/.ansible/plugins/modules',
'/usr/share/ansible/plugins/modules']
  ansible python module location =
/usr/local/Cellar/ansible/2.9.6_1/libexec/lib/python3.8/site-
packages/ansible
  executable location = /usr/local/bin/ansible
  python version = 3.8.2 (default, Mar 11 2020, 00:28:52) [Clang 11.0.0
(clang-1100.0.33.17)]
```

Just as we were able to learn about the various configuration parameters directly through Ansible, we can also learn about the command-line arguments. Almost all of the Ansible executables have a `--help` option that you can run to display the valid command-line parameters. Let's try this out now:

1. You can view all the options and arguments when you execute the `ansible` command line. Use the following command:

   ```
   $ ansible --help
   ```

   You will see a great deal of helpful output when you run the preceding command; an example of this is shown in the following code block (you might want to pipe this into a pager, such as `less`, so that you can read it all easily):

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

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

positional arguments:
  pattern host pattern

optional arguments:
  --ask-vault-pass ask for vault password
  --list-hosts outputs a list of matching hosts; does not execute
anything else
  --playbook-dir BASEDIR
                        Since this tool does not use playbooks, use
this as a substitute playbook directory.This sets the relative path
for many features including roles/ group_vars/ etc.
  --syntax-check perform a syntax check on the playbook, but do not
execute it
  --vault-id VAULT_IDS the vault identity to use
  --vault-password-file VAULT_PASSWORD_FILES
                        vault password file
  --version show program's version number, config file location,
configured module search path, module location, executable location
and exit
  -B SECONDS, --background SECONDS
                        run asynchronously, failing after X seconds
(default=N/A)
  -C, --check don't make any changes; instead, try to predict some
of the changes that may occur
  -D, --diff when changing (small) files and templates, show the
differences in those files; works great with --check
  -M MODULE_PATH, --module-path MODULE_PATH
                        prepend colon-separated path(s) to module
library
(default=~/.ansible/plugins/modules:/usr/share/ansible/plugins/modu
les)
  -P POLL_INTERVAL, --poll POLL_INTERVAL
                        set the poll interval if using -B
(default=15)
  -a MODULE_ARGS, --args MODULE_ARGS
                        module arguments
  -e EXTRA_VARS, --extra-vars EXTRA_VARS
                        set additional variables as key=value or
YAML/JSON, if filename prepend with @
```

Index