

Learn Ansible

Automate cloud, security, and network infrastructure using Ansible 2.x



Packt

www.packt.com

By Russ Kendrick

Learn Ansible

Automate cloud, security, and network infrastructure using Ansible 2.x

Russ McKendrick

Packt›

BIRMINGHAM - MUMBAI

Learn Ansible

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Gebin George

Acquisition Editor: Shrilekha Inani

Content Development Editor: Nithin George Varghese

Technical Editor: Khushbu Sutar

Copy Editor: Safis Editing

Project Coordinator: Virginia Dias

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Graphics: Tom Scaria

Production Coordinator: Shantanu Zagade

First published: June 2018

Production reference: 1270618

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78899-875-8

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Russ McKendrick is an experienced solution architect who has been working in IT and related industries for over 25 years. During his career, he has had varied responsibilities, from looking after an entire IT infrastructure to providing first-line, second-line, and senior support in both client-facing and internal teams for large organizations.

Russ supports open source systems and tools on public and private clouds at Node4 Limited, where he heads up the public cloud team.

I would like to thank my family and friends for their support and being so understanding about all of the time I have spent in front of the computer writing. I would also like to thank my colleagues at Node4 and our customers for their kind words of support and encouragement throughout the writing process.

About the reviewer

Paul Adamson has worked as an Ops engineer, a developer, a DevOps engineer, and all variations and mixes of all of these. When not reviewing this book, Paul keeps busy helping companies embrace the AWS infrastructure. His language of choice is PHP for all the good reasons and even some of the bad, but mainly habit. While reviewing this book, Paul has been working for Healthy Performance Ltd, helping to apply cutting-edge technology to a cutting-edge approach to well-being.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

[Title Page](#)

[Copyright and Credits](#)

[Learn Ansible](#)

[Packt Upsell](#)

[Why subscribe?](#)

[PacktPub.com](#)

[Contributors](#)

[About the author](#)

[About the reviewer](#)

[Packt is searching for authors like you](#)

[Preface](#)

[Who this book is for](#)

[What this book covers](#)

[To get the most out of this book](#)

[Download the example code files](#)

[Download the color images](#)

[Conventions used](#)

[Get in touch](#)

[Reviews](#)

[1. An Introduction to Ansible](#)

[Ansible's story](#)

[The term](#)

[The software](#)

[Ansible versus other tools](#)

[Declarative versus imperative](#)

[Configuration versus orchestration](#)

[Infrastructure as code](#)

[Summary](#)

[Further reading](#)

2. Installing and Running Ansible

Technical requirements

Installing Ansible

Installing on macOS

Homebrew

The pip method

Pros and cons

Installing on Linux

Installing on Windows 10 Professional

Launching a virtual machine

An introduction to playbooks

Host inventories

Playbooks

Summary

Questions

Further reading

3. The Ansible Commands

[Technical requirements](#)

[Inbuilt commands](#)

[Ansible](#)

[The ansible-config command](#)

[The ansible-console command](#)

[The ansible-doc command](#)

[The ansible-inventory command](#)

[Ansible Vault](#)

[Third-party commands](#)

[The ansible-inventory-grapher command](#)

[Ansible Run Analysis](#)

[Summary](#)

[Questions](#)

[Further reading](#)

4. Deploying a LAMP Stack

```
    Technical requirements  
  
    Playbook structure  
  
        LAMP stack  
  
            Common  
  
                Updating packages  
  
                Installing common packages  
  
                Configuring NTP  
  
                Creating a user  
  
                Running the role  
  
            Apache  
  
                Installing Apache  
  
                Configuring Apache  
  
                Configuring SELinux  
  
                Copying an HTML file  
  
                Running the role  
  
            MariaDB  
  
                Installing MariaDB  
  
                Configuring MariaDB  
  
                Importing a sample database  
  
                Running the role  
  
            PHP  
  
                Installing PHP  
  
                The phpinfo file  
  
                Adminer  
  
                Running the role
```

[Overriding variables](#)

[Summary](#)

[Questions](#)

[Further reading](#)

5. Deploying WordPress

Technical requirements

Preinstallation tasks

The stack-install command

Enabling the repositories

Installing the packages

The stack-config role

WordPress system user

NGINX configuration

PHP and PHP-FPM configuration

Starting NGINX and PHP-FPM

MariaDB Configuration

SELinux configuration

WordPress installation tasks

WordPress CLI installation

Creating the WordPress database

Downloading, configuring, and installing WordPress

WordPress plugins and theme installation

Running the WordPress playbook

Summary

Questions

Further reading

6. Targeting Multiple Distributions

Technical requirements

Launching multiple Vagrant boxes

Multi-operating system considerations

Adapting the roles

Operating system family

The stack-install role

The stack-config role

The wordpress role

Running the playbook

Summary

Questions

Further reading

7. The Core Network Modules

Technical requirements

Manufacturer and device support

The modules

A10 Networks

Cisco Application Centric Infrastructure (ACI)

Cisco AireOS

Apstra Operating System (AOS)

Aruba Mobility Controller

Cisco Adaptive Security Appliance (ASA)

Avi Networks

Big Switch Networks

Citrix Netscaler

Huawei CloudEngine (CE)

Arista CloudVision (CV)

Lenovo CNOS

Cumulus Linux (CL)

Dell operating system 10 (DellOS10)

Ubiquiti EdgeOS

Lenovo Enterprise Networking Operating System (ENOS)

Arista EOS

F5 BIG-IP

FortiGate FortiManager

FortiGate FortiOS

illumos

Cisco IOS and IOS XR

[Brocade IronWare](#)

[Juniper Junos](#)

[Nokia NetAct](#)

[Pluribus Networks Netvisor OS](#)

[Cisco Network Services Orchestrator \(NSO\)](#)

[Nokia Nuage Networks Virtualized Services Platform \(VSP\)](#)

[Cisco NX-OS \(NXOS\)](#)

[Mellanox ONYX](#)

[Ordnance](#)

[Open vSwitch \(OVS\)](#)

[Palo Alto Networks PAN-OS](#)

[Radware](#)

[Nokia Networks Service Router Operating System \(SROS\)](#)

[VyOS](#)

[System](#)

[Interacting with a network device](#)

[Launching the network device](#)

[The VyOS role](#)

[Running the playbook](#)

[Summary](#)

[Questions](#)

[Further reading](#)

8. Moving to the Cloud

[Technical requirements](#)

[Interacting with DigitalOcean](#)

[Generating a personal access token](#)

[Installing dopy](#)

[Launching a Droplet](#)

[Running the playbook](#)

[WordPress on DigitalOcean](#)

[The host inventory](#)

[Variables](#)

[The playbook](#)

[The droplet role](#)

[Running the playbook](#)

[Summary](#)

[Questions](#)

[Further reading](#)

9. Building Out a Cloud Network

[Technical requirements](#)

[An introduction to AWS](#)

[Amazon Virtual Private Cloud overview](#)

[Creating an access key and secret](#)

[The VPC playbook](#)

[The VPC role](#)

[The subnets role](#)

[The internet gateway role](#)

[The security group role](#)

[The ELB role](#)

[Summary](#)

[Questions](#)

[Further reading](#)

10. Highly Available Cloud Deployments

Technical requirements

Planning the deployment

Costing the deployment

WordPress considerations and high availability

The playbook

Amazon VPC

Amazon RDS

Amazon EFS

Testing the playbook

Terminating resources

EC2 instances

Instance discovery

New deployment

Existing deployment

Stack

Default variables

Deploy

WordPress

AMI

Autoscaling

Running the playbook

Terminating all the resources

Summary

Questions

Further reading

11. Building Out a VMware Deployment

Technical requirements

An introduction to VMware

The VMware modules

Requirements

vCloud Air

The vca_fw module

The vca_nat module

The vca_vapp module

VMware vSphere

The vmware_cluster module

The vmware_datacenter module

The vmware_vm_facts module

The vmware_vm_shell module

The vmware_vm_vm_drs_rule module

The vmware_vm_vss_dvs_migrate module

The vsphere_copy module

The vsphere_guest module

VMware vCentre

The vcenter_folder module

The vcenter_license module

The vmware_guest module

The vmware_guest_facts module

The vmware_guest_file_operation module

The vmware_guest_find module

The vmware_guest_powerstate module

The `vmware_guest_snapshot` module

The `vmware_guest_tools_wait` module

VMware ESXi

The `vmware_dns_config` module

The `vmware_host_dns_facts` module

The `vmware_host` module

The `vmware_host_facts` module

The `vmware_host_acceptance` module

The `vmware_host_config_manager` module

The `vmware_host_datastore` module

The `vmware_host_firewall_manager` module

The `vmware_host_firewall_facts` module

The `vmware_host_lockdown` module

The `vmware_host_ntp` module

The `vmware_host_package_facts` module

The `vmware_host_service_manager` module

The `vmware_host_service_facts` module

The `vmware_datastore_facts` module

The `vmware_host_vmnic_facts` module

The `vmware_local_role_manager` module

The `vmware_local_user_manager` module

The `vmware_cfg_backup` module

The `vmware_vmkernel` module

The `vmware_vmkernel_facts` module

The `vmware_target_canonical_facts` module

The `vmware_vmotion` module

The `vmware_vsan_cluster` module

[The vmware_vswitch module](#)

[The vmware_drs_rule_facts module](#)

[The vmware_dvswitch module](#)

[The vmware_dvs_host module](#)

[The vmware_dvs_portgroup module](#)

[The vmware_maintenancemode module](#)

[The vmware_portgroup module](#)

[The vmware_resource_pool module](#)

[An example playbook](#)

[Summary](#)

[Questions](#)

[Further reading](#)

12. Ansible Windows Modules

Technical requirements

Up-and-running

Vagrantfile

Ansible preparation

The ping module

The setup module

Installing a web server

IIS role

ASP.NET role

Interacting with AWS Windows instances

AWS role

User role

Chocolatey role

Information role

Running the playbook

Summary

Questions

Further reading

13. Hardening Your Servers Using Ansible and OpenSCAP

[Technical requirements](#)

[OpenSCAP](#)

[Preparing the host](#)

[The playbook](#)

[Install role](#)

[Scan role](#)

[Running the initial scan](#)

[Generating the remediation Ansible playbook](#)

[Generating the remediation bash script](#)

[Running a standalone scan](#)

[Fixing the remaining failed checks](#)

[Destroying the Vagrant box](#)

[Summary](#)

[Questions](#)

[Further reading](#)

14. Deploying WPScan and OWASP ZAP

[Preparing the boxes](#)

[The WordPress playbook](#)

[The scan playbook](#)

[The Docker role](#)

[Testing the playbook](#)

[The WPScan role](#)

[Running a WPScan](#)

[The OWASP ZAP role](#)

[Running OWASP ZAP](#)

[Summary](#)

[Questions](#)

[Further reading](#)

15. Introducing Ansible Tower and Ansible AWX

[Technical requirements](#)

[Web-based Ansible](#)

[Ansible Tower](#)

[Updating the inventory file](#)

[Running the playbook](#)

[Requesting a license](#)

[The hello world demo project](#)

[Launching the AWS playbook](#)

[Adding a new project](#)

[Adding credentials](#)

[Adding an inventory](#)

[Adding the templates](#)

[Running the playbook](#)

[Removing the cluster](#)

[Tower summary](#)

[Ansible AWX](#)

[Preparing the playbook](#)

[The docker role](#)

[The awx role](#)

[Running the playbook](#)

[Using Ansible AWX](#)

[AWX summary](#)

[Summary](#)

[Questions](#)

[Further reading](#)

16. Ansible Galaxy

[Technical requirements](#)

[Introduction to Ansible Galaxy](#)

[Jenkins playbook](#)

[Publishing a role](#)

[Creating the docker role](#)

[Variables](#)

[Tasks](#)

[Metadata](#)

[README](#)

[Committing the code and publishing](#)

[Testing the role](#)

[Ansible Galaxy commands](#)

[Logging in](#)

[Importing](#)

[Searching](#)

[Info](#)

[Summary](#)

[Questions](#)

[Further reading](#)

17. Next Steps with Ansible

[Integrating with third-party services](#)

[Slack](#)

[Generating a token](#)

[The Ansible playbook](#)

[Running the playbook](#)

[Other services](#)

[Campfire](#)

[Cisco Webex Teams \(Cisco Spark\)](#)

[CA Flowspace](#)

[Hipchat](#)

[Mail](#)

[Mattermost](#)

[Say](#)

[ServiceNow](#)

[Syslog](#)

[Twilio](#)

[Summary of third-party services](#)

[The Ansible playbook debugger](#)

[Debugging the task](#)

[Summary of the Ansible debugger](#)

[Real-world examples](#)

[The chat example](#)

[Automated deployment](#)

[Summary](#)

[Further reading](#)

Assessments

[Chapter 2, Installing and Running Ansible](#);

[Chapter 3, The Ansible Commands](#)

[Chapter 4, Deploying a LAMP Stack](#)

[Chapter 5, Deploying WordPress](#)

[Chapter 6, Targeting Multiple Distributions](#)

[Chapter 7, The Core Network Modules](#)

[Chapter 8, Moving to the Cloud](#)

[Chapter 9, Building Out a Cloud Network](#)

[Chapter 10, Highly Available Cloud Deployments](#)

[Chapter 11, Building Out a VMware Deployment](#)

[Chapter 12, Ansible Windows Modules](#)

[Chapter 13, Hardening Your Servers Using Ansible and OpenSCAP](#)

[Chapter 14, Deploying WPScan and OWASP ZAP](#)

[Chapter 15, Introducing Ansible Tower and Ansible AWX](#)

Other Books You May Enjoy

[Leave a review - let other readers know what you think](#)

Preface

Ansible has rapidly grown from a small open source orchestration tool to a full-blown orchestration and configuration management tool owned by Red Hat. In this book, you will learn how to write playbooks using the core Ansible modules to deploy everything from basic LAMP stacks to a full-blown highly available public cloud infrastructure.

By the end of the book, you will have learned how to do the following:

- Written your own playbooks to configure servers running CentOS 7, Ubuntu 17.04, and Windows Server
- Defined a highly available cloud infrastructure in code, making it easy to distribute your infrastructure configuration alongside your own code
- Deployed and configured Ansible Tower and Ansible AWX
- Used community contributed roles and learned how to contribute your own roles
- Worked through several use cases of how you can use Ansible in your day-to-day role and projects

By the end of the book, you should have a good idea of how to integrate Ansible in your day-to-day roles as system administrators, developers, and DevOps practitioners.

Who this book is for

This book is perfect for system administrators, developers, and DevOps practitioners who want to take their current workflows and transform them into repeatable playbooks using Ansible. No prior knowledge of Ansible is required.

What this book covers

[Chapter 1](#), *An Introduction to Ansible*, discusses what problems Ansible has been developed to resolve, who wrote it, and talks about Red Hat's involvement following their acquisition of Ansible.

[Chapter 2](#), *Installing and Running Ansible*, discusses how we will work through installing Ansible on macOS and Linux, after covering its background. We will also discuss why there is no native Windows installer and cover installing Ansible on the Ubuntu shell in Windows 10 Professional.

[Chapter 3](#), *The Ansible Commands*, explains how, before moving onto writing and executing more advanced playbooks, we are going to take a look at the Ansible commands. Here, we will cover the usage of the set of commands that make up Ansible.

[Chapter 4](#), *Deploying a LAMP Stack*, discusses the deployment of a full LAMP stack using the various core modules that ship with Ansible. We will target the CentOS 7 virtual machine running locally.

[Chapter 5](#), *Deploying WordPress*, explains the use of the LAMP stack, which we deployed in the previous chapter as our base. We will use Ansible to download, install, and configure WordPress.

[Chapter 6](#), *Targeting Multiple Distributions*, explains how we will work through adapting the playbook, so it can run against both Ubuntu 17.04 and CentOS 7 servers. The final playbook from the previous two chapters have been written to target a CentOS 7 virtual machine.

[Chapter 7](#), *The Core Network Modules*, explains how we will take a look at the core network modules that ship with Ansible. Due to the requirements of these modules, we will only be touching upon the functionality these modules provide.

[Chapter 8](#), *Moving to the Cloud*, discusses how we will move from using local virtual machines to using Ansible to launch a Droplet in DigitalOcean, and then we will use the playbook from the previous chapters to install and configure a

LAMP stack and WordPress.

[Chapter 9](#), *Building Out a Cloud Network*, discusses how after launching the servers in DigitalOcean. We will move onto Amazon Web Services before we launch instances. We will need to create a network for them to be hosted in.

[Chapter 10](#), *Highly Available Cloud Deployments*, continues with our Amazon Web Services deployment. We will start to deploy services into the network we created in the previous chapter, and by the end of the chapter, we will be left with a highly available WordPress installation.

[Chapter 11](#), *Building Out a VMware Deployment*, discusses the core modules that allow you to interact with the various components that make up a typical VMware installation.

[Chapter 12](#), *Ansible Windows Modules*, takes a look at the ever-growing collection of core Ansible modules that support and interact with Windows-based servers.

[Chapter 13](#), *Hardening Your Servers Using Ansible and OpenSCAP*, explains how you can use Ansible to install and execute OpenSCAP. We will also look at using Ansible to solve any problems found during the OpenSCAP scan.

[Chapter 14](#), *Deploying WPScan and OWASP ZAP*, explains the creation of a playbook that deploys and runs two security tools, OWASP ZAP and WPScan. Then, using the playbooks from previous chapters from previous chapters launches a WordPress installation to run them against.

[Chapter 15](#), *Introducing Ansible Tower and Ansible AWX*, takes a look at the two graphical interfaces to Ansible, the commercial Ansible Tower and Open Source Ansible AWX.

[Chapter 16](#), *Ansible Galaxy*, discusses Ansible Galaxy, which is an online repository of community contributed roles. In this chapter, we will discover some of the best roles available, how to use them, and how to create your own role and have it hosted on Ansible Galaxy.

[Chapter 17](#), *Next Steps with Ansible*, teaches us how Ansible can be integrated into our day-to-day workflows, from interacting with collaboration services to troubleshooting your playbooks with the built-in debugger. We will also look at

some real-world examples of how I use Ansible.

To get the most out of this book

To get the most out of this book, I assume you that you:

- Have had some experience of using the command line on both Linux-based machines and also macOS
- Have a basic understanding of how to install and configure services on a Linux server
- Have a working knowledge of services and languages, such as Git, YAML, and virtualization

Download the example code files

You can download the example code files for this book from your account at www.packtpub.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.packtpub.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
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
- Ziipeg/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/Learn-Ansible>. 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: https://www.packtpub.com/sites/default/files/downloads/LearnAnsible_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: "This will create a key and store it in the `.ssh` folder in your `user` directory."

A block of code is set as follows:

```
| config.vm.provider "virtualbox" do |v|
|   v.memory = "2024"
|   v.cpus = "2"
| end
```

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

```
| $ sudo -H apt-get install python-pip
| $ sudo -H pip install ansible
```

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: "To do this, open Control Panel, then click on Programs and Features, and then on Turn Windows features on or off."



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: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@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/submit-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@packtpub.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 packtpub.com.

An Introduction to Ansible

In our first chapter, we are going to be looking at the technology world before tools such as Ansible came into existence in order to get an understanding of why Ansible was needed.

Before we start to talk about Ansible, let's quickly discuss the old world. I have been working with servers, mostly ones that serve web pages, since the late 90s, and the landscape is unrecognizable. To give you an idea of how I used to operate my early servers, here is a quick overview of my first few years running servers.

Like most people at the time, I started with a shared hosting account where I had very little control over anything on the server side when the site I was running at the time outgrew shared hosting. I moved to a dedicated server—this is where I thought I would be able to flex my future system administrator muscles, but I was wrong.

The server I got was a Cobalt RaQ 3, a 1U server appliance, which, in my opinion, was ahead of its time. However, I did not have root level access to the machine and for everything I needed to do, I had to use the web-based control panel. Eventually, I got a level of access where I could access the server using SSH or Telnet (I know, it was the early days), and I started to teach myself how to be a system administrator by making changes in the web control panel and looking at the changes to the configuration files on the server.

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

After I got my second server for another project, I realized that was probably a good time to type out my notes so that I could copy and paste them when I needed to deploy a server, which I am glad I did, as it was shortly after my first

server failed—my host apologized and replaced it with a higher-specification but completely fresh machine with an updated operating system.

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

One of the important lessons I learned, other than that there is no such thing as too many backups, was to not use Microsoft Word to store these types of notes; the command doesn't care if your notes are all nicely formatted with headings and courier font for the bits you need to paste. What it does care about is using proper syntax, which Word had managed to autocorrect and format for print.

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

These scraps of commands, one-liners, and scripts were all adapted through Red Hat Linux 6—note the lack of the word *Enterprise*—all the way through to CentOS 3 and 4.

Things got complicated when I changed roles, stopped consuming services from web hosts, and started working for one. All of a sudden, I was building servers for customers who may have different requirements than my own projects—no one server was the same.

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

Over the next few years, my single text file quickly morphed into a complex collection of notes, scripts, precompiled binaries, and spreadsheets of information that, if I am being honest, really only made sense to me.

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

My story is probably typical of many people, while the operating systems used will probably be considered quite ancient. Now, the entry point of using a GUI and moving to the command line, while also keeping a scratch pad of common commands, is quite a common one I have heard.

We will be covering the following topics:

- Who is behind Ansible
- The differences between Ansible and other tools
- The problem Ansible solves

Ansible's story

Let's take quick a look at who wrote Ansible, and also what Ansible means.

The term

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



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

The term has since been used by several other notable authors within the genre to describe communication devices that are capable of relaying messages over interstellar distances.

The software

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



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

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

A few years after leaving Puppet, DeHaan made the first public commit on the Ansible project; this was on February 23, 2012. The original README file gave quite a simple description that laid the foundation for what Ansible would eventually become: "Ansible is an extra-simple Python API for doing 'remote things' over SSH. As Func, which I co-wrote, aspired to avoid using SSH and have its own daemon infrastructure, Ansible aspires to be quite different and more minimal, but still able to grow more modularly over time."

Since that first commit, and at the time of writing, there have been over 35,000 commits by 3,000 contributors over 38 branches and 195 releases.

In 2013, the project had grown and Ansible, Inc., was founded to offer commercial support to Ansible users who had relied on the project to manage both their instructors and servers, be they physical, virtual, or hosted on public clouds.

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

Then, in October 2015, Red Hat announced that they were to acquire Ansible for \$150 million. In the announcement, Joe Fitzgerald, who was Vice President,

Management, Red Hat at the time of the acquisition, was quoted as saying:
"Ansible is a clear leader in IT automation and DevOps, and helps Red Hat take
a significant step forward in our goal of creating frictionless IT."

During the course of this book, you will find that the statement in the original README file and Red Hat's statement at the time of acquiring Ansible both still ring true.

Before we look at rolling our sleeves up and installing Ansible, which we will be doing in the next chapter, we should look at some of the core concepts surrounding it.

Ansible versus other tools

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

- **Agentless:** Everything should be managed by the SSH daemon, the WinRM protocol in the case of Windows machines, or API calls—there should be no reliance on either custom agents or additional ports that need to be opened or interacted with on the target host
- **Minimal:** You should be able to manage new remote machines without having to install any new software as each Linux host will typically have at least SSH and Python installed as part of a minimal installation
- **Descriptive:** You should be able to describe your infrastructure, stack, or task in a language that is readable by both machines and humans
- **Simple:** The setup processes and learning curve should be simple and feel intuitive
- **Easy to use:** It should be the easiest IT automation system to use, ever

A few of these principles make Ansible quite different to other tools. Let's take a look at the most basic difference between Ansible and other tools, such as Puppet and Chef.

Declarative versus imperative

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

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

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

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

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

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

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

- **Run 1:** The tasks are executed in the following order: 2, 1, 3, and 4. This means that on the first run, as the group called `team` does not exist, adding the user `Alice` fails, which means that `Alice` is never given escalated

privileges. However, the group `team` is added and the user called `Bob` is added.

- **Run 2:** Again, the tasks are executed in the following order: 2, 1, 3, and 4. Because the group `team` was created during run 1, the user `Alice` is now created and she is also given escalated privileges. As the group `team` and user `Bob` already exist, they are left as is.
- **Run 3:** The tasks are executed in the same order as runs 1 and 2; however, as the desired configuration had been reached, no changes were made.

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

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

- **Run 1:** The tasks are executed in the order we defined them, meaning that the group is created, then the two users, and finally the escalated privileges of `Alice` are applied
- **Run 2:** Again, the tasks are executed in the order and checks are made to ensure that our desired configuration is in place

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

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

This is the position I found myself in before I started to use Ansible. Puppet was great at enforcing my desired end configuration; however, when it came to getting there, I found myself having to worry about building a lot of logic into my manifests to arrive at my desired state.

What was also annoying is that each successful run would take about 40 minutes to complete. But as I was having dependency issues, I had to start from scratch

with each failure and change to ensure that I was actually fixing the problem and not because things were starting to become consistent—not what you want when you are on a deadline.

Configuration versus orchestration

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

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

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

Infrastructure as code

Before we finish this chapter and move on to installing Ansible, let's quickly discuss infrastructure as code, first of all by looking at some actual code. The following bash script installs several RPMs using the `yum` package manager:

```
#!/bin/sh
LIST_OF_APPS="dstat lsof mailx rsync tree vim-enhanced git whois iptables-services"
yum install -y $LIST_OF_APPS
```

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

```
class common::apps {
  package{
    [
      'dstat',
      'lsof',
      'mailx',
      'rsync',
      'tree',
      'vim-enhanced',
      'git',
      'whois',
      'iptables-services',
    ]:
      ensure => installed,
  }
}
```

Next up, we have the same task using SaltStack:

```
common.packages:
  pkg.installed:
    - pkgs:
      - dstat
      - lsof
      - mailx
      - rsync
      - tree
      - vim-enhanced
      - git
      - whois
      - iptables-services
```

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

```
- name: install packages we need
  yum:
    name: "{{ item }}"
    state: "latest"
```

```
with_items:  
  - dstat  
  - lsof  
  - mailx  
  - rsync  
  - tree  
  - vim-enhanced  
  - git  
  - whois  
  - iptables-services
```

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

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

This approach to infrastructure management has a few advantages, one being that you, as system administrators, are using the same processes and tooling as your developer colleagues, meaning that any procedures that apply to them also apply to you. This makes for a more consistent working experience, as well as exposing you to tools that you may not have been exposed to or used before.

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

Summary

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

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

It is around this time, which was December 2014, that I decided to look at Ansible. I know this because I wrote a blog post entitled *First Steps With Ansible*, and since then, I don't think I have looked back. I have since introduced several of my work colleagues and customers to Ansible, as well as writing previous books for Packt.

In this chapter, we have taken a look at my own personal history with both Ansible and some of the other tools that Ansible is compared to, and we have discussed the differences between these tools and also where Ansible originated.

In the next chapter, we are going to look at installing Ansible and running our first playbooks against a local virtual machine.

Further reading

In this chapter, we mentioned Puppet and SaltStack:

- Puppet is a configuration management tool that runs a server/agent configuration. It comes in two flavors—the open source version and an enterprise version that is supported by Puppet, the company. It is a declarative system and is closely tied to Ruby. For more information on Puppet, see <https://puppet.com/>.
- SaltStack is another configuration management tool. It is extremely scalable and, while it shares a design approach with Ansible, it works in a similar way to Puppet in that it has a server/agent approach. You can find more information on SaltStack at <https://saltstack.com/>.
- I also mentioned my blog, which you can find at <https://media-glass.es/>.

Installing and Running Ansible

Now that we know a little about the background of Ansible, we will work on installing it and, once installed, we will run our first set of playbooks against a test virtual machine running CentOS 7.

The following topics will be covered in this chapter:

- How to install Ansible on macOS and Linux
- Running Ansible on Windows 10 Professional using the Linux subsystem
- Launching a test virtual machine
- An introduction to playbooks

Technical requirements

In this chapter, we will be installing Ansible, so you will need a machine capable of running it. I will go into more details about these requirements in the next part of the chapter. We will also be using Vagrant to launch a virtual machine locally. There is a section that walks through installing Vagrant as well as downloading a CentOS 7 Vagrant box, which is around 400 MB.

You can find complete versions of all of the playbooks in the GitHub repository that accompanies this book at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter02>.



You can also find the code bundle in author's repository: <https://github.com/russmckendrick/learn-ansible-fundamentals-of-ansible-2x>.

Installing Ansible

Let's dive straight in and install Ansible. Throughout this book, I will be assuming that you are running a macOS host machine running macOS High Sierra or a Linux machine with Ubuntu 18.04 LTS installed. While we will cover running Ansible on Windows 10 Professional using the Linux subsystem for Windows, using Windows as a host machine will not be supported in this book.

Installing on macOS

There are a few different ways you can install Ansible on your macOS High Sierra host machine. I will cover them both.



As we are discussing two different ways of installing Ansible, I would recommend reading through this section and also the Pros and cons section at the end before choosing which installation method to use on your own machine.

Homebrew

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



Homebrew is a package manager of macOS. It can be used to install command-line tools and also desktop packages. It describes itself as The missing package manager for macOS and it is normally one of the first tools I install after a clean installation or when getting a new computer. You can find out more about the project at <https://brew.sh/>.

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

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

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

Once installed, or if you already have Homebrew installed, run the following commands to update your list of packages and also check that your Homebrew installation is optimal:

```
| $ brew update  
| $ brew doctor
```

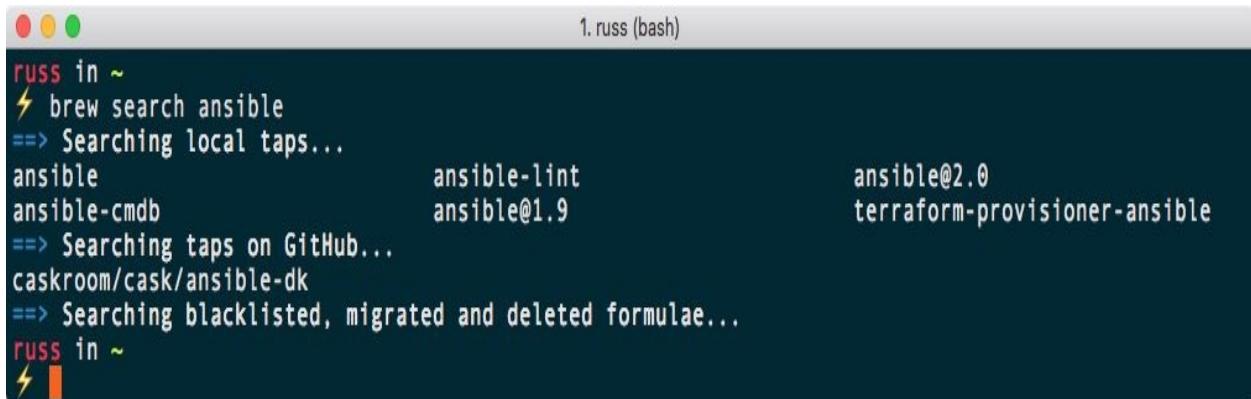
Depending on how new your installation is, or when you last used it, you might see a different output to the following screenshot:

```
russ in ~  
⚡ brew update  
Already up-to-date.  
russ in ~  
⚡ brew doctor  
Your system is ready to brew.  
russ in ~  
⚡
```

Next, we can check what packages Homebrew provides for Ansible by running the following command:

```
| $ brew search ansible
```

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

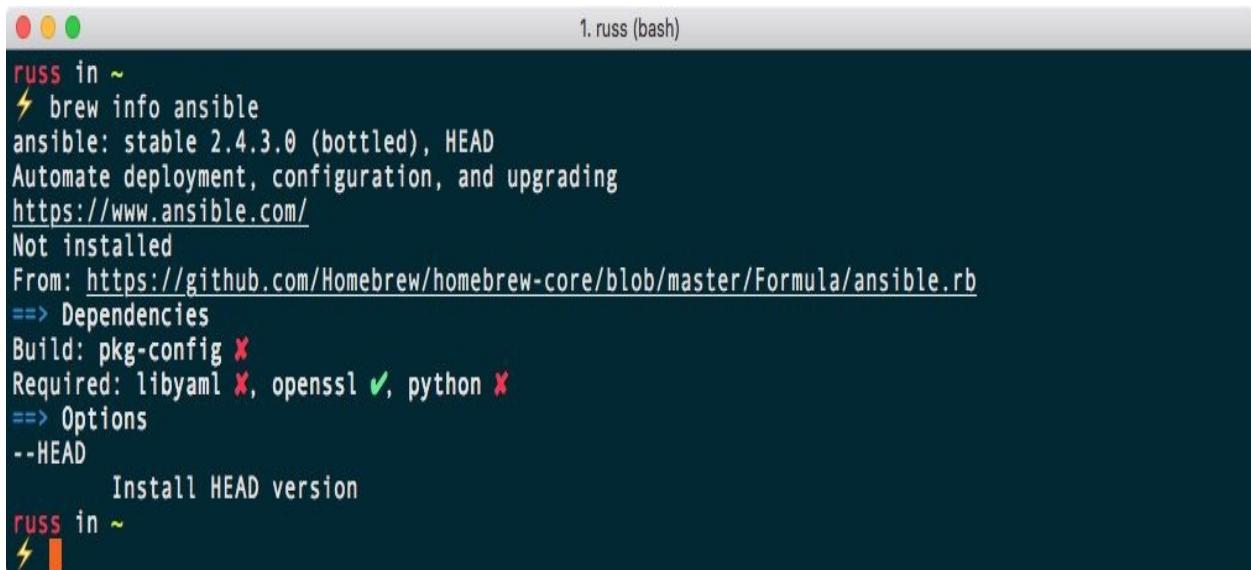


```
russ in ~
⚡ brew search ansible
==> Searching local taps...
ansible                  ansible-lint          ansible@2.0
ansible-cmdb             ansible@1.9        terraform-provisioner-ansible
==> Searching taps on GitHub...
caskroom/cask/ansible-dk
==> Searching blacklisted, migrated and deleted formulae...
russ in ~
⚡
```

We just want the Ansible package. You can find out more about the package by running the following command:

```
| $ brew info ansible
```

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



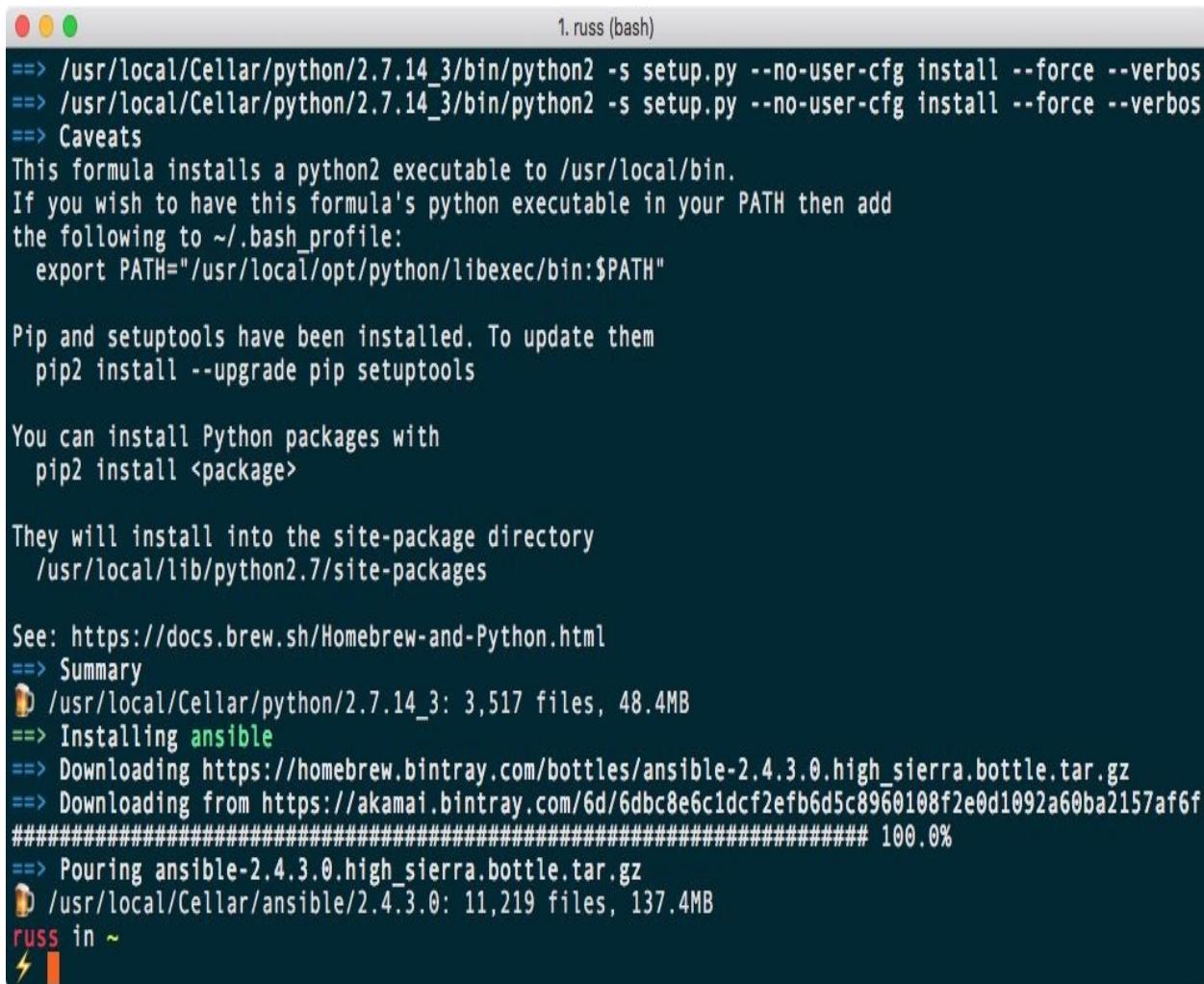
```
russ in ~
⚡ brew info ansible
ansible: stable 2.4.3.0 (bottled), HEAD
Automate deployment, configuration, and upgrading
https://www.ansible.com/
Not installed
From: https://github.com/Homebrew/homebrew-core/blob/master/Formula/ansible.rb
==> Dependencies
Build: pkg-config ✘
Required: libyaml ✘, openssl ✓, python ✘
==> Options
--HEAD
      Install HEAD version
russ in ~
⚡
```

As you can see, the command returns information on the version of the package that will be installed, as well as details on where you can see the code for the formula that installs the package. In our case, you can view details of the formula at <https://github.com/Homebrew/homebrew-core/blob/master/Formula/ansible.rb>.

To install Ansible using Homebrew, we simply have to run the following command:

```
| $ brew install ansible
```

This will download and install all of the dependencies and then the Ansible package itself. Depending on how much of the dependencies are already installed on your machine, this may take a few minutes. Once installed, you should see something like the following screenshot:



```
1. russ (bash)
==> /usr/local/Cellar/python/2.7.14_3/bin/python2 -s setup.py --no-user-cfg install --force --verbose
==> /usr/local/Cellar/python/2.7.14_3/bin/python2 -s setup.py --no-user-cfg install --force --verbose
==> Caveats
This formula installs a python2 executable to /usr/local/bin.
If you wish to have this formula's python executable in your PATH then add
the following to ~/.bash_profile:
  export PATH="/usr/local/opt/python/libexec/bin:$PATH"

Pip and setuptools have been installed. To update them
  pip2 install --upgrade pip setuptools

You can install Python packages with
  pip2 install <package>

They will install into the site-package directory
  /usr/local/lib/python2.7/site-packages

See: https://docs.brew.sh/Homebrew-and-Python.html
==> Summary
🍺 /usr/local/Cellar/python/2.7.14_3: 3,517 files, 48.4MB
==> Installing ansible
==> Downloading https://homebrew.bintray.com/bottles/ansible-2.4.3.0.high_sierra.bottle.tar.gz
==> Downloading from https://akamai.bintray.com/6d/6dbc8e6c1dcf2efb6d5c8960108f2e0d1092a60ba2157af6f
#####
==> Pouring ansible-2.4.3.0.high_sierra.bottle.tar.gz
🍺 /usr/local/Cellar/ansible/2.4.3.0: 11,219 files, 137.4MB
russ in ~
```

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

The pip method

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



*`pip` is a package manager for Python software. It is a recursive acronym for `pip install packages`. It is a good frontend for installing packages from the **Python Package Index (PyPI)**. You can find the index at <https://pypi.python.org/pypi/>.*

Depending on what you have installed on your machine, you may have to install `pip`. To do this, run the following command:

```
| $ easy_install pip
```

This will install `pip` using the `easy_install` installer, which ships with macOS by default. Once installed, you can install Ansible by running the following command:

```
| $ sudo -H pip install ansible
```

You will be prompted for your password, as we are using the `sudo` command, like Homebrew. This command will download and install all of the prerequisites needed to run Ansible on your system. While it is as verbose as Homebrew, its output contains information on what it has done, rather than hints on what do to next:

```
russ in ~
⚡ sudo -H pip install ansible
Collecting ansible
  Downloading ansible-2.4.3.0.tar.gz (6.5MB)
    100% |██████████| 6.5MB 193kB/s
Requirement already satisfied: jinja2 in /Library/Python/2.7/site-packages (from ansible)
Requirement already satisfied: PyYAML in /Library/Python/2.7/site-packages (from ansible)
Requirement already satisfied: paramiko in /Library/Python/2.7/site-packages (from ansible)
Requirement already satisfied: cryptography in /Library/Python/2.7/site-packages (from ansible)
Requirement already satisfied: setuptools in /Library/Python/2.7/site-packages (from ansible)
Requirement already satisfied: MarkupSafe>=0.23 in /Library/Python/2.7/site-packages (from jinja2->ansible)
Requirement already satisfied: pyasn1>=0.1.7 in /Library/Python/2.7/site-packages (from paramiko->ansible)
Requirement already satisfied: bcrypt>=3.1.3 in /Library/Python/2.7/site-packages (from paramiko->ansible)
Requirement already satisfied: pynacl>=1.0.1 in /Library/Python/2.7/site-packages (from paramiko->ansible)
Requirement already satisfied: six>=1.4.1 in /Library/Python/2.7/site-packages (from cryptography->ansible)
Requirement already satisfied: cffi>=1.7; platform_python_implementation != "PyPy" in /Library/Python/2.7/site-packages (from cryptography->ansible)
Requirement already satisfied: enum34; python_version < "3" in /Library/Python/2.7/site-packages (from cryptography->ansible)
Requirement already satisfied: idna>=2.1 in /Library/Python/2.7/site-packages (from cryptography->ansible)
Requirement already satisfied: asn1crypto>=0.21.0 in /Library/Python/2.7/site-packages (from cryptography->ansible)
Requirement already satisfied: ipaddress; python_version < "3" in /Library/Python/2.7/site-packages (from cryptography->ansible)
Requirement already satisfied: pycparser in /Library/Python/2.7/site-packages (from cffi>=1.7; platform_python_implementation != "PyPy"->cryptography->ansible)
Installing collected packages: ansible
  Running setup.py install for ansible ... done
Successfully installed ansible-2.4.3.0
russ in ~
⚡
```

As you can see, a lot of the requirements were already satisfied.

Pros and cons

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

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

```
| $ brew update  
| $ brew upgrade
```

Every once in a while, do update your installed packages to the latest. If you already do this, then it would make sense to use Homebrew to manage your Ansible installation.

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

```
| $ sudo -H pip install ansible --upgrade --ignore-installed setuptools
```

I have found that I need to use the `--ignore-installed setuptools` flag, as there are problems and conflicts with the version managed by macOS and the one which is part of Ansible updates. I have not known this to cause any problems.

Should you need to, then you can install older versions of Ansible using both Homebrew and `pip`. To do this using Homebrew, you just need to remove the current version by running the following command:

```
| $ brew uninstall ansible
```

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

```
| $ brew install ansible@2.0
```

Or, for an even earlier version, you can use the command:

```
| $ brew install ansible@1.9
```

For details on exactly which version of the package is going to be installed, you can run one of the following two commands:

```
| $ brew info ansible@2.0  
$ brew info ansible@1.9
```

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

```
| $ sudo -H pip install ansible==2.3.1.0 --ignore-installed setuptools
```

You should never need to do this. However, I have found that on some occasions, I have had to downgrade to help debug *quirks* in my playbooks introduced by upgrading to a later version.

As mentioned, I spend the bulk of my time in front of a macOS machine in some form or other, so which of the two methods do I use? Primarily, I use Homebrew, as I have several other tools installed using Homebrew. However, if I need to roll back to a previous version, I use `pip` and then return to Homebrew once the issue is resolved.

Installing on Linux

There are a few different ways of installing Ansible on Ubuntu 18.04. However, I am only going to cover one of them here. While there are packages available for Ubuntu from which you can install with `apt`, they tend to quickly become out of date and are typically behind the current release.



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

Because of this, we will be using `pip`. The first thing to do is install `pip`, and this can be done by running the following command: `$ sudo -H apt-get install python-pip`

Once `pip` is installed, the instructions for installing Ansible are the same as installing on macOS. Simply run the following command:

```
| $ sudo -H pip install ansible
```

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

```
russ@ubuntu:~$ sudo -H pip install ansible
Collecting ansible
  Downloading ansible-2.4.3.0.tar.gz (6.5MB)
    100% |██████████| 6.5MB 182kB/s
Collecting PyYAML (from ansible)
  Downloading PyYAML-3.12.tar.gz (253kB)
    100% |██████████| 256kB 3.0MB/s
Requirement already satisfied: cryptography in /usr/lib/python2.7/dist-packages
(from ansible)
Collecting jinja2 (from ansible)
  Downloading Jinja2-2.10-py2.py3-none-any.whl (126kB)
    100% |██████████| 133kB 4.0MB/s
Collecting paramiko (from ansible)
  Downloading paramiko-2.4.0-py2.py3-none-any.whl (192kB)
    100% |██████████| 194kB 3.2MB/s
Requirement already satisfied: setuptools in /usr/lib/python2.7/dist-packages (f
rom ansible)
Collecting MarkupSafe>=0.23 (from jinja2->ansible)
  Downloading MarkupSafe-1.0.tar.gz
Collecting pynacl>=1.0.1 (from paramiko->ansible)
  Downloading PyNaCl-1.2.1-cp27-cp27mu-manylinux1_x86_64.whl (696kB)
    100% |██████████| 706kB 1.6MB/s
Collecting pyasn1>=0.1.7 (from paramiko->ansible)
  Downloading pyasn1-0.4.2-py2.py3-none-any.whl (71kB)
```

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

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

Note that this time, we do not have to ignore anything as there shouldn't be any problems with the default installation. Also, downgrading Ansible is the same command:

```
| $ sudo -H pip install ansible==2.3.1.0 --ignore-installed setuptools
```

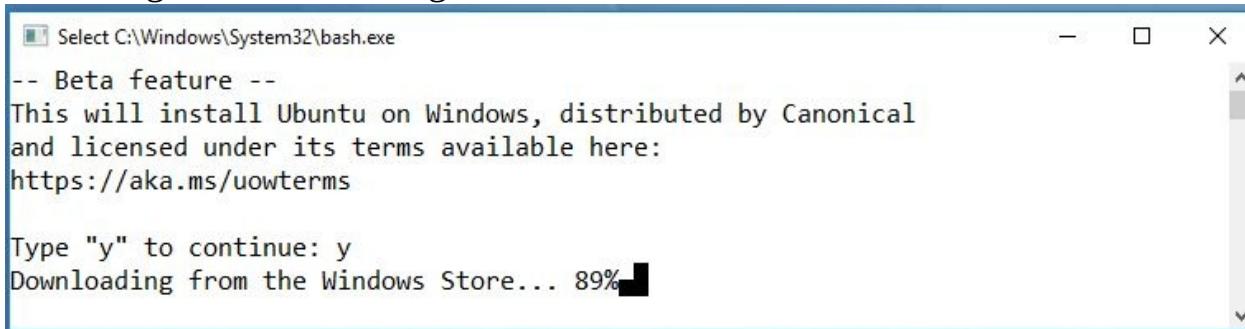
The preceding commands should work on most Linux distributions, such as CentOS, Red Hat Enterprise Linux, Debian, and, Linux Mint.

Installing on Windows 10 Professional

The last platform we are going to cover is Windows 10 Professional; well, sort of. There is no supported way of running an Ansible controller on a Windows machine natively. Because of this, we will be using the Windows subsystem for Linux.

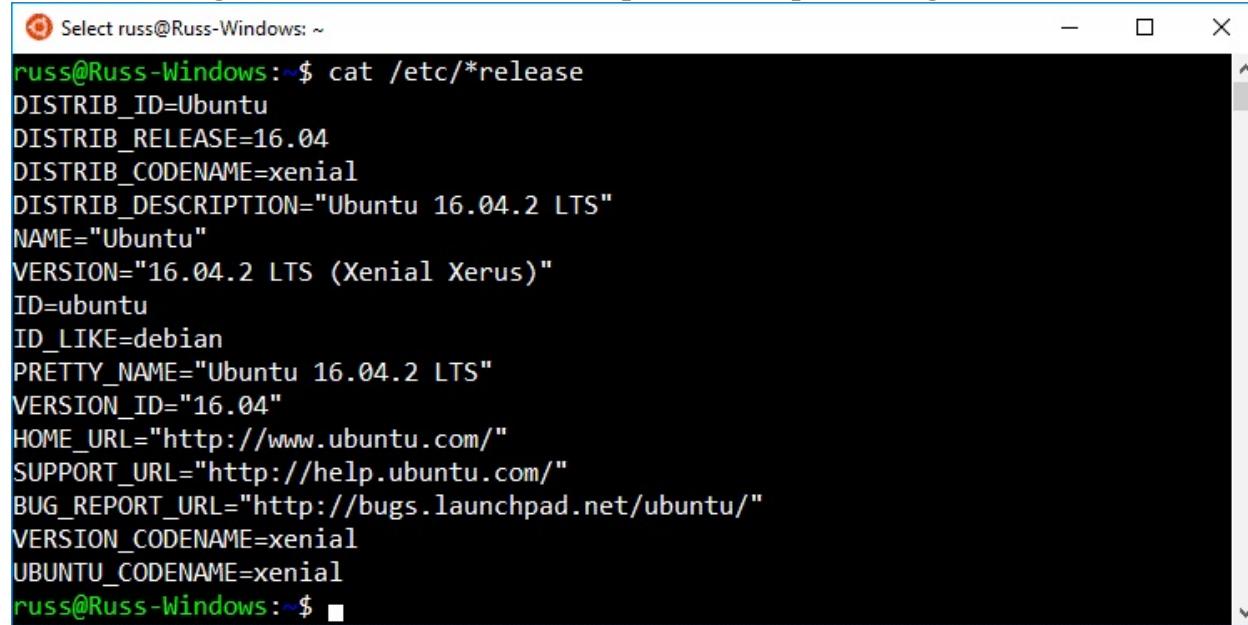
This is a feature, which, at the time of writing, is in beta, and is only available to Windows 10 Professional users. To enable it, you first need to enable developer mode. To do this, open the Windows 10 Settings app and then toggle Developer mode, which can be found under UPDATE & SECURITY, and then For Developers.

Once Developer mode has been enabled, you will be able to enable the shell. To do this, open Control Panel, then click on Programs and Features, and then on Turn Windows features on or off. In the list of features, you should see Windows Subsystem for Linux (Beta) listed. Tick the box next to it and then OK. You will be prompted to reboot your machine. Following the reboot, click on the Start menu and type `bash`. This will trigger the installation process. You should see something like the following screenshot:



Once downloaded, it will extract and install the subsystem. You will be asked a few questions. As needed, the entire process will take between 5 and 10 minutes. Once installed, you should now have a full Ubuntu 16.04 system running on your Windows machine. You can check this by running the following command:
`$ cat /etc/*release`

The following screenshot shows the output for the preceding command:



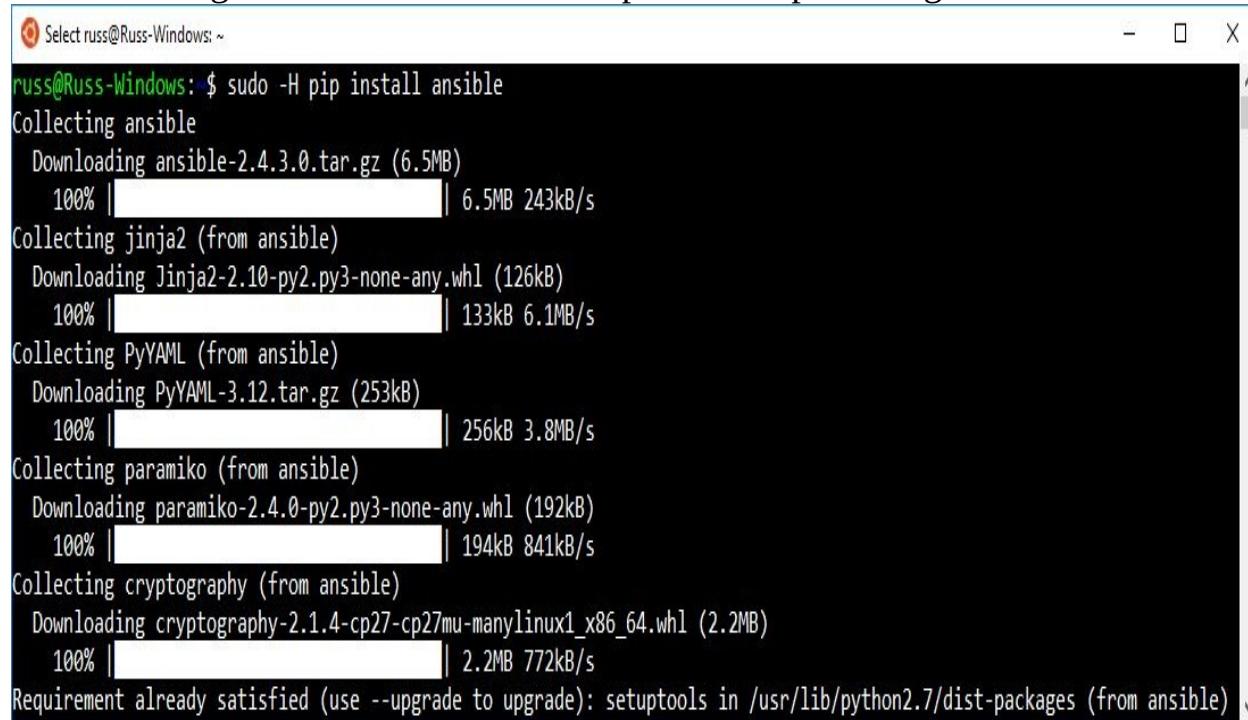
```
russ@Russ-Windows:~$ cat /etc/*release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=16.04
DISTRIB_CODENAME=xenial
DISTRIB_DESCRIPTION="Ubuntu 16.04.2 LTS"
NAME="Ubuntu"
VERSION="16.04.2 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.2 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
russ@Russ-Windows:~$
```

From here, you can run the following commands to install Ansible: **\$ sudo -H**

apt-get install python-pip

\$ sudo -H pip install ansible

The following screenshot shows the output for the preceding command:



```
russ@Russ-Windows: $ sudo -H pip install ansible
Collecting ansible
  Downloading ansible-2.4.3.0.tar.gz (6.5MB)
    100% |██████████| 6.5MB 243kB/s
Collecting jinja2 (from ansible)
  Downloading Jinja2-2.10-py2.py3-none-any.whl (126kB)
    100% |██████████| 133kB 6.1MB/s
Collecting PyYAML (from ansible)
  Downloading PyYAML-3.12.tar.gz (253kB)
    100% |██████████| 256kB 3.8MB/s
Collecting paramiko (from ansible)
  Downloading paramiko-2.4.0-py2.py3-none-any.whl (192kB)
    100% |██████████| 194kB 841kB/s
Collecting cryptography (from ansible)
  Downloading cryptography-2.1.4-cp27-cp27mu-manylinux1_x86_64.whl (2.2MB)
    100% |██████████| 2.2MB 772kB/s
Requirement already satisfied (use --upgrade to upgrade): setuptools in /usr/lib/python2.7/dist-packages (from ansible)
```

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



The Windows Subsystem for Linux (WSL) is not running on a virtual machine. It is a full native Linux experience baked right into Windows 10 Professional. It is targeted at developers who need to run Linux tools as part of their toolchain. While the overall support for Linux commands is excellent, I would recommend reading through the FAQs written and maintained by Microsoft to get an idea of the limits and also any quirks of the subsystem. The FAQs can be found at <https://docs.microsoft.com/en-us/windows/wsl/faq/>.

As already mentioned, while this is a viable way of running an Ansible control node on a Windows-based machine, some of the other tools we will be covering in future chapters may not work with Windows. So while you may follow along using the Ubuntu instructions, some parts may not work. Where possible, I will add a note saying that it may not work on Windows-based systems.

Launching a virtual machine

In order to launch a virtual machine to run our first set of Ansible commands against, we are going to use Vagrant.

Please note that these instructions may not work if you are running WSL.



Vagrant is a virtual machine manager developed by HashiCorp. It can manage both local and remote virtual machines, and supports hypervisors, such as VirtualBox, VMware, and Hyper-V.

To install Vagrant on macOS, we can use Homebrew along with cask. To install cask, run the following command:

```
| $ brew install cask
```



VirtualBox is an open source hypervisor for x86-based computers. It is currently being developed by Oracle and supports both software- and hardware-based virtualization.

By default, Vagrant uses VirtualBox. Once cask is installed, you can use VirtualBox and Vagrant by running the following command:

```
| $ brew cask install virtualbox vagrant
```

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

```
| $ sudo apt-get install virtualbox vagrant
```

Next up, if you don't already have one, we need to generate a private and public key for your user. To do this, run the following command, but if you already have a key, you can skip this part:

```
| $ ssh-keygen -t rsa -C "youremail@example.com"
```

This will create a key and store it in the `.ssh` folder in your user directory. We will use this key to inject into our Vagrant managed CentOS 7 virtual machine. To launch the virtual machine or box, as Vagrant calls it, we need a `vagrantfile`. This is the configuration that Vagrant uses to create and boot the box.

The `vagrantfile` we are going to be using is shown in the following code. You can also find a copy in the `chapter02` folder in the code examples that accompany this

book and also in the GitHub repository, which can be found at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter02>:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

API_VERSION = "2"
BOX_NAME     = "centos/7"
BOX_IP       = "192.168.50.4"
DOMAIN       = "nip.io"
PRIVATE_KEY  = "~/.ssh/id_rsa"
PUBLIC_KEY   = '~/.ssh/id_rsa.pub'

Vagrant.configure(API_VERSION) do |config|
  config.vm.box = BOX_NAME
  config.vm.network "private_network", ip: BOX_IP
  config.vm.host_name = BOX_IP + '.' + DOMAIN
  config.ssh.insert_key = false
  config.ssh.private_key_path = [PRIVATE_KEY,
    "~/.vagrant.d/insecure_private_key"]
  config.vm.provision "file", source: PUBLIC_KEY, destination:
    "~/.ssh/authorized_keys"

  config.vm.provider "virtualbox" do |v|
    v.memory = "2024"
    v.cpus = "2"
  end

  config.vm.provider "vmware_fusion" do |v|
    v.vmx["memsize"] = "2024"
    v.vmx["numvcpus"] = "2"
  end
end
```

As you can see from the preceding file, there are a few things going on. First of all, in the top section, we are defining a few variables for the following:

- **API_VERSION**: This is the Vagrant API version to use. This should stay at 2.
- **BOX_NAME**: This is the base image we want to use. In our case, this is the official CentOS 7 Vagrant box image, which can be found at <https://app.vagrantup.com/centos/boxes/7>.
- **BOX_IP**: This is the private IP address of the machine we are launching. Typically, you shouldn't need to hardcode the IP address, but in our case, we will need a resolvable hostname and also a fixed IP address for the examples in the next section of the chapter.
- **DOMAIN**: This is the domain name that is used to configure the hostname on the machine. We are using the <http://nip.io/> service. This provides free wildcard DNS entries. This means that our domain `192.168.50.4.nip.io` will resolve to `192.168.50.4`.
- **PRIVATE_KEY**: This is the path to your private key. This will be used to SSH

into the virtual machine once launched.

- `PUBLIC_KEY`: This is the path to your public key. When the machine is being launched, this will be injected into the host, which means that we can access it using our private key.

The next section takes the preceding values and configures the Vagrant box. We then define settings that are just for the two providers that the `vagrantfile` supports. As you can see, the file will launch a box using either VirtualBox or, if you have it installed, VMware Fusion.



For more information on the VMware provider plugin for Vagrant, visit <https://www.vagrantup.com/vmware>. Please note that this part of Vagrant requires a license, which is chargeable, as well as requiring you to have VMware Fusion or Workstation installed on your host machine.

Now that we have our `vagrantfile`, we just need to run the following command to launch the Vagrant box:

```
| $ vagrant up
```

If you do not pass a provider along, it will default to using VirtualBox. If you have the VMware plugin, like me, you can run the following command:

```
| $ vagrant up --provider=vmware_fusion
```

It will take a few minutes while the appropriate box file is downloaded and the virtual machine is configured:

```
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ vagrant up --provider=vmware_fusion
Bringing machine 'default' up with 'vmware_fusion' provider...
==> default: Cloning VMware VM: 'centos/7'. This can take some time...
==> default: Checking if box 'centos/7' is up to date...
==> default: Verifying vmnet devices are healthy...
==> default: Preparing network adapters...
WARNING: The VMX file for this box contains a setting that is automatically overwritten by Vagrant
WARNING: when started. Vagrant will stop overwriting this setting in an upcoming release which may
WARNING: prevent proper networking setup. Below is the detected VMX setting:
WARNING:
WARNING:   ethernet0.pcislotnumber = "32"
WARNING:
WARNING: If networking fails to properly configure, it may require this VMX setting. It can be manua-
lly
WARNING: applied via the Vagrantfile:
WARNING:
WARNING:   Vagrant.configure(2) do |config|
WARNING:     config.vm.provider :vmare_fusion do |vmware|
WARNING:       vmware.vmx["ethernet0.pcislotnumber"] = "32"
WARNING:     end
WARNING:   end
WARNING:
WARNING: For more information: https://www.vagrantup.com/docs/vmware/boxes.html#vmx-whitelisting
==> default: Starting the VMware VM...
==> default: Waiting for the VM to receive an address...
==> default: Forwarding ports...
  default: -- 22 => 2222
==> default: Waiting for machine to boot. This may take a few minutes...
  default: SSH address: 127.0.0.1:2222
  default: SSH username: vagrant
  default: SSH auth method: private key
  default:
  default: Vagrant insecure key detected. Vagrant will automatically replace
  default: this with a newly generated keypair for better security.
  default:
  default: Inserting generated public key within guest...
  default: Removing insecure key from the guest if it's present...
  default: Key inserted! Disconnecting and reconnecting using new SSH key...
==> default: Machine booted and ready!
==> default: Setting hostname...
==> default: Configuring network adapters within the VM...
  default: SSH address: 127.0.0.1:2222
  default: SSH username: vagrant
  default: SSH auth method: private key
==> default: Rsyncing folder: /Users/russ/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02/ => /vagrant
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡
```

As you can see from the Terminal output, the launch process is very verbose and you receive informative feedback at each stage.

Once the Vagrant box is launched, you can check connectivity to it by running the following commands. This will log you into the Vagrant box as the Vagrant user and check the details on the hostname and kernel:

```
$ vagrant ssh
$ hostname
$ uname -a
$ exit
```

Your Terminal should look like the following:



1. chapter02 (bash)

```
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ vagrant ssh
Last login: Sun Feb 11 15:18:22 2018 from 172.16.20.2
[vagrant@192 ~]$ hostname
192.168.50.4.nip.io
[vagrant@192 ~]$ uname -a
Linux 192.168.50.4.nip.io 3.10.0-693.11.6.el7.x86_64 #1 SMP Thu Jan 4 01:06:37 UTC 2018 x86_64 x86_6
4 x86_64 GNU/Linux
[vagrant@192 ~]$ exit
logout
Connection to 127.0.0.1 closed.
```

As you can see, because we have told Vagrant which private key to use when accessing the box, we have been let straight into the box and can run the commands without issue. However, we won't be using the `vagrant ssh` command in the next section, which is why we needed to inject our public key into the host. Instead, we will be SSHing directly into the machine from our host machine. To test this, you should be able to run the following command:

```
| $ ssh vagrant@192.168.50.4.nip.io
```

You should be asked to establish the authenticity of the host by typing `yes`. Once you are logged in, you will be able to run the following commands:

```
| $ hostname
| $ uname -a
| $ exit
```

Your Terminal should look like the following:



1. chapter02 (bash)

```
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ ssh vagrant@192.168.50.4.nip.io
The authenticity of host '192.168.50.4.nip.io (192.168.50.4)' can't be established.
ECDSA key fingerprint is SHA256:28Wmps8lrENYuOYSc+EcruiyUfOLcZOkNr0rLmanbdA.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.50.4.nip.io,192.168.50.4' (ECDSA) to the list of known hosts.
Last login: Sun Feb 11 15:18:52 2018 from 192.168.50.1
[vagrant@192 ~]$ hostname
192.168.50.4.nip.io
[vagrant@192 ~]$ uname -a
Linux 192.168.50.4.nip.io 3.10.0-693.11.6.el7.x86_64 #1 SMP Thu Jan 4 01:06:37 UTC 2018 x86_64 x86_6
4 x86_64 GNU/Linux
[vagrant@192 ~]$ exit
logout
Connection to 192.168.50.4.nip.io closed.
```

As you can see, we have resolved and connected to `192.168.50.4.nip.io` using the Vagrant user and have authenticated using our private key. Before we move on to the next section and try running Ansible for the first time, we should discuss Vagrant provisioners.

No doubt you will have already looked at the Vagrant website, which can be found at <http://vagrantup.com/>, and may have spotted that Vagrant actually supports Ansible out of the box. If we were to use the Ansible provisioner, then Vagrant would dynamically create a host inventory and run our playbook against the box as part of the launch process. Before we look at this, I believe it is important for us to understand how the host inventory works, so we will look at the Ansible provisioner in the next chapter.

Before then, however, let's take a look at some basic playbooks and how we can interact with our Vagrant box using Ansible.

An introduction to playbooks

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

In the traditional sense, a playbook is typically a collection of scripts or instructions for a user to follow and, while they are meant to introduce consistency and conformity across systems, even with the best intentions, this is almost never the case.

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

Before we run a playbook, let's discuss how we provide Ansible with a list of hosts to target. To do this, we will be using the `setup` command. This simply connects to a host and then fetches as much information on the host as it can.

Host inventories

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

In its simplest form, our hosts file could contain a single line:

```
| 192.168.50.4.nip.io ansible_user=vagrant
```

What this is telling Ansible is that the host we want to contact is `192.168.50.4.nip.io` and to use the username of `vagrant`. If we didn't provide the username, it would fall back to the user you are logged into your Ansible control host as, which in my case is simple—the user `russ`, which does not exist on the Vagrant box. There is a copy of the hosts file called `hosts-simple` in the `Chapter02` folder of the repository alongside the `vagrantfile` we used to launch the Vagrant box.

To run the `setup` command, we need to run the following command from within the same folder where `hosts-simple` is stored:

```
| $ ansible -i hosts-simple 192.168.50.4.nip.io -m setup
```

You should see some output which looks like the following:

1. chapter02 (bash)

```
⚡ ansible -i hosts-simple 192.168.50.4.nip.io -m setup
192.168.50.4.nip.io | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "192.168.50.4",
            "172.16.20.132"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::20c:29ff:fe70:2511",
            "fe80::20c:29ff:fe70:2507"
        ],
        "ansible_apparmor": {
            "status": "disabled"
        },
        "ansible_architecture": "x86_64",
        "ansible_bios_date": "05/19/2017",
        "ansible_bios_version": "6.00",
        "ansible_cmdline": {
            "BOOT_IMAGE": "/vmlinuz-3.10.0-693.11.6.el7.x86_64",
            "biosdevname": "0",
            "console": "ttyS0,115200n8",
            "crashkernel": "auto",
            "net.ifnames": "0",
            "no_timer_check": true,
            "quiet": true,
            "rd.lvm.lv": "VolGroup00/LogVol01",
            "rhgb": true,
            "ro": true,
            "root": "/dev/mapper/VolGroup00-LogVol00"
        },
        "ansible_date_time": {
            "date": "2018-02-17",
            "day": "17",
            "epoch": "1518891516",
            "hour": "18",
            "iso8601": "2018-02-17T18:18:36Z",
            "iso8601_basic": "20180217T181836Z",
            "iso8601_basic_short": "20180217T181836",
            "iso8601_micro": "2018-02-17T18:18:36.309471Z",
            "minute": "18",
            "month": "02",
            "second": "36",
            "time": "18:18:36",
            "tz": "UTC",
            "tz_offset": "+0000",
            "weekday": "Saturday",
            "weekday_number": "6",
            "weeknumber": "07",
            "year": "2018"
        }
    }
}
```

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

Going back to the command we ran:

```
| $ ansible -i hosts-simple 192.168.50.4.nip.io -m setup
```

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

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

```
| $ ansible --inventory=hosts-simple 192.168.50.4.nip.io --module-name=setup
```

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

```
box ansible_host=192.168.50.4.nip.io

[boxes]
box

[boxes:vars]
ansible_connection=ssh
ansible_user=vagrant
ansible_private_key_file=~/ssh/id_rsa
host_key_checking=False
```

There is a copy of the file, called just `hosts`, in the same folder as the `hosts-simple` file. As you can see, there is a lot more going on, so let's quickly work through it from top to bottom.

The first line defines our individual host. Unlike the simple example, we are going to be calling our `hosts` box and using `ansible_host`, so we are giving Ansible details of where it can SSH to. This means that we can now use the name `box` when referring to `192.168.50.4.nip.io`. This means our command would now look something like this:

```
| $ ansible -i hosts box -m setup
```

Next up in the file, we are creating a group of hosts called `boxes` and, in that group, we are adding our single host `box`. This means that we can also run:

```
| $ ansible -i hosts boxes -m setup
```

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

We will be revisiting the inventory host files in later chapters. From now on, we will be using the `hosts` file to target the `boxes` group.

Playbooks

In the previous section, running the `ansible` command allowed us to call a single module. In this section, we are going to look at calling several modules. The following playbook is called `playbook.yml`. It calls the `setup` module we called in the previous section and then uses the `debug` module to print a message to the screen:

```
---
- hosts: boxes
  gather_facts: true
  become: yes
  become_method: sudo

  tasks:
    - debug:
        msg: "I am connecting to {{ ansible_nodename }} which is running {{ ansible_distribution }} {{ ansible_distribution_version }}"
```

Before we start to break the configuration down, let's take a look at the results of running the playbook. To do this, use the following command:

```
| $ ansible-playbook -i hosts playbook01.yml
```

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

```
1. chapter02 (bash)
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ ansible-playbook -i hosts playbook01.yml

PLAY [boxes] ****
TASK [Gathering Facts] ****
ok: [box]

TASK [debug] ****
ok: [box] => {
    "msg": "I am connecting to 192.168.50.4.nip.io which is running CentOS 7.4.1708"
}

PLAY RECAP ****
box : ok=2    changed=0    unreachable=0    failed=0

russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡
```

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

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

```
|---
```

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

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

```
- hosts: boxes
gather_facts: true
become: yes
become_method: sudo
tasks:
```

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

- `hosts`: This tells Ansible the host or host group to target in the playbook. This must be defined in a host inventory like the ones we covered in the previous section.
- `gather_facts`: This tells Ansible to run the `setup` module when it first connects to the host. This information is then available to the playbook during the remainder of the run.
- `become`: This is present because we are connecting to our host as a basic user. In this case, the Vagrant user. Ansible may not have enough access privileges to execute some of the commands we are telling it to so this instructs Ansible to execute all of its commands as the root user.
- `become_method`: This tells Ansible how to become the root user; in our case, we have a passwordless `sudo` configured by Vagrant so we are using `sudo`.
- `tasks`: These are the tasks we can tell Ansible to run when connected to the target host.

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

```
| - debug:  
|   msg: "I am connecting to {{ ansible_nodename }} which is running {{  
|     ansible_distribution }} {{ ansible_distribution_version }}"
```

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

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

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

```
| ---  
| - hosts: boxes  
|   gather_facts: true
```

```
become: yes
become_method: sudo

tasks:
  - debug:
      msg: "I am connecting to {{ ansible_nodename }} which is running {{ ansible_distribution }} {{ ansible_distribution_version }}"
  - yum:
      name: "*"
      state: "latest"
```

As you can see, we have added a second task which calls the `yum` module. This module is designed to help us interact with the package manager used by CentOS and other Red Hat-based operating systems called `yum`. We are setting two key values here:

- `name`: This is a wildcard. It tells Ansible to use all of the installed packages rather than just a single named package. For example, we could have just used something like `HTTPD` here to target just Apache.
- `state`: Here, we are telling Ansible to ensure the package we have defined in the `name` key is the `latest` version. As we have named all of the installed packages, this will update everything we have installed.

Run the playbook using:

```
| $ ansible-playbook -i hosts playbook02.yml
```

This will give us the following results:

```
1. chapter02 (bash)
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ ansible-playbook -i hosts playbook02.yml

PLAY [boxes] ****
TASK [Gathering Facts] ****
ok: [box]

TASK [debug] ****
ok: [box] => {
    "msg": "I am connecting to 192.168.50.4.nip.io which is running CentOS 7.4.1708"
}

TASK [yum] ****
changed: [box]

PLAY RECAP ****
box : ok=3    changed=1    unreachable=0    failed=0

russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ |
```

The `yum` task has been marked as `changed` on the host `box`. This means that packages were updated. Running the same command again shows the following:

```
1. chapter02 (bash)
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ ansible-playbook -i hosts playbook02.yml

PLAY [boxes] ****
TASK [Gathering Facts] ****
ok: [box]

TASK [debug] ****
ok: [box] => {
    "msg": "I am connecting to 192.168.50.4.nip.io which is running CentOS 7.4.1708"
}

TASK [yum] ****
ok: [box]

PLAY RECAP ****
box : ok=3    changed=0    unreachable=0    failed=0

russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ |
```

As you can see, the `yum` task is now showing as `ok` on our host. This is because there are currently no longer any packages requiring updating.

Before we finish this quick look at playbooks, let's do something more interesting. The following playbook, called `playbook03.yml`, adds installing, configuring, and starting the NTP service to our Vagrant box. It also adds a few new sections to our playbook as well as using a template:

```

---  

- hosts: boxes  

gather_facts: true  

become: yes  

become_method: sudo  

vars:  

ntp_servers:  

- "0.centos.pool.ntp.org"  

- "1.centos.pool.ntp.org"  

- "2.centos.pool.ntp.org"  

- "3.centos.pool.ntp.org"  

handlers:  

- name: "restart ntp"  

service:  

name: "ntpd"  

state: "restarted"  

tasks:  

- debug:  

msg: "I am connecting to {{ ansible_nodename }} which is  

running {{ ansible_distribution }}  

{{ ansible_distribution_version }}"  

- yum:  

name: "*"  

state: "latest"  

- yum:  

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

with_items:  

- "ntp"  

- "ntpdate"  

- template:  

src: "./ntp.conf.j2"  

dest: "/etc/ntp.conf"  

notify: "restart ntp"

```

Before we work through the additions to our playbook, let's run it to get an idea of the feedback you get from Ansible:

```
| $ ansible-playbook -i hosts playbook03.yml
```

The following screenshot shows the output for the preceding command:

```
1. chapter02 (bash)
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ ansible-playbook -i hosts playbook03.yml

PLAY [boxes] ****
TASK [Gathering Facts] ****
ok: [box]

TASK [debug] ****
ok: [box] => {
    "msg": "I am connecting to 192.168.50.4.nip.io which is running CentOS 7.4.1708"
}

TASK [yum] ****
ok: [box]

TASK [yum] ****
changed: [box] => (item=[u'ntp', u'ntpd'])

TASK [template] ****
changed: [box]

RUNNING HANDLER [restart ntp] ****
changed: [box]

PLAY RECAP ****
box : ok=6    changed=3    unreachable=0    failed=0

russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡
```

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

```
1. chapter02 (bash)
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ ansible-playbook -i hosts playbook03.yml

PLAY [boxes] ****

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

TASK [debug] ****
ok: [box] => {
    "msg": "I am connecting to 192.168.50.4.nip.io which is running CentOS 7.4.1708"
}

TASK [yum] ****
ok: [box]

TASK [yum] ****
ok: [box] => (item=[u'ntp', u'ntpd'])

TASK [template] ****
ok: [box]

PLAY RECAP ****
box : ok=5    changed=0    unreachable=0    failed=0

russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡
```

As expected, because we haven't changed the playbook or anything on the Vagrant box, there are no changes and Ansible is reporting everything as `ok`.

Let's go back to our playbook and discuss the additions. You will notice that we have added two new sections, `vars` and `handlers`, as well as two new tasks, a second task which uses the `yum` module and the final task, which utilizes the `template` module.

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

```
vars:  
  ntp_servers:  
    - "0.centos.pool.ntp.org"  
    - "1.centos.pool.ntp.org"  
    - "2.centos.pool.ntp.org"  
    - "3.centos.pool.ntp.org"
```

As you can see, we are actually providing four different values for the same key. These will be used in the `template` task. We could have also written this as follows:

```
vars:  
  ntp_servers: [ "0.centos.pool.ntp.org", "1.centos.pool.ntp.org",  
    "2.centos.pool.ntp.org", "3.centos.pool.ntp.org" ]
```

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

```
handlers:  
  - name: "restart ntp"  
    service:  
      name: "ntpd"  
      state: "restarted"
```

In our case, the `restart ntp` handler uses the `service` module to restart `ntpd`. Next up, we have the two new tasks, starting with one which installs the NTP service and also the `ntpdate` package using `yum`:

```
- yum:  
  name: "{{ item }}"  
  state: "installed"  
  with_items:  
    - "ntp"  
    - "ntpdate"
```

As we are installing two packages, we need a way to provide two different package names to the `yum` module so that we don't have to have two different tasks for each of the package installations. To achieve this, we are using the `with_items` command, as part of the task section. Note that this is in addition to the `yum` module and is not part of the module—you can tell this by the indentation.

The `with_items` command allows you to provide a variable or list of items to the task. Wherever `{{ item }}` is used, it will be replaced with the content of the `with_items` value.

The final addition to the playbook is the following task:

```
| - template:  
|   src: "./ntp.conf.j2"  
|   dest: "/etc/ntp.conf"  
|   notify: "restart ntp"
```

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

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

```
| # {{ ansible_managed }}  
| driftfile /var/lib/ntp/drift  
| restrict default nomodify notrap nopeer noquery  
| restrict 127.0.0.1  
| restrict ::1  
| {% for item in ntp_servers %}  
| server {{ item }} iburst  
| {% endfor %}  
| includefile /etc/ntp/crypto/pw  
| keys /etc/ntp/keys  
| disable monitor
```

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

```
| # {{ ansible_managed }}
```

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

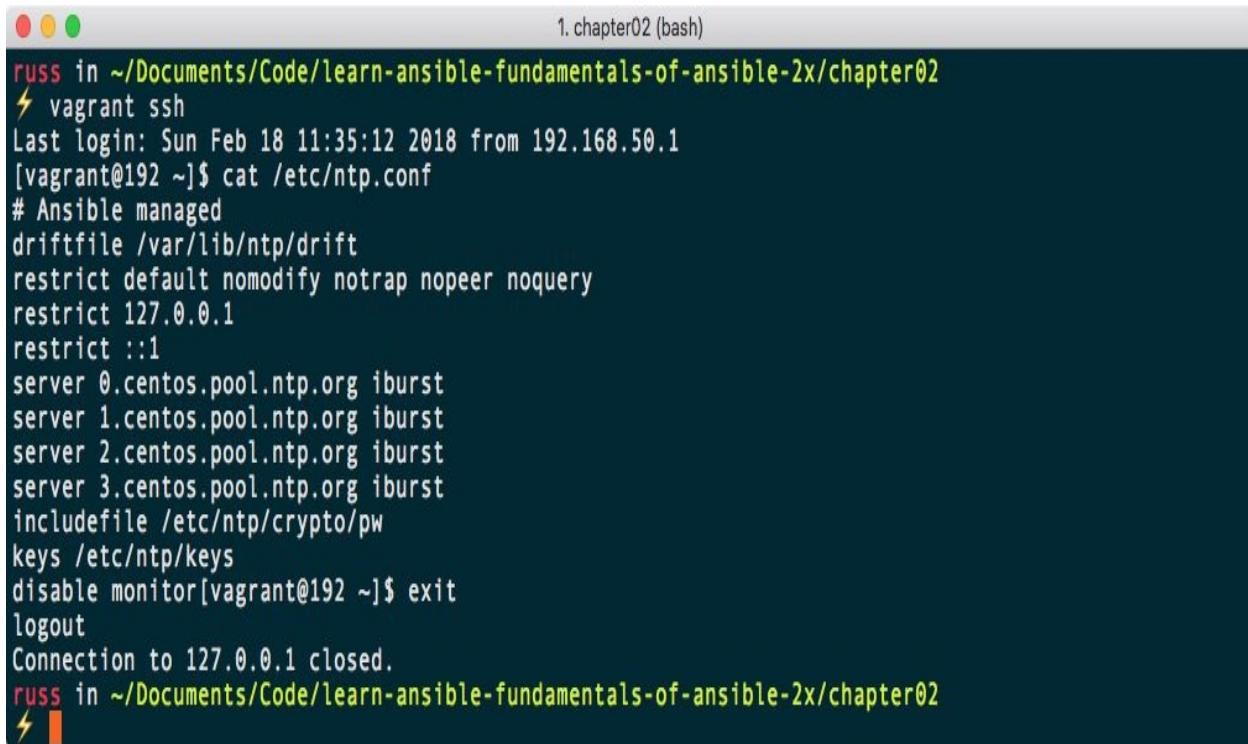
The next part loops through the `ntp_servers` values we defined in the `vars` section of the playbook:

```
| {% for item in ntp_servers %}  
|   server {{ item }} iburst  
| {% endfor %}
```

For each of the values, add a line that contains the server, then the value, and then `iburst`. You can see the output of this by SSHing into the Vagrant machine and opening `/etc/ntp.conf`:

```
| $ vagrant ssh  
| $ cat /etc/ntp.conf
```

The following screenshot shows the output for the preceding command:



```
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02  
⚡ vagrant ssh  
Last login: Sun Feb 18 11:35:12 2018 from 192.168.50.1  
[vagrant@192 ~]$ cat /etc/ntp.conf  
# Ansible managed  
driftfile /var/lib/ntp/drift  
restrict default nomodify notrap nopeer noquery  
restrict 127.0.0.1  
restrict ::1  
server 0.centos.pool.ntp.org iburst  
server 1.centos.pool.ntp.org iburst  
server 2.centos.pool.ntp.org iburst  
server 3.centos.pool.ntp.org iburst  
includefile /etc/ntp/crypto/pw  
keys /etc/ntp/keys  
disable monitor[vagrant@192 ~]$ exit  
logout  
Connection to 127.0.0.1 closed.  
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02  
⚡
```

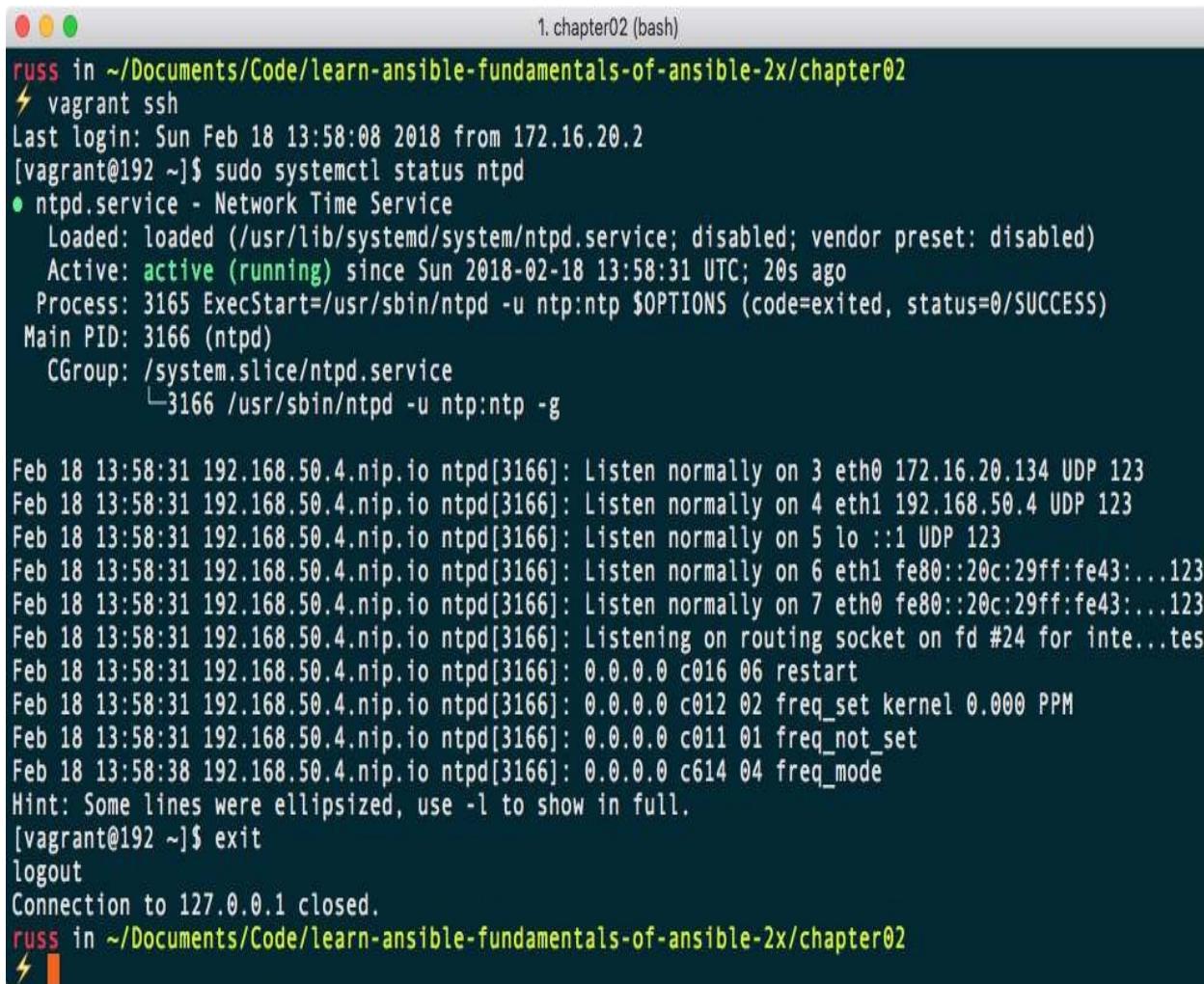
As you can see from the preceding screenshot of the fully rendered file, we have the comment on the first line noting that the file is managed by Ansible and also the four lines containing the NTP servers to use.

Finally, you can check that NTP is running as expected by running the following commands:

```
| $ vagrant ssh
```

```
| $ sudo systemctl status ntpd
```

The following screenshot shows the output for the preceding command:



```
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ vagrant ssh
Last login: Sun Feb 18 13:58:08 2018 from 172.16.20.2
[vagrant@192 ~]$ sudo systemctl status ntpd
● ntpd.service - Network Time Service
  Loaded: loaded (/usr/lib/systemd/system/ntp.service; disabled; vendor preset: disabled)
  Active: active (running) since Sun 2018-02-18 13:58:31 UTC; 20s ago
    Process: 3165 ExecStart=/usr/sbin/ntpd -u ntp:ntp $OPTIONS (code=exited, status=0/SUCCESS)
   Main PID: 3166 (ntpd)
     CGroup: /system.slice/ntp.service
             └─3166 /usr/sbin/ntpd -u ntp:ntp -g

Feb 18 13:58:31 192.168.50.4.nip.io ntpd[3166]: Listen normally on 3 eth0 172.16.20.134 UDP 123
Feb 18 13:58:31 192.168.50.4.nip.io ntpd[3166]: Listen normally on 4 eth1 192.168.50.4 UDP 123
Feb 18 13:58:31 192.168.50.4.nip.io ntpd[3166]: Listen normally on 5 lo ::1 UDP 123
Feb 18 13:58:31 192.168.50.4.nip.io ntpd[3166]: Listen normally on 6 eth1 fe80::20c:29ff:fe43:...123
Feb 18 13:58:31 192.168.50.4.nip.io ntpd[3166]: Listen normally on 7 eth0 fe80::20c:29ff:fe43:...123
Feb 18 13:58:31 192.168.50.4.nip.io ntpd[3166]: Listening on routing socket on fd #24 for interface eth0
Feb 18 13:58:31 192.168.50.4.nip.io ntpd[3166]: 0.0.0.0 c016 06 restart
Feb 18 13:58:31 192.168.50.4.nip.io ntpd[3166]: 0.0.0.0 c012 02 freq_set kernel 0.000 PPM
Feb 18 13:58:31 192.168.50.4.nip.io ntpd[3166]: 0.0.0.0 c011 01 freq_not_set
Feb 18 13:58:38 192.168.50.4.nip.io ntpd[3166]: 0.0.0.0 c614 04 freq_mode
Hint: Some lines were ellipsized, use -l to show in full.
[vagrant@192 ~]$ exit
logout
Connection to 127.0.0.1 closed.
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡
```

As you can see from the preceding output, NTP is loaded and running as expected. Let's remove our Vagrant box and launch a fresh one by running the following command:

```
| $ vagrant destroy
```

Then launch the box again by running one of the following two commands:

```
| $ vagrant up
| $ vagrant up --provider=vmware_fusion
```

Once the box is up and running, we can run the final playbook with:

```
| $ ansible-playbook -i hosts playbook03.yml
```

After a minute or two, you should receive the results of the playbook run. You should see five `changed` and six `ok`:

```
● ● ● 1. chapter02 (bash)
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ ansible-playbook -i hosts playbook03.yml

PLAY [boxes] ****
TASK [Gathering Facts] ****
ok: [box]

TASK [debug] ****
ok: [box] => {
    "msg": "I am connecting to 192.168.50.4.nip.io which is running CentOS 7.4.1708"
}

TASK [yum] ****
changed: [box]

TASK [yum] ****
changed: [box] => (item=[u'ntp', u'ntpdate'])

TASK [template] ****
changed: [box]

RUNNING HANDLER [restart ntp] ****
changed: [box]

PLAY RECAP ****
box : ok=6    changed=4    unreachable=0    failed=0

russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ |
```

Running for the second time will just show five `ok`:

```
1. chapter02 (bash)
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡ ansible-playbook -i hosts playbook03.yml

PLAY [boxes] ****
TASK [Gathering Facts] ****
ok: [box]

TASK [debug] ****
ok: [box] => {
    "msg": "I am connecting to 192.168.50.4.nip.io which is running CentOS 7.4.1708"
}

TASK [yum] ****
ok: [box]

TASK [yum] ****
ok: [box] => (item=[u'ntp', u'ntpd'])

TASK [template] ****
ok: [box]

PLAY RECAP ****
box : ok=5    changed=0    unreachable=0    failed=0

russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter02
⚡
```

The reason why we got six `ok` on the first run and five `ok` on the second run is that nothing has changed since the first run. Therefore, the handler to restart NTP is never being notified so that task to restart the service never executes.

Once you have finished with the example playbooks, you can terminate the running box using:

```
| $ vagrant destroy
```

We will be using the box again in the following chapter.

Summary

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

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

In the next chapter, we are going to take a look at other Ansible commands we can use.

Questions

1. What is the command to install Ansible using `pip`?
2. True or false: You can choose exactly which version of Ansible to install or roll back to when using Homebrew.
3. True or false: The Windows Subsystem for Linux runs in a virtual machine.
4. Name three hypervisors that are supported by Vagrant.
5. State and explain what a host inventory is.
6. True or false: Indentation in YAML files is extremely important to their execution and isn't just cosmetic.
7. Update the final playbook to install a service of your choice and have it notify a handler to start the service with its default configuration.

Further reading

In this chapter, we used the following Ansible modules, and you can find out more information on each module in the following links:

- `setup`: http://docs.ansible.com/ansible/latest/setup_module.html
- `debug`: http://docs.ansible.com/ansible/latest/debug_module.html
- `yum`: http://docs.ansible.com/ansible/latest/yum_module.html
- `service`: http://docs.ansible.com/ansible/latest/service_module.html

The Ansible Commands

Before we move on to writing and executing more advanced playbooks, we are going to take a look at the inbuilt Ansible commands. Here, we will cover the usage of the set of commands that makes up Ansible. Towards the end of the chapter, we will also install a few third-party tools, one being an inventory grapher, which will let us visualize our hosts, and the second allows you to record your playbook runs.

The following topics will be covered in this chapter:

- Inbuilt commands:

- ansible
- ansible-config
- ansible-console
- ansible-doc
- ansible-inventory
- ansible-vault

- Third-party commands:

- ansible-inventory-grapher
- ara

Technical requirements

We will be reusing the Vagrant box we launched in the previous chapter; if you have not been following along, please refer to the previous chapter for instructions on how to install both Ansible and Vagrant. There are a few playbook examples throughout this chapter; you can find the complete examples at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter03>.

Inbuilt commands

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

- `ansible`
- `ansible-config`
- `ansible-console`
- `ansible-doc`
- `ansible-inventory`
- `ansible-vault`

We will be covering a few of the commands, such as `ansible-galaxy`, `ansible-playbook`, and `ansible-pull`, in later chapters, so I will not go into any detail about those commands in this chapter. Let's make a start at the top of the list with a command we have already used.

Ansible

Now, you would have thought that `ansible` is going to be the most common command we will be using throughout this book, but it isn't.

The `ansible` command is really only ever used for running ad hoc commands again in a single or collection of hosts. In the last chapter, we created a host inventory file that targeted a single local virtual machine. For this chapter, let's take a look at targeting four different hosts running in DigitalOcean; my hosts file looks as follows:

```
ansible01 ansible_host=46.101.92.240
ansible02 ansible_host=159.65.63.218
ansible03 ansible_host=159.65.63.217
ansible04 ansible_host=138.68.145.116

[London]
ansible01
ansible02

[NYC]
ansible03
ansible04

[digitalocean:children]
London
NYC

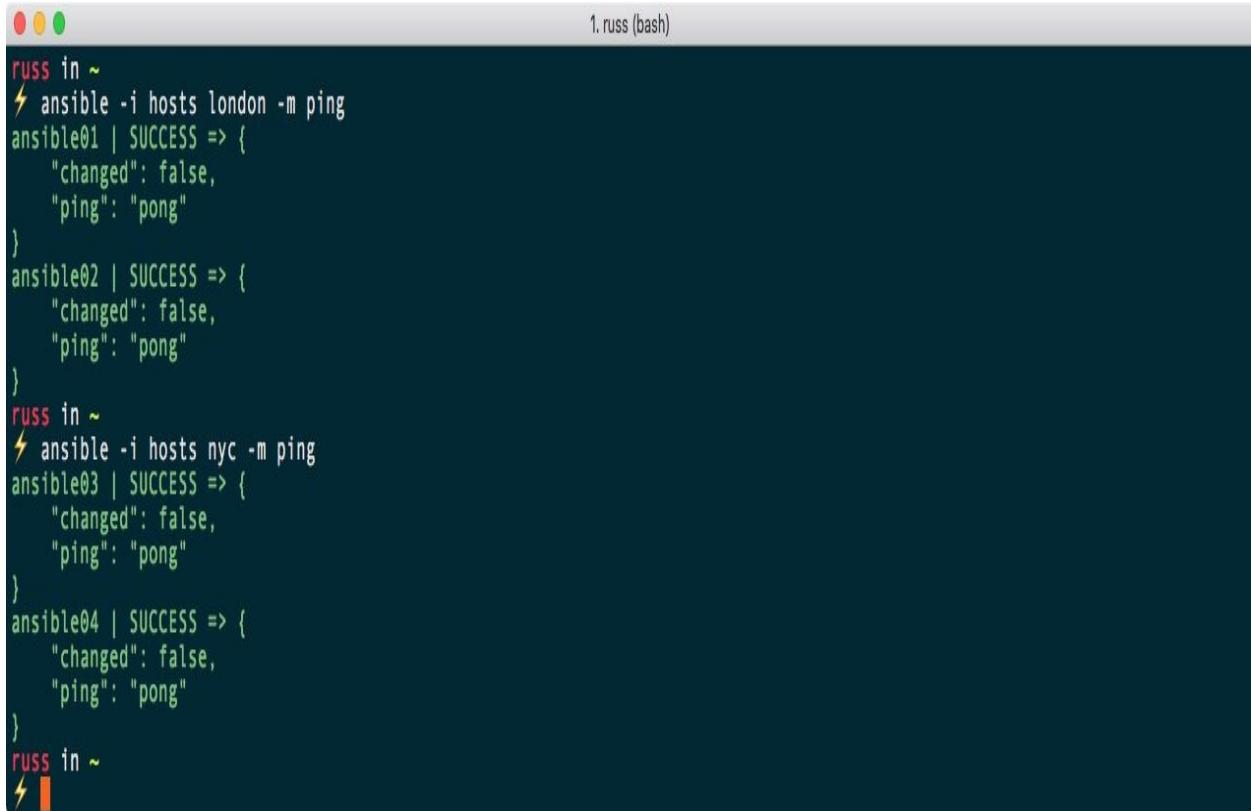
[digitalocean:vars]
ansible_connection=ssh
ansible_user=root
ansible_private_key_file=~/ssh/id_rsa
host_key_checking=False
```

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

Using the `ping` module, I can check connectivity to the hosts by running the following commands:

```
$ ansible -i hosts London -m ping
$ ansible -i hosts NYC -m ping
```

As you can see from these results, all four hosts return pong:



The screenshot shows a terminal window titled "1. russ (bash)". The command run is "ansible -i hosts london -m ping". The output shows four hosts: ansible01, ansible02, ansible03, and ansible04, each returning a "pong" response with "changed": false. The terminal prompt "russ in ~" appears at the end.

```
russ in ~
⚡ ansible -i hosts london -m ping
ansible01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
ansible02 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
ansible03 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
ansible04 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
russ in ~
```

I can also target all four hosts at once by using the following:

```
| $ ansible -i hosts all -m ping
```



1. russ (bash)

```
russ in ~
⚡ ansible -i hosts all -m ping
ansible03 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
ansible01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
ansible02 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
ansible04 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
russ in ~
⚡
```

Now that we have our host accessible through Ansible, we can target them with some ad hoc commands; let's start with something basic:

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

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

```
russ in ~
⚡ ansible -i hosts london -a "ping -c 3 google.com"
ansible01 | SUCCESS | rc=0 >>
PING google.com (216.58.213.110) 56(84) bytes of data.
64 bytes from lhr25s02-in-f110.1e100.net (216.58.213.110): icmp_seq=1 ttl=58 time=1.41 ms
64 bytes from lhr25s02-in-f110.1e100.net (216.58.213.110): icmp_seq=2 ttl=58 time=1.10 ms
64 bytes from lhr25s02-in-f110.1e100.net (216.58.213.110): icmp_seq=3 ttl=58 time=1.10 ms

--- google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 1.101/1.209/1.417/0.147 ms

ansible02 | SUCCESS | rc=0 >>
PING google.com (172.217.23.14) 56(84) bytes of data.
64 bytes from lhr35s01-in-f14.1e100.net (172.217.23.14): icmp_seq=1 ttl=57 time=1.76 ms
64 bytes from lhr35s01-in-f14.1e100.net (172.217.23.14): icmp_seq=2 ttl=57 time=1.25 ms
64 bytes from lhr35s01-in-f14.1e100.net (172.217.23.14): icmp_seq=3 ttl=57 time=1.24 ms

--- google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 1.241/1.418/1.764/0.248 ms

russ in ~
```

We can also run a single module using the `ansible` command; we did this in the previous chapter using the `setup` module. However, a better example would be updating all of the installed packages:

```
| $ ansible -i hosts nyc -m yum -a "name=* state=latest"
```

In the previous example, we are using the `yum` module to update all of the installed packages in the `nyc` group:

```
russ in ~
⚡ ansible -i hosts nyc -m yum -a "name=* state=latest"
ansible03 | SUCCESS => {
    "changed": true,
    "msg": "warning: /var/cache/yum/x86_64/7/updates/packages/bind-license-9.9.4-51.el7_4.2.noarch.rpm: Header V3 RSA/SHA256 Signature, key ID f4a80eb5: NOKEY\nImporting GPG key 0xF4A80EB5:\nUserid : \"CentOS-7 Key (CentOS 7 Official Signing Key) <security@centos.org>\"\nFingerprint: 6341 ab27 53d7 8a78 a7c2 7bb1 24c6 a8a7 f4a8 0eb5\nPackage : centos-release-7-4.1708.el7.centos.x86_64 (installed)\nFrom : /etc/pki/rpm-gpg/RPM-GPG-CentOS-7\n",
    "rc": 0,
    "results": [
        "Loaded plugins: fastestmirror\nLoading mirror speeds from cached hostfile\n * base: mirror.sov.uk.goscomb.net\n * extras: mirror.clustered.net\n * updates: mirror.sov.uk.goscomb.net\nResolving Dependencies\n--> Running transaction check\n---> Package bind-libs-lite.x86_64 32:9.9.4-51.el7_4.1 will be updated\n---> Package bind-libs-lite.x86_64 32:9.9.4-51.el7_4.2 will be an update\n---> Package bind-license.noarch 32:9.9.4-51.el7_4.1 will be updated\n---> Package bind-license.noarch 32:9.9.4-51.el7_4.2 will be an update\n---> Package binutils.x86_64 0:2.25.1-32.base.el7_4.1 will be updated\n---> Package binutils.x86_64 0:2.25.1-32.base.el7_4.2 will be an update\n---> Package dhclient.x86_64 12:4.2.5-58.el7.centos will be updated\n---> Package dhclient.x86_64 12:4.2.5-58.el7.centos.1 will be an update\n---> Package dhcp-common.x86_64 12:4.2.5-58.el7.centos will be updated\n---> Package dhcp-common.x86_64 12:4.2.5-58.el7.centos.1 will be an update\n---> Package dhcp-libs.x86_64 12:4.2.5-58.el7.centos will be updated\n---> Package dhcp-libs.x86_64 12:4.2.5-58.el7.centos.1 will be an update\n---> Package initscripts.x86_64 0:9.49.39-1.el7 will be updated\n---> Package initscripts.x86_64 0:9.49.39-1.el7_4.1 will be an update\n---> Package iw17265-firmware.noarch 0:22.0.7.0-57.el7 will be updated\n---> Package iw17265-firmware.noarch 0:22.0.7.0-58.el7_4 will be an update\n---> Package kernel.x86_64 0:3.10.0-693.17.1.el7 will be installed\n--> Processing Dependency: linux-firmware >= 20170606-55 for package: kernel-3.10.0-693.17.1.el7.x86_64\n---> Package kernel-tools.x86_64 0:3.10.0-693.11.6.el7 will be updated\n---> Package kernel-tools.x86_64 0:3.10.0-693.17.1.el7 will be an update\n---> Package kernel-tools-libs.x86_64 0:3.10.0-693.11.6.el7 will be updated\n---> Package kernel-tools-libs.x86_64 0:3.10.0-693.17.1.el7 will be an update\n---> Package kmod.x86_64 0:20-15.el7_4.6 will be updated\n---> Package kmod.x86_64 0:20-15.el7_4.7 will be an update\n---> Package kmod-libs.x86_64 0:20-15.el7_4.6 will be updated\n---> Package kmod-libs.x86_64 0:20-15.el7_4.7 will be an update\n---> Package kpartx.x86_64 0:0.4.9-111.el7 will be updated\n---> Package kpartx.x86_64 0:0.4.9-111.el7_4.2 will be an update\n---> Package libdb.x86_64 0:5.3.21-20.el7 will be updated\n---> Package libdb.x86_64 0:5.3.21-21.el7_4 will be an update\n---> Package libdb-utils.x86_64 0:5.3.21-20.el7 will be updated\n---> Package libdb-utils.x86_64 0:5.3.21-21.el7_4 will be an update\n---> Package microcode_ctl.x86_64 2:2.1-22.2.el7 will be updated\n---> Package microcode_ctl.x86_64 2:2.1-22.5.el7_4 will be an update\n---> Package nfs-utils.x86_64 1:1.3.0-0.48.el7_4 will be updated\n---> Package nfs-utils.x86_64 1:1.3.0-0.48.el7_4.1 will be an update\n---> Package python-perf.x86_64 0:3.10.0-693.11.6.el7 will be updated\n---> Package python-perf.x86_64 0:3.10.0-693.17.1.el7 will be an update\n---> Package systemd.x86_64 0:219-42.el7_4.4 will be updated\n---> Package systemd.x86_64 0:219-42.el7_4.7 will be updated
```

As you can see from the screenshot, the output when running Ansible is quite verbose, and it has feedback to tell us exactly what it has done during the ad hoc execution. Let's run the command again against all of our hosts, but this time just for a single package, say `kpartx`:

```
$ ansible -i hosts all -m yum -a "name=kpartx state=latest"
```

The Terminal output gives you a better idea of the information being returned by each host as the command is executed on it:

As you can see, the two hosts in the `nyc` group, while returning a `SUCCESS` status, are showing no changes; the two hosts in the `london` group again show a `SUCCESS`

status but show changes.

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

First of all, let's take a look at two of the commands:

```
| $ ansible -i hosts london -a "ping -c 3 google.com"
| $ ansible -i hosts london -m yum -a "name=* state=latest"
```

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

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

So why would you want to use Ansible like this? Well, it's great for running commands directly against non-Ansible managed hosts in an extremely controlled way. Ansible just SSHs in, runs the command, and lets you know the results. Just be careful, as it is very easy to get overconfident and run something like the following:

```
| $ ansible -i hosts all -a "reboot now"
```

If Ansible has permissions to execute the command, then it will do. Running the previous command will reboot all the servers in the host inventory file:



1. russ (bash)

```
⚡ ansible -i hosts all -a "reboot now"
ansible01 | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: Shared connection to 46.101.92.240 closed.\r\n",
    "unreachable": true
}
ansible03 | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: Shared connection to 159.65.63.217 closed.\r\n",
    "unreachable": true
}
ansible02 | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: Shared connection to 159.65.63.218 closed.\r\n",
    "unreachable": true
}
ansible04 | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: Shared connection to 138.68.145.116 closed.\r\n",
    "unreachable": true
}
```

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

```
| $ ansible -i hosts all -a "uptime"
```

The following screenshot shows the output for the preceding command:



1. russ (bash)

```
⚡ ansible -i hosts all -a "uptime"
ansible01 | SUCCESS | rc=0 >>
16:30:57 up 1 min, 1 user, load average: 0.18, 0.12, 0.05

ansible03 | SUCCESS | rc=0 >>
16:30:57 up 1 min, 1 user, load average: 0.12, 0.09, 0.04

ansible02 | SUCCESS | rc=0 >>
16:30:57 up 1 min, 1 user, load average: 0.13, 0.09, 0.04

ansible04 | SUCCESS | rc=0 >>
16:30:57 up 1 min, 1 user, load average: 0.25, 0.10, 0.04
```



As mentioned, be extremely careful when using Ansible to manage hosts using ad hoc commands.

The ansible-config command

The `ansible-config` command is used to manage Ansible configuration files. To be honest, Ansible ships with some quite sensible defaults, so there is not much to configure outside of these. You can view the current configuration by running: `$ ansible-config dump`

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

```
russ in ~ 1. russ (less)
⚡ ansible-config dump
ACCELERATE_CONNECT_TIMEOUT(default) = 1.0
ACCELERATE_DAEMON_TIMEOUT(default) = 30
ACCELERATE_KEYS_DIR(default) = ~/.fireball.keys
ACCELERATE_KEYS_DIR_PERMS(default) = 700
ACCELERATE_KEYS_FILE_PERMS(default) = 600
ACCELERATE_MULTI_KEY(default) = False
ACCELERATE_PORT(default) = 5099
ACCELERATE_TIMEOUT(default) = 30
ALLOW_WORLD_READABLE_TMPFILES(default) = False
ANSIBLE_COW_SELECTION(default) = default
ANSIBLE_COW_WHITELIST(default) = ['bud-frogs', 'bunny', 'cheese', 'daemon', 'default', 'dragon', 'el
ANSIBLE_FORCE_COLOR(default) = False
ANSIBLE_NOCOLOR(default) = False
ANSIBLE_NO_COWS(default) = False
ANSIBLE_PIPELINING(default) = False
ANSIBLE_SSH_ARGS(default) = -C -o ControlMaster=auto -o ControlPersist=60s
ANSIBLE_SSH_CONTROL_PATH(env: ANSIBLE_SSH_CONTROL_PATH) = /tmp/%h-%p-%r
ANSIBLE_SSH_CONTROL_PATH_DIR(default) = ~/.ansible/cp
ANSIBLE_SSH_EXECUTABLE(default) = ssh
ANSIBLE_SSH_RETRIES(default) = 0
ANY_ERRORS_FATAL(default) = False
BECOME_ALLOW_SAME_USER(default) = False
CACHE_PLUGIN(default) = memory
```

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

The following screenshot shows the output for the preceding command:

```
1. russ (less)
ANSIBLE_COW_SELECTION:
:
description: This allows you to chose a specific cowsay stencil for the banners
or use 'random' to cycle through them.
env:
- {name: ANSIBLE_COW_SELECTION}
ini:
- {key: cow_selection, section: defaults}
name: Cowsay filter selection
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]
description: White list of cowsay templates that are 'safe' to use, set to empty
list if you want to enable all installed templates.
env:
- {name: ANSIBLE_COW_WHITELIST}
ini:
- {key: cow_whitelist, section: defaults}
name: Cowsay filter whitelist
type: list
yaml: {key: display.cowsay_whitelist}
```

If you had a configuration file, say at `~/.ansible.cfg`, then you can load it using the `-c` or `--config` flags: **\$ ansible-config --config="~/.ansible.cfg" view**

The previous command will show you the configuration file.

The ansible-console command

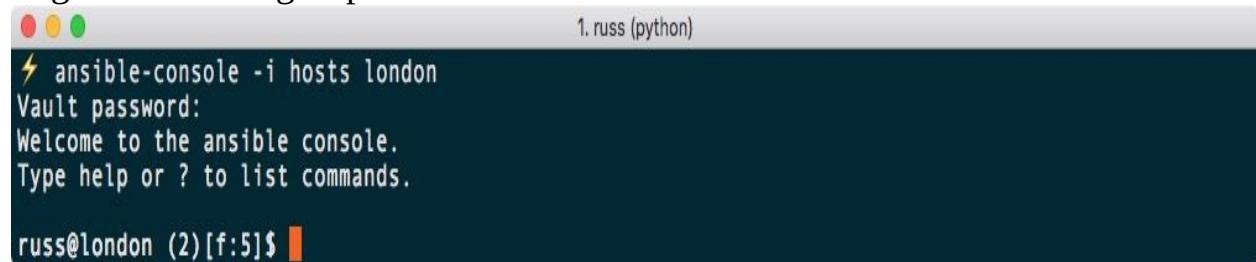
Ansible has its own built-in console. Personally, it is not something I have used much at all. To start the console, we simply need to run one of the following commands:

```
$ ansible-console -i hosts
```

```
$ ansible-console -i hosts london
```

```
$ ansible-console -i hosts nyc
```

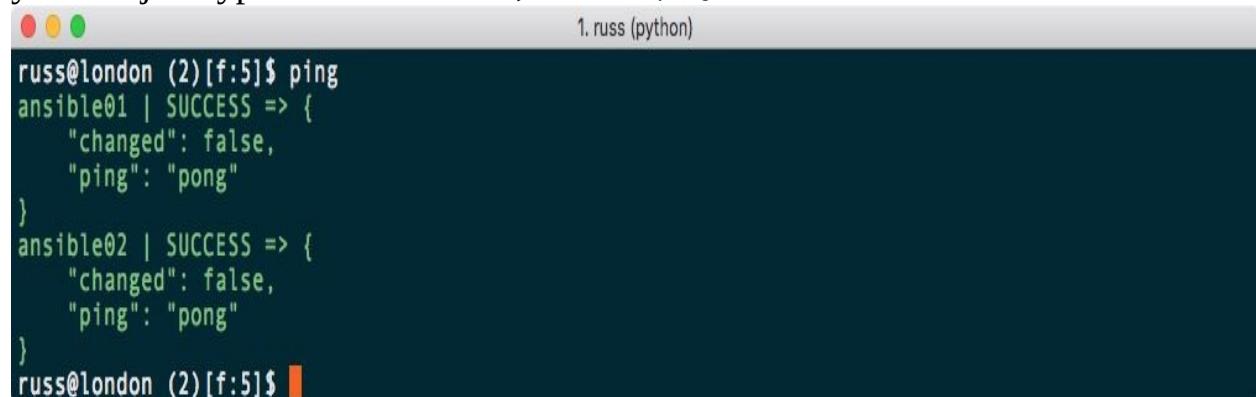
The first of the three commands targets all of the hosts, while the next two just target the named groups:



```
ansible-console -i hosts london
Vault password:
Welcome to the ansible console.
Type help or ? to list commands.

russ@london (2)[f:5]$
```

As you can see from the Terminal output, you are asked for an Ansible Vault password. Just enter anything here as we do not have anything protected by Ansible Vault; more on that later in the chapter. Once connected, you can see that I am connected to the `london` group, in which there are two hosts. From here, you can just type a module name, such as `ping`:



```
russ@london (2)[f:5]$ ping
ansible01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
ansible02 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
russ@london (2)[f:5]$
```

Or use the `raw` module, by typing `raw uptime`:

```
1. russ (python)
russ@london (2)[f:5]$ raw uptime
ansible01 | SUCCESS | rc=0 >>
 18:38:44 up 2:09, 1 user, load average: 0.00, 0.01, 0.05
Shared connection to 46.101.92.240 closed.

ansible02 | SUCCESS | rc=0 >>
 18:38:44 up 2:09, 1 user, load average: 0.05, 0.07, 0.06
Shared connection to 159.65.63.218 closed.

russ@london (2)[f:5]$
```

You can also use the same syntax as we did when running the `ansible` command to pass key-value pairs, for example, `yum name=kpartx state=latest`:

```
1. russ (python)
russ@london (2)[f:5]$ yum name=kpartx state=latest
ansible01 | SUCCESS => {
    "changed": false,
    "msg": "",
    "rc": 0,
    "results": [
        "All packages providing kpartx are up to date",
        ""
    ]
}
ansible02 | SUCCESS => {
    "changed": false,
    "msg": "",
    "rc": 0,
    "results": [
        "All packages providing kpartx are up to date",
        ""
    ]
}
russ@london (2)[f:5]$
```

To leave the console, simple type `exit` to return to your regular shell.

The `ansible-doc` command

The `ansible-doc` command has one function—to provide documentation for Ansible. It mostly covers the core Ansible modules, which you can find a full list of by running the following command: `$ ansible-doc --list`

For information on a module, just run the command followed by the module name, for example:

```
| $ ansible-doc raw
```

As you can see from the following output, the documentation is quite detailed:

```
1. russ (less)
> RAW (/usr/local/Cellar/ansible/2.4.3.0/libexec/lib/python2.7/site-packages/ansible/modules/comm

    Executes a low-down and dirty SSH command, not going through the module
    subsystem. This is useful and should only be done in two cases. The
    first case is installing `python-simplejson` on older (Python 2.4 and
    before) hosts that need it as a dependency to run modules, since nearly
    all core modules require it. Another is speaking to any devices such as
    routers that do not have any Python installed. In any other case, using
    the [shell] or [command] module is much more appropriate. Arguments
    given to `raw` are run directly through the configured remote shell.
    Standard output, error output and return code are returned when
    available. There is no change handler support for this module. This
    module does not require python on the remote system, much like the
    [script] module. This module is also supported for Windows targets.

    * note: This module has a corresponding action plugin.

OPTIONS (= is mandatory):
- executable
    change the shell used to execute the command. Should be an absolute path
    to the executable.
    when using privilege escalation (`become`), a default shell will be
    assigned if one is not provided as privilege escalation requires a
    shell.
    [Default: (null)]
    version_added: 1.0

= free_form
:
```

If you just want to see how to use the example in your playbook, then you can use the following command:

```
| $ ansible-doc --snippet raw
```

This will give you an idea of what your playbook should contain, as you can see from the following output for the `raw` module:

```
● ● ● 1. russ (bash)
russ in ~
⚡ ansible-doc --snippet raw
- name: Executes a low-down and dirty SSH command
  raw:
    executable:          # change the shell used to execute the command. Should be an absolute
                        path to the executable. When using
                        privilege escalation ('become'), a
                        default shell will be assigned if one
                        is not provided as privilege
                        escalation requires a shell.
    free_form:           # (required) the raw module takes a free form command to run. There is
                        no parameter actually named 'free
                        form'; see the examples!
russ in ~
⚡ █
```

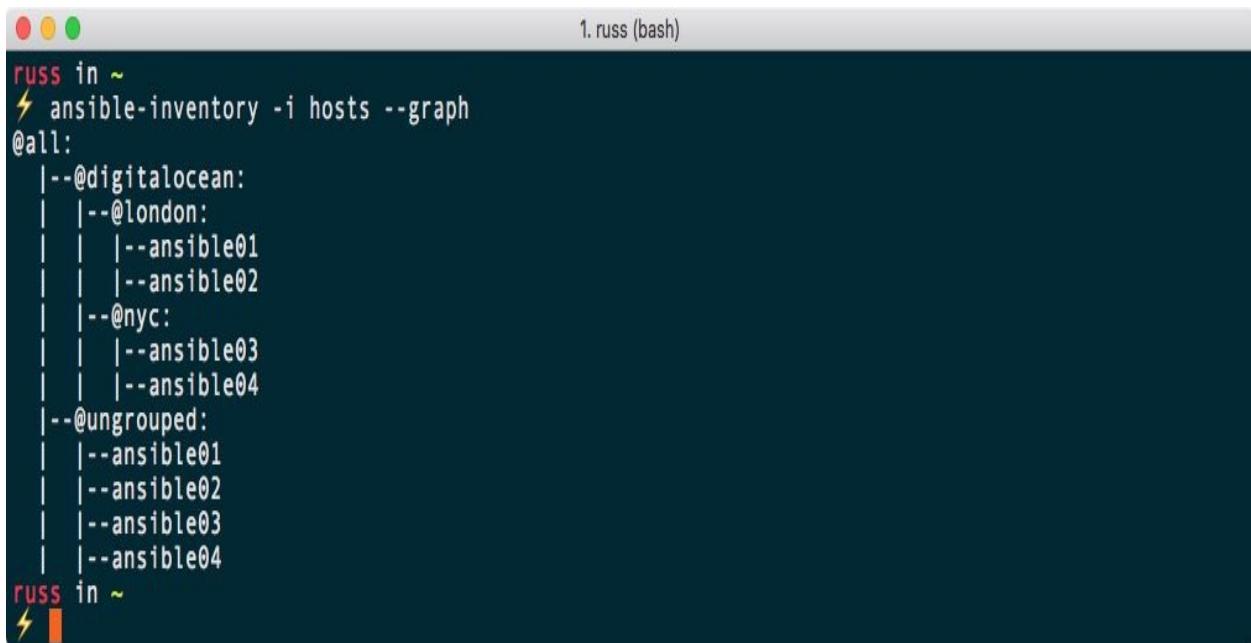
The content of the `ansible-doc` command mirrors the documentation that can be found at the Ansible website, but it's useful if you want to quickly check the syntax needed for a module.

The ansible-inventory command

Using the `ansible-inventory` command provides you with details of your host inventory files. It can be useful if you want to get an idea of how your hosts are grouped. For example, run the following:

```
| $ ansible-inventory -i hosts --graph
```

This gives you a logical overview of your host groups. Here is the hosts inventory file we first used with the `ansible` command at the start of the chapter:



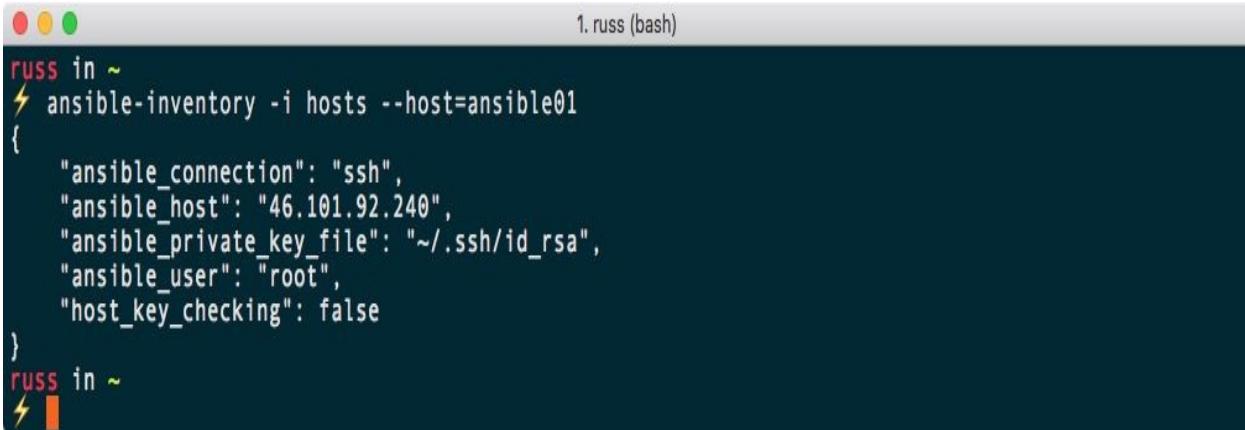
```
russ in ~
⚡ ansible-inventory -i hosts --graph
@all:
|--@digitalocean:
|   |--@london:
|   |   |--ansible01
|   |   |--ansible02
|   |--@nyc:
|       |--ansible03
|       |--ansible04
|--@ungrouped:
    |--ansible01
    |--ansible02
    |--ansible03
    |--ansible04
russ in ~
⚡
```

As you can see, it displays the groups, starting with all, then the main host group, followed by the child groups, and then finally the hosts themselves.

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

```
| $ ansible-inventory -i hosts --host=ansible01
```

The following screenshot shows the output for the preceding command:



```
russ in ~
⚡ ansible-inventory -i hosts --host=ansible01
{
    "ansible_connection": "ssh",
    "ansible_host": "46.101.92.240",
    "ansible_private_key_file": "~/.ssh/id_rsa",
    "ansible_user": "root",
    "host_key_checking": false
}
russ in ~
```

You may have noticed that it displays the configuration information that the host inherited from the configuration we set for all of the DigitalOcean hosts. You can view all the information on each of your hosts and groups by running:

```
| $ ansible-inventory -i hosts --list
```

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

Ansible Vault

In Ansible, it is possible to load in variables from files. We will be looking at this in our next chapter in more detail. These files can contain sensitive information such as password and API keys. An example of this would be the following:

```
| secret: "mypassword"  
| secret-api-key: "myprivateapikey"
```

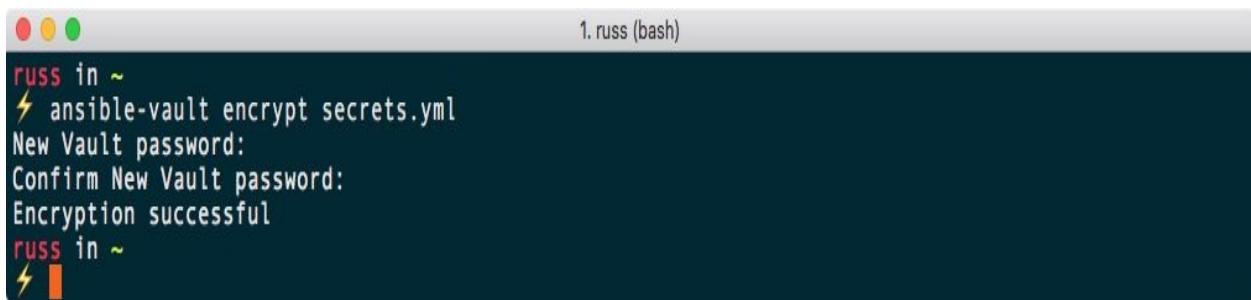
As you can see, we have two sensitive bits of information visible as plaintext. This is OK while the file is on our local machine, but what if we want to check the file into source control to share it with our colleagues? Even if the repository is private, we shouldn't be storing this type of information in plaintext.

Ansible introduced Vault to help solve this very problem. Using Vault, we can encrypt the file and then when, Ansible is executed, it can be decrypted in memory and the content read.

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

```
| $ ansible-vault encrypt secrets.yml
```

The following screenshot shows the output for the preceding command:



```
russ in ~  
⚡ ansible-vault encrypt secrets.yml  
New Vault password:  
Confirm New Vault password:  
Encryption successful  
russ in ~
```

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

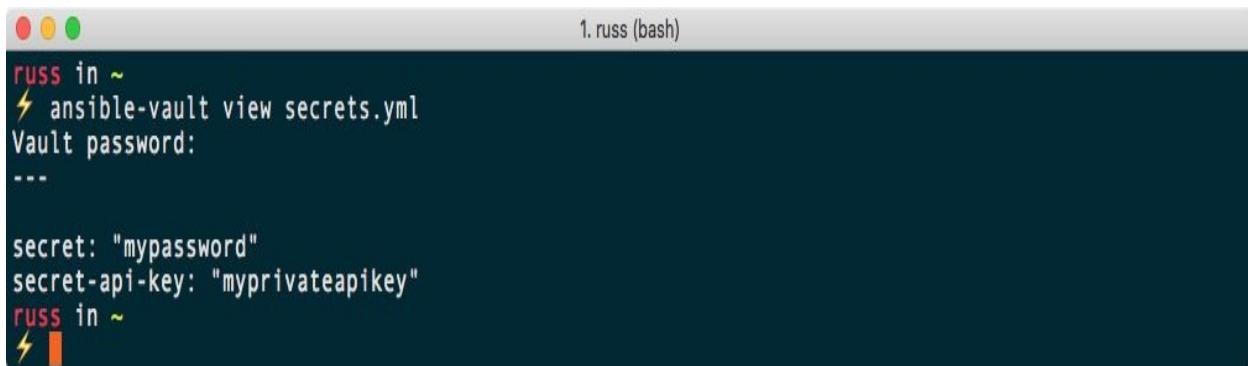
```
$ANSIBLE_VAULT;1.1;AES256  
32643164646266353962363635363831366431316264366261616238333237383063313035343062
```

```
| 6431336434356661646336393061626130373233373161660a363532316138633061643430353235  
| 3234346661303866333383835633831363436343363613933626332383565663562366163393866  
| 6532393661633762310a393935373533666230383063376639373831383965303461636433356365  
| 6432616261363733663036373330373234306537323333263613538656361396163376165353237  
| 30393265616630366134383830626335646338343739353638313264336638363338356136636637  
| 623236653139386534613236623434626131
```

As you can see, the details are encoded using text. This makes sure that our `secrets.yml` file will still work without any problems with source control. You can view the content of a file by running:

```
| $ ansible-vault view secrets.yml
```

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



The screenshot shows a terminal window titled "1. russ (bash)". The user has run the command `ansible-vault view secrets.yml`. A prompt "Vault password:" appears, followed by three dots indicating a password entry. After entering the password, the terminal displays the decrypted contents of the file:

```
russ in ~
⚡ ansible-vault view secrets.yml
Vault password:
---
secret: "mypassword"
secret-api-key: "myprivateapikey"
russ in ~
⚡
```

You can decrypt the file on disk by running:

```
| $ ansible-vault decrypt secrets.yml
```

When using this command, please remember not to check the decrypted file into your source control system!

Since Ansible 2.4, it is now possible to encrypt a single variable in a file. Let's add some more variables to our file:

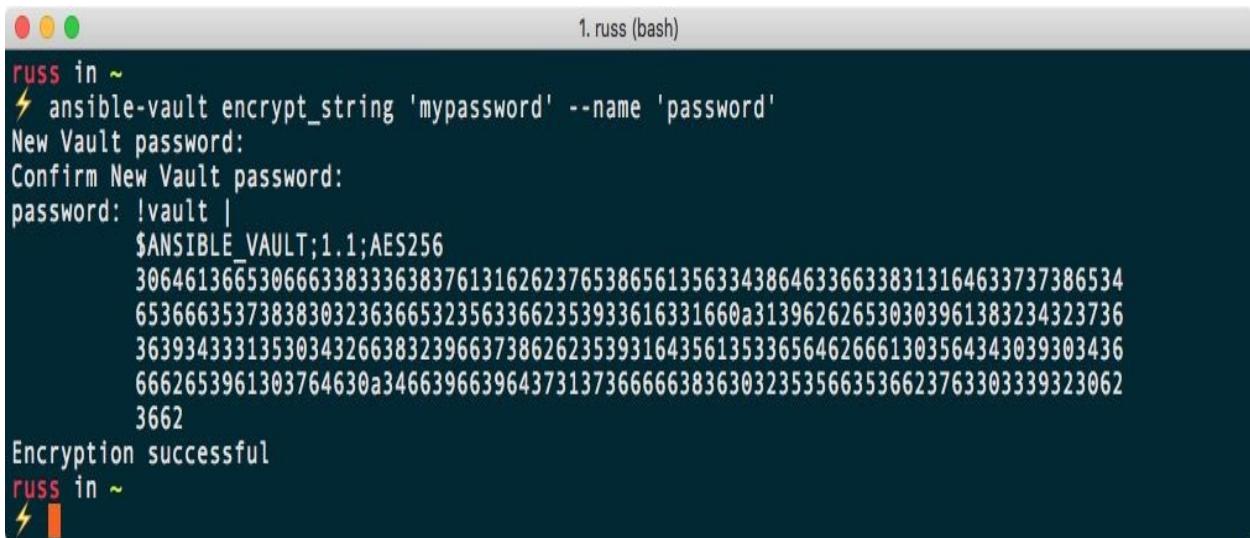
```
username: russmckendrick
password: "mypassword"
secretapikey: "myprivateapikey"
packages:
  - httpd
  - php
  - mariadb
```

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

Let's encrypt the password content by running the following:

```
| $ ansible-vault encrypt_string 'mypassword' --name 'password'
```

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



The screenshot shows a terminal window titled "1. russ (bash)". The user runs the command `ansible-vault encrypt_string 'mypassword' --name 'password'`. It prompts for a new vault password, which is then confirmed. The user enters the password `!vault` followed by a long hex string representing the encrypted password. The output ends with "Encryption successful".

```
russ in ~
⚡ ansible-vault encrypt_string 'mypassword' --name 'password'
New Vault password:
Confirm New Vault password:
password: !vault |
$ANSIBLE_VAULT;1.1;AES256
30646136653066633833363837613162623765386561356334386463366338313164633737386534
6536663537383830323636653235633662353933616331660a313962626530303961383234323736
36393433313530343266383239663738626235393164356135336564626661303564343039303436
6662653961303764630a346639663964373137366666383630323535663536623763303339323062
3662
Encryption successful
russ in ~
⚡
```

We can then copy and paste the output into our file, repeat the process again for the `secret-api-key`, and we end up with the following:

```
username: "russmckendrick"
password: !vault |
$ANSIBLE_VAULT;1.1;AES256
30646136653066633833363837613162623765386561356334386463366338313164633737386534
6536663537383830323636653235633662353933616331660a313962626530303961383234323736
36393433313530343266383239663738626235393164356135336564626661303564343039303436
6662653961303764630a346639663964373137366666383630323535663536623763303339323062
3662
secretapikey: !vault |
$ANSIBLE_VAULT;1.1;AES256
6361393231393336532303237373732386337663662656337623962313638313338333763396232
3463303765303530323133323064346539653234343933330a656537646262633765353766323737
32303633323166643664323133303336393161663838386632346336626535303466303863346239
3764633164613862350a363830336633356233626631636266303632663335346234373034376235
3836
packages:
- "httpd"
- "php"
- "mariadb"
```

As you can see, that is a lot easier to read and is just as secure as encrypting the file as a whole. There is one last thing with Ansible Vault, and that is that you can also read the password from a file; for example, I have been encoding my Vaults using the password of `password`. Let's put that in a file and then use it to unlock our Vault:

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

As you can see in the following `playbook.yml` file, we are reading the `secrets.yml` file and then outputting the content using the `debug` module:

```
---
- hosts: localhost
  vars_files:
    - secrets.yml
  tasks:
    - debug:
        msg: "The username is {{ username }} and password is {{ password }}, also the
          API key is {{ secretapikey }}"
    - debug:
        msg: "I am going to install {{ packages }}"
```

Run the `playbook.yml` file using the following command:

```
| $ ansible-playbook playbook.yml
```

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

```
● ● ● 1. chapter03 (bash)
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter03
⚡ ansible-playbook playbook.yml
[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit
localhost does not match 'all'

PLAY [localhost] ****
TASK [Gathering Facts] ****
ok: [localhost]

TASK [debug] ****
fatal: [localhost]: FAILED! => {"msg": "Attempting to decrypt but no vault secrets found"}
    to retry, use: --limit @/Users/russ/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/
chapter03/playbook.retry

PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=1

russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter03
⚡ |
```

As you can see, it is complaining that it found Vault-encrypted data in our file, but we haven't provided the secret to unlock it. Running the following command will read the content of /tmp/vault-file and decrypt the data:

```
| $ ansible-playbook --vault-id /tmp/vault-file playbook.yml
```

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

```
● ● ● 1. chapter03 (bash)
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter03
⚡ ansible-playbook --vault-id /tmp/vault-file playbook.yml
[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit
localhost does not match 'all'

PLAY [localhost] ****
TASK [Gathering Facts] ****
ok: [localhost]

TASK [debug] ****
ok: [localhost] => {
    "msg": "The username is russmckendrick and password is mypassword, also the API key is myprivate
apikey"
}

TASK [debug] ****
ok: [localhost] => {
    "msg": "I am going to install [u'httpd', u'php', u'mariadb']"
}

PLAY RECAP ****
localhost : ok=3    changed=0    unreachable=0    failed=0
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter03
⚡ |
```

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

```
| $ ansible-playbook --vault-id @prompt playbook.yml
```

You can find a copy of `playbook.yml` and `secrets.yml` in the `chapter03` folder of the accompanying repository.

Third-party commands

Before we finish up looking at Ansible commands, there are a few different third-party commands I would like to cover, the first of which is `ansible-inventory-grapher`.

The ansible-inventory-grapher command

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

```
| $ brew install graphviz
```

Or, to install Graphviz on Ubuntu, use:

```
| $ sudo apt-get install graphviz
```

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

```
| $ sudo install ansible-inventory-grapher
```

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

```
ansible01 ansible_host=46.101.92.240
ansible02 ansible_host=159.65.63.218
ansible03 ansible_host=159.65.63.217
ansible04 ansible_host=138.68.145.116

[london]
ansible01
ansible02

[nyc]
ansible03
ansible04

[digitalocean:children]
london
nyc

[digitalocean:vars]
ansible_connection=ssh
ansible_user=root
ansible_private_key_file=~/ssh/id_rsa
host_key_checking=False
```

We can run the following command to generate the raw graph file:

```
| $ ansible-inventory-grapher -i hosts digitalocean
```

This will generate the following output:

```
digraph "digitalocean" {
    rankdir=TB;

    "all" [shape=record label=<
        <table border="0" cellborder="0">
            <tr><td><b><font face="Times New Roman, Bold" point-size="16">
                all</font></b></td></tr>
        </table>
    >]
    "ansible01" [shape=record style="rounded" label=<
        <table border="0" cellborder="0">
            <tr><td><b><font face="Times New Roman, Bold" point-size="16">
                ansible01</font></b></td></tr>
            <hr/><tr><td><font face="Times New Roman, Bold" point-size="14">ansible_connection<br/>ansible_host<br/>
                ansible_private_key_file<br/>ansible_user<br/>
                host_key_checking<br/></font></td></tr>
        </table>
    >]
    "ansible02" [shape=record style="rounded" label=<
        <table border="0" cellborder="0">
            <tr><td><b><font face="Times New Roman, Bold" point-size="16">
                ansible02</font></b></td></tr>
            <hr/><tr><td><font face="Times New Roman, Bold" point-size="14">ansible_connection<br/>ansible_host<br/>
                ansible_private_key_file<br/>ansible_user<br/>
                host_key_checking<br/></font></td></tr>
        </table>
    >]
    "ansible03" [shape=record style="rounded" label=<
        <table border="0" cellborder="0">
            <tr><td><b><font face="Times New Roman, Bold" point-size="16">
                ansible03</font></b></td></tr>
            <hr/><tr><td><font face="Times New Roman, Bold" point-size="14">ansible_connection<br/>ansible_host<br/>
                ansible_private_key_file<br/>ansible_user<br/>
                host_key_checking<br/></font></td></tr>
        </table>
    >]
    "ansible04" [shape=record style="rounded" label=<
        <table border="0" cellborder="0">
            <tr><td><b><font face="Times New Roman, Bold" point-size="16">
                ansible04</font></b></td></tr>
            <hr/><tr><td><font face="Times New Roman, Bold" point-size="14">ansible_connection<br/>ansible_host<br/>
                ansible_private_key_file<br/>ansible_user<br/>
                host_key_checking<br/></font></td></tr>
        </table>
    >]
    "digitalocean" [shape=record label=<
        <table border="0" cellborder="0">
            <tr><td><b><font face="Times New Roman, Bold" point-size="16">
                digitalocean</font></b></td></tr>
        </table>
    >]
    "london" [shape=record label=<
        <table border="0" cellborder="0">
            <tr><td><b><font face="Times New Roman, Bold" point-size="16">
                london</font></b></td></tr>
        </table>
    >]
    "nyc" [shape=record label=<
```

```


|                                                                                |
|--------------------------------------------------------------------------------|
| <b><font face="Times New Roman, Bold" point-size="16">nyc</font></b></td></tr> |
|--------------------------------------------------------------------------------|


]

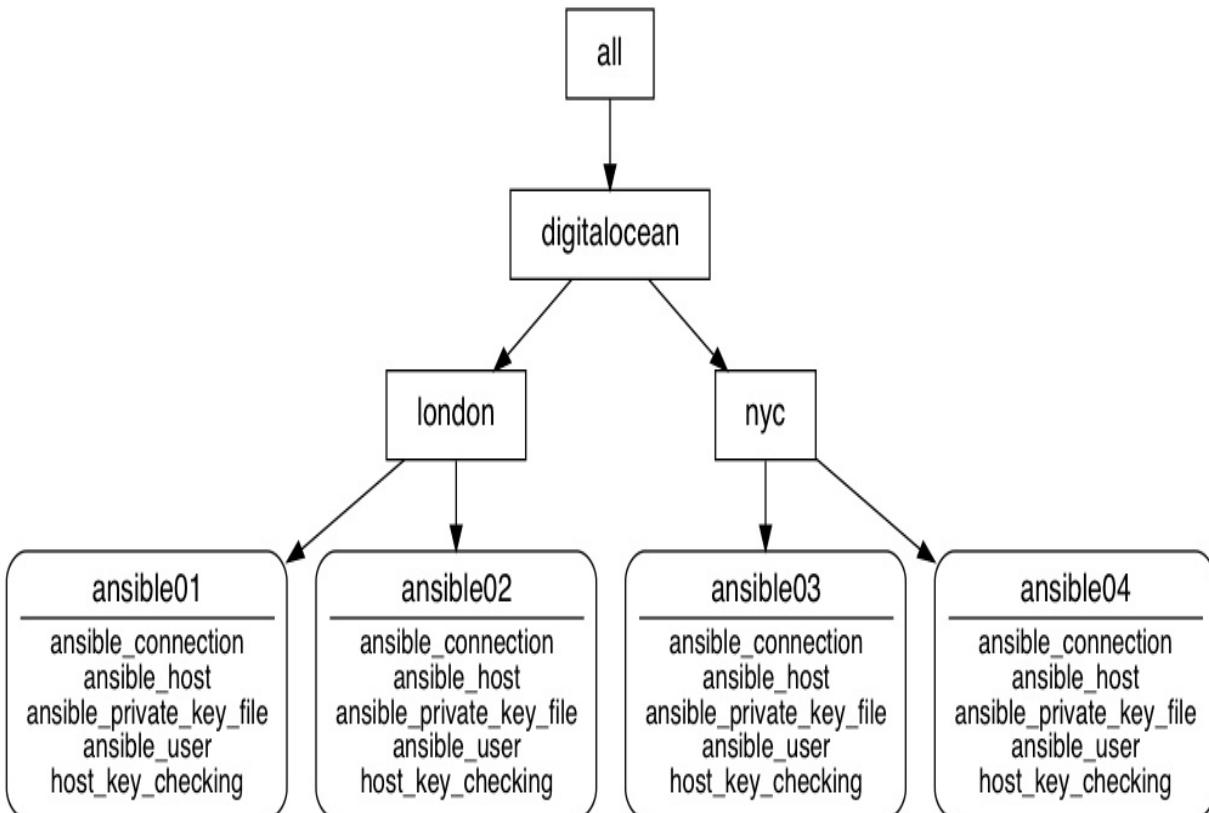
"all" -> "digitalocean";
"digitalocean" -> "london";
"digitalocean" -> "nyc";
"london" -> "ansible01";
"london" -> "ansible02";
"nyc" -> "ansible03";
"nyc" -> "ansible04";
}

```

This is the raw output of the graph. As you can see, it is similar to HTML. We can render this using the `dot` command, which ships as part of Graphviz. The `dot` command creates hierarchical drawings from graphs. To do this, run:

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

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



We will be using this tool in later chapters to get an idea of what our inventory files look like as they are generated.

Ansible Run Analysis

ARA, which is a recursive acronym that stands for **Ansible Run Analysis**, records Ansible. It is a tool written in Python that records your playbook runs and displays the results in an intuitive web interface. To install it on macOS, I had to use the following command:

```
| $ sudo pip install ara --ignore-installed pyparsing
```

To install on Ubuntu, I could use just this:

```
| $ sudo pip install ara
```

Once installed, you should be able to run the following commands to configure your environment to record your Ansible playbook runs:

```
$ export ara_location=$(python -c "import os,ara; print(os.path.dirname(ara.__file__))")  
$ export ANSIBLE_CALLBACK_PLUGINS=$ara_location/plugins/callbacks  
$ export ANSIBLE_ACTION_PLUGINS=$ara_location/plugins/actions  
$ export ANSIBLE_LIBRARY=$ara_location/plugins/modules
```

When you have your environment configured, you can run a playbook. For example, let's rerun the playbook from the Ansible Vault section of this chapter using:

```
| $ ansible-playbook --vault-id @prompt playbook.yml
```

Once the playbook has been executed, running the following command will start the ARA web server:

```
| $ ara-manage runserver
```

Opening your browser and going to the URL mentioned in the output of the previous command, <http://127.0.0.1:9191/>, will give you the results of your playbook run:

The screenshot shows the Ara web interface at the URL 127.0.0.1. The top navigation bar includes links for Documentation, ARA 0.14.6, Ansible 2.4.3.0, and Python 2.7. Below the header, there are two tabs: "Playbook reports" (selected) and "About".

The main content area displays four entries, each representing a playbook execution:

- playbook.yml (2018-02-24 14:36:10)**: Status is successful (green checkmark). Execution time: 0:00:02. Details: 0 Parameters, 1 Hosts, 1 Plays, 1 Files, 3 Tasks, 0 Records.
- playbook.yml (2018-02-24 12:11:21)**: Status is failed (red X). Execution time: 0:00:01. Details: 0 Parameters, 1 Hosts, 1 Plays, 1 Files, 2 Tasks, 0 Records.
- playbook.yml (2018-02-24 12:11:11)**: Status is successful (green checkmark). Execution time: 0:00:01. Details: 0 Parameters, 1 Hosts, 1 Plays, 1 Files, 3 Tasks, 0 Records.
- playbook.yml (2018-02-24 11:26:33)**: Status is successful (green checkmark). Execution time: 0:00:01. Details: 0 Parameters, 1 Hosts, 1 Plays, 1 Files, 3 Tasks, 0 Records.

At the bottom of the list, a message states: "Displaying 4 playbook reports out of a total of 4."

As you can see, I have run the playbook four different times, and one of those executions failed. Clicking on the elements will show you more detail:

The screenshot shows the ARA web interface at 127.0.0.1. The top navigation bar includes links for Documentation, ARA 0.14.6, Ansible 2.4.3.0, and Python 2.7. The main content area displays a playbook report for the file `/Users/russ/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter03/playbook.yml`. The report was last updated on 2018-02-24 14:36:12 and was run with Ansible version 2.4.3.0. The playbook content is as follows:

```
1 ---  
2  
3 - hosts: localhost  
4  
5 vars_files:  
6   - secrets.yml  
7  
8 tasks:  
9   - debug:  
10     msg: "The username is {{ username }} and password is {{ password }}, also the API key is {{ secretapikey }}"  
11   - debug:  
12     msg: "I am going to install {{ packages }}"
```

Below the code editor, a message states: "Displaying 4 playbook reports out of a total of 4."

Again, we will be using ARA in later chapters in a lot more detail; we have just touched upon the very basics here.

Summary

In this chapter, we took a brief look at some of the supporting tools that ship as part of a standard Ansible installation, as well as some useful third-party tools that have been designed to work with Ansible. We will be using these commands, as well as the few we have purposely missed, in later chapters.

In our next chapter, we are going to make a start by writing a more complex playbook that installs a basic LAMP stack on our local Vagrant box.

Questions

1. Of the commands that provide information about your host inventory that we have covered in this chapter, which ships with Ansible by default?
2. True or false: Variable files that have strings encrypted with Ansible Vault will work with versions of Ansible lower than 2.4.
3. What command would you run to get an example of how you should call the `yum` module as part of your task?
4. Explain why you would want to run single modules against hosts within your inventory.
5. Using your own host inventory file, generate a diagram showing the content.

Further reading

You can find the project pages for the two third-party tools covered at the end of the chapter at the following URLs:

- `ansible-inventory-grapher`: <https://github.com/willthames/ansible-inventory-grapher>
- `ara`: <https://github.com/openstack/ara>

Deploying a LAMP Stack

In this chapter, we will look at deploying a full LAMP stack using the various core modules that ship with Ansible. We will be targeting the CentOS 7 Vagrant box we deployed in [Chapter 2, *Installing and Running Ansible*](#).

We will talk about the following:

- Playbook layout—how the playbook should be structured
- Linux—preparing the Linux server
- Apache—installing and configuring Apache
- MariaDB—installing and configuring MariaDB
- PHP—installing and configuring PHP

Before we start writing the playbook, we should discuss the structure we are going to be using after we quickly discuss what we need for the chapter.

Technical requirements

We are going to again use the CentOS 7 Vagrant box we launched in the previous chapters. As we will be installing all of the elements of a LAMP stack on the virtual machine, your Vagrant box will need to be able to download packages from the internet; in all, there is around 500 MB of packages and configuration to download.

You can find a complete copy of the playbook at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter04/lamp>.

Playbook structure

In the previous chapters, the playbooks we have been running have typically been as basic as possible. They have all been in a single file, which is accompanied by a host inventory file. In this chapter, as we are going to be greatly expanding the amount of work our playbook is doing, so we are going to be using the directory structure recommended by Ansible.

As you can see from the following layout, there are several folders and files:

```
russ in ~/lamp 1. lamp (bash)
⚡ tree

.
├── Vagrantfile
├── group_vars
│   └── common.yml
├── production
└── roles
    └── common
        ├── README.md
        ├── defaults
        │   └── main.yml
        ├── files
        ├── handlers
        │   └── main.yml
        ├── meta
        │   └── main.yml
        ├── tasks
        │   └── main.yml
        ├── templates
        ├── tests
        │   └── inventory
        │       └── test.yml
        └── vars
            └── main.yml

└── site.yml

11 directories, 12 files
russ in ~/lamp
⚡
```

Let's work on creating the structure and discuss each item as we create it. The

first folder we need to create is our top-level folder. This is the folder that will contain our playbook folders and files:

```
| $ mkdir lamp  
| $ cd lamp
```

The next folder we are going to create is one called `group_vars`. This will contain the variable files used in our playbook. For now, we are going to be creating a single variable file call `common.yml`:

```
| $ mkdir group_vars  
| $ touch group_vars/common.yml
```

Next, we are going to be creating two files: our host inventory file, which we will name `production`, and also our master playbook, which is typically called `site.yml`:

```
| $ touch production  
| $ touch site.yml
```

The final folder we are going to create manually is called `roles`. In here, we are going to use the `ansible-galaxy init` command to create a role called `common`. To do this, we use the following commands:

```
| $ mkdir roles  
| $ ansible-galaxy init roles/common
```

As you may have noticed from the initial structure at the start of this section, the `common` role has several files and folders itself; all of these are created for us when we run the `ansible-galaxy init` command. We will discuss what each of these does in the next section where we will be using the `common` role to configure our base Linux server.

The only other file that isn't part of the default Ansible structure is our `vagrantfile`. This contains the following content:

```
# -*- mode: ruby -*-  
# vi: set ft=ruby :  
  
API_VERSION = "2"  
BOX_NAME    = "centos/7"  
BOX_IP      = "192.168.50.4"  
DOMAIN      = "nip.io"  
PRIVATE_KEY = "~/.ssh/id_rsa"  
PUBLIC_KEY  = '~/.ssh/id_rsa.pub'  
  
Vagrant.configure(API_VERSION) do |config|
```

```
config.vm.box = BOX_NAME
config.vm.network "private_network", ip: BOX_IP
config.vm.host_name = BOX_IP + '.' + DOMAIN
config.ssh.insert_key = false
config.ssh.private_key_path = [PRIVATE_KEY,
  "~/.vagrant.d/insecure_private_key"]
config.vm.provision "file", source: PUBLIC_KEY,
destination: "~/.ssh/authorized_keys"

config.vm.provider "virtualbox" do |v|
  v.memory = "2024"
  v.cpus = "2"
end

config.vm.provider "vmware_fusion" do |v|
  v.vmx["memsize"] = "2024"
  v.vmx["numvcpus"] = "2"
end

end
```



While we will be working through each of the files individually in this and the following sections, a complete copy of the playbook is available in the accompanying GitHub repository.

LAMP stack

The LAMP stack is the term used to describe an all-in-one web and database server. Typically, the components are:

- **Linux:** The underlying operating system; in our case, we will be using CentOS 7.
- **Apache:** The web server element of the stack.
- **MariaDB:** The database component of the stack; typically, it is MySQL-based. As CentOS 7 ships with MariaDB, we will be using that rather than PHP.
- **PHP:** The dynamic language used by the web server to generate content.

There is also a common variation of the LAMP stack called **LEMP**; this replaces *Apache* with *NGINX*, which is pronounced *engine-x*, hence the *E* rather than *N*.

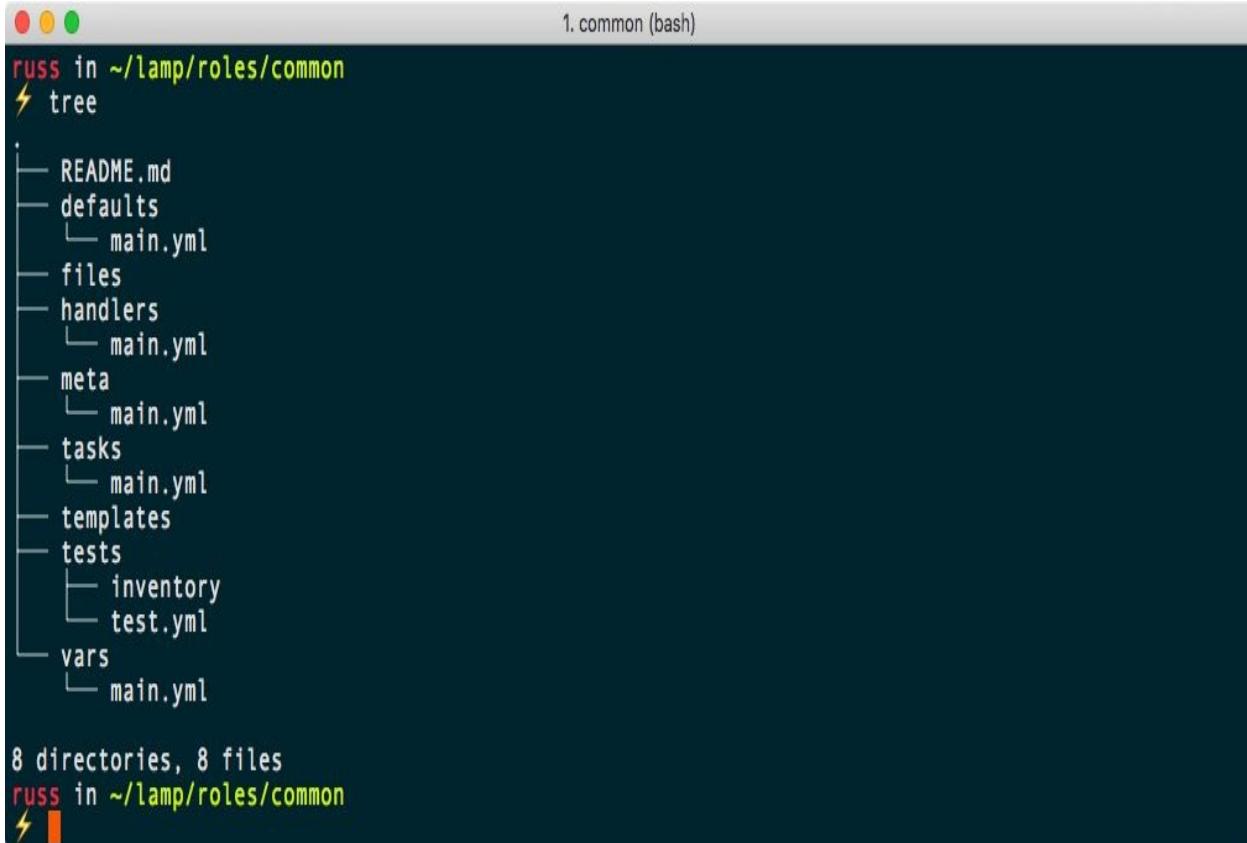
We are going to be looking at creating roles to deal with these components; these are:

- `common`: This role will prepare our CentOS server, installing any supporting packages and services we need
- `apache`: This role will install the Apache web server and also configure a default virtual host
- `mariadb`: This role will not only install MariaDB, but it will also secure the installation and create a default database and user
- `php`: This role will install PHP, a set of common PHP modules, and also Composer, which is a package manager for PHP

Let's make a start on getting the basics ready by writing the common role.

Common

In the previous section of this chapter, we used the `ansible-galaxy init` command to create the `common` role. This creates several folders and files; before we start editing them, let's quickly discuss what each of them is used for:



The screenshot shows a terminal window titled "1. common (bash)". The command `tree` is run, displaying the directory structure of the `common` role. The structure is as follows:

```
.
├── README.md
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml
```

8 directories, 8 files

It is really only the top level we are worried about; the `main.yml` files are just the default YAML files that are called for each part of the role:

- `README.md`: This is the file used to create any documentation about the role when the role is checked into a service such as GitHub. This file will be displayed along with the folder listing whenever someone browses to the `common` folder.
- `defaults`: This is where the default variables for the role are stored. These can be overridden by any variables with the same name called in the `vars` folder.
- `files`: This folder contains any static files we may wish to copy to the target

hosts using the `copy` module.

- `handlers`: Handlers are tasks that are executed once a playbook has been executed; typically, `handlers` are used to restart services when a configuration file has changed.
- `meta`: This contains information about the role and is used if the role was to be published to Ansible Galaxy.
- `tasks`: This is where the bulk of the work happens.
- `templates`: This folder contains the Jinja2 templates used by the `template` module.
- `tests`: Used to store any tests for the module.
- `vars`: You can override any of the variables defined in the `default` folder using the variables defined here; variables defined here can also be overridden by any variables loaded from the `group_vars` folder and the top level of the playbook.

Let's make a start by adding some tasks.

Updating packages

First of all, let's update our server by adding the following to the beginning of the `roles/common/tasks/main.yml` file:

- name: update all of the installed packages

`yum:`

`name: "*"`

`state: "latest"`

`update_cache: "yes"`

You will notice that there is a difference from when we last ran `yum` to update all of the installed packages and that is what we are now doing is starting the task using the `name` key, this will print out the content of the value we assigned to the `name` key when the playbook runs, which will give us a better idea of what is going on during the playbook run.

Installing common packages

Now that we have updated the installed packages, let's install the packages we want to install on all of the Linux servers we will be launching:

```
- name: install the common packages
  yum:
    name: "{{ item }}"
    state: "installed"
  with_items: "{{ common_packages }}
```

As you can see, we are again using the `yum` module and we have added a descriptive name for the task. Rather than providing a list of packages in the task, we are using a variable called `common_packages`, which is defined in the `roles/common/defaults/main.yml` file as the following:

```
common_packages:
  - "ntp"
  - "ntpdate"
  - "vim-enhanced"
  - "git"
  - "unzip"
  - "policycoreutils-python"
  - "epel-release"
  - "https://centos7.iuscommunity.org/ius-release.rpm"
```

As you can see, we are installing `ntp` and `ntpdate`; we will be configuring `ntp` shortly. Next, we are installing `vim-enhanced` and `git` as they are always useful to have installed on a server. Then, we are installing the `policycoreutils-python` package, more on that later, before finally installing and enabling two additional `yum` repositories, EPEL and IUS.



Extra Packages for Enterprise Linux (EPEL) is a special interest group that maintains a collection of packages that are not part of the Red Hat Enterprise Linux core. EPEL packages are typically based on their Fedora counterparts and have been packaged so they will never conflict with, or replace, packages in the core Enterprise Linux distributions.

CentOS 7 ships with a package called `epel-release`, which enables the EPEL repository. However, there is no release package for IUS, so here, rather than using a package that is a part of the core CentOS repository, we are providing the full URL of the RPM file that enabled the IUS repository for CentOS 7.



The IUS Community Project is a collection of RPMs for Red Hat Enterprise Linux and compatible operating systems, such as CentOS, which aims to provide packages that are



I *Inline with Upstream Stable*, hence **IUS**. They provide packages for Apache, PHP, and MariaDB, which are all the latest release. The packages supplied by IUS adhere to the rules laid out in the SafeRepo Initiative, meaning they can be trusted.

Configuring NTP

Next up, we are copying the `ntp.conf` file from the `templates` folder, adding the list of NTP servers as we do, and then telling Ansible to restart NTP whenever the configuration file changes:

```
- name: copy the ntp.conf to /etc/ntp.conf
  template:
    src: "ntp.conf.j2"
    dest: "/etc/ntp.conf"
  notify: "restart ntp"
```

The template file can be found at `roles/common/templates/ntp.conf.j2`:

```
# {{ ansible_managed }}
driftfile /var/lib/ntp/drift
restrict default nomodify notrap nopeer noquery
restrict 127.0.0.1
restrict ::1
{% for item in ntp_servers %}
server {{ item }} iburst
{% endfor %}
includefile /etc/ntp/crypto/pw
keys /etc/ntp/keys
disable monitor
```

As you can see, we are using the `ntp_servers` variable; this is stored in the `roles/common/defaults/main.yml` file:

```
ntp_servers:
  - "0.centos.pool.ntp.org"
  - "1.centos.pool.ntp.org"
  - "2.centos.pool.ntp.org"
  - "3.centos.pool.ntp.org"
```

Finally, the following task has been added to `roles/common/handlers/main.yml`:

```
- name: "restart ntp"
  service:
    name: "ntpd"
    state: "restarted"
```

While we have notified the handler here, NTP will not be restarted to the end of the playbook run along with any other tasks we have notified.

Creating a user

The final part of the common role is to add a user called `lamp` and add our public key to the user. Before we look at the task, let's look at the variable we will be using, which is defined in `roles/common/defaults/main.yml`:

```
users:
  - { name: "lamp", group: "lamp", state: "present", key: "{{ lookup('file',
  '~/.ssh/id_rsa.pub') }}" }
```

As you can see, we are providing three bits of information:

- `name`: This is the name of the user we want to create
- `group`: This is the group we want to add our user to
- `state`: If we want the user to be present or absent
- `key`: Here, we are using an Ansible lookup task to read the content of the file at `~/.ssh/id_rsa.pub` and use that as the value

The task in the `roles/common/tasks/main.yml` file for creating the user is split into three parts; the first part uses the `group` module to create the group:

```
- name: add group for our users
  group:
    name: "{{ item.group }}"
    state: "{{ item.state }}"
  with_items: "{{ users }}"
```

As you can see, we are using `with_items` to load in the `users` variable, as the variable contains three different items, only two of which are being used here. We can just name them, so here we are using `item.group` and `item.state`.

The second part of the task creates the user using the `user` module, as you can see:

```
- name: add users to our group
  user:
    name: "{{ item.name }}"
    group: "{{ item.group }}"
    comment: "{{ item.name }}"
    state: "{{ item.state }}"
  with_items: "{{ users }}"
```

The final part of the task adds the user's public key to the authorized key file

using the `authorized_key` module:

```
- name: add keys to our users
  authorized_key:
    user: "{{ item.name }}"
    key: "{{ item.key }}"
  with_items: "{{ users }}"
```

As you can see, we are using the `item.name` and `item.key` variables this time. This module creates a file called `.ssh/authorized_keys` in the user's home folder, which is defined by `item.name`, and then places the content of `item.key` in there, giving the holder of the private portion of the key access to the user we have just created.

Running the role

First of all, let's launch the CentOS 7 Vagrant box by running one of the following commands:

```
| $ vagrant up  
| $ vagrant up --provider=vmware_fusion
```

Now that we have our server, we need to update the host inventory; in the `production` file, enter the following:

```
box ansible_host=192.168.50.4.nip.io  
[boxes]  
box  
  
[boxes:vars]  
ansible_connection=ssh  
ansible_user=vagrant  
ansible_private_key_file=~/ssh/id_rsa  
host_key_checking=False
```

Finally, we need something that will execute our role. Add the following content to the `site.yml` file:

```
---  
- hosts: boxes  
  gather_facts: true  
  become: yes  
  become_method: sudo  
  
  vars_files:  
    - group_vars/common.yml  
  
  roles:  
    - roles/common
```

Now that we have our playbook file ready, we can run it against our Vagrant box by running:

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

After a few minutes, you should see something similar to the following output:

```
| PLAY [boxes]  
*****
```

```

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

TASK [roles/common : update all of the installed packages]
*****
changed: [box]

TASK [roles/common : install the common packages]
*****
changed: [box] => (item=[u'ntp', u'ntpdate', u'vim-enhanced', u'git', u'unzip',
u'policycoreutils-python', u'epel-release', u'https://centos7.iuscommunity.org/ius-
release.rpm'])

TASK [roles/common : copy the ntp.conf to /etc/ntp.conf]
*****
changed: [box]

TASK [roles/common : add group for our users]
*****
changed: [box] => (item={u'state': u'present', u'group': u'lamp', u'name': u'lamp',
u'key': u'ssh-rsa'
AAAAAB3NzaC1yc2EAAAQABAAQDmuoFR01i/Yf3HATl9c3sufJvghTFgYzK/Zt29JiTqWlSQhmXhNNTh6iIE
russmckendrick@me.com'})

TASK [roles/common : add users to our group]
*****
changed: [box] => (item={u'state': u'present', u'group': u'lamp', u'name': u'lamp',
u'key': u'ssh-rsa'
AAAAAB3NzaC1yc2EAAAQABAAQDmuoFR01i/Yf3HATl9c3sufJvghTFgYzK/Zt29JiTqWlSQhmXhNNTh6iIE
russmckendrick@me.com})

TASK [roles/common : add keys to our users]
*****
changed: [box] => (item={u'state': u'present', u'group': u'lamp', u'name': u'lamp',
u'key': u'ssh-rsa'
AAAAAB3NzaC1yc2EAAAQABAAQDmuoFR01i/Yf3HATl9c3sufJvghTFgYzK/Zt29JiTqWlSQhmXhNNTh6iIE
russmckendrick@me.com})

RUNNING HANDLER [roles/common : restart ntp]
*****
changed: [box]

PLAY RECAP
*****
box : ok=8 changed=7 unreachable=0 failed=0

```

As you can see, everything has been installed and configured as expected. Rerunning the playbook gives these results:

```

PLAY [boxes]
*****
TASK [Gathering Facts]
*****
ok: [box]

TASK [roles/common : update all of the installed packages]
*****
ok: [box]

TASK [roles/common : install the common packages]

```

```
*****
ok: [box] => (item=[u'ntp', u'ntpdate', u'vim-enhanced', u'git', u'unzip',
u'policycoreutils-python', u'epel-release', u'https://centos7.iuscommunity.org/ius-
release.rpm'])

TASK [roles/common : copy the ntp.conf to /etc/ntp.conf]
*****
ok: [box]

TASK [roles/common : add group for our users]
*****
ok: [box] => (item={u'state': u'present', u'group': u'lamp', u'name': u'lamp', u'key':
u'ssh-rsa
AAAAAB3NzaC1yc2EAAAQABAAQDmuoFR01i/Yf3HATl9c3sufJvghTFgYzK/Zt29JiTqW1SQhmXhNNTh6iIE
russmckendrick@me.com'})}

TASK [roles/common : add users to our group]
*****
ok: [box] => (item={u'state': u'present', u'group': u'lamp', u'name': u'lamp', u'key':
u'ssh-rsa
AAAAAB3NzaC1yc2EAAAQABAAQDmuoFR01i/Yf3HATl9c3sufJvghTFgYzK/Zt29JiTqW1SQhmXhNNTh6iIE
russmckendrick@me.com'})}

TASK [roles/common : add keys to our users]
*****
ok: [box] => (item={u'state': u'present', u'group': u'lamp', u'name': u'lamp', u'key':
u'ssh-rsa
AAAAAB3NzaC1yc2EAAAQABAAQDmuoFR01i/Yf3HATl9c3sufJvghTFgYzK/Zt29JiTqW1SQhmXhNNTh6iIE
russmckendrick@me.com'})}

PLAY RECAP
*****
box : ok=7    changed=0    unreachable=0    failed=0
```

As you can see, we have skipped the task which restarts NTP and there are no additional packages or updates to install, nor are there any changes to the user or group we created. Now that we have our basic packages updated and installed, and our base operating system configured, we are ready to install Apache.

Apache

At the moment, we have no role for Apache, so let's create one using the following command:

```
| $ ansible-galaxy init roles/apache
```

As before, this will create the basic scaffold for our Apache role.

Installing Apache

The first task we are going to add is one that installs the basic Apache packages. In `roles/apache/tasks/main.yml`, add the following:

```
- name: install the apache packages
  yum:
    name: "{{ item }}"
    state: "installed"
  with_items: "{{ apache_packages }}"
```

As you may have already guessed, the defaults for `apache_packages` can be found in `roles/apache/defaults/main.yml`:

```
apache_packages:
  - "httpd24u"
  - "httpd24u-filesystem"
  - "httpd24u-tools"
  - "httpd24u-mod_ssl"
  - "openssl"
  - "openssl-libs"
```

This installs the latest Apache 2.4 package from IUS, along with some of the supporting tools we will need. Once installed, we will now need to configure Apache.

Configuring Apache

You may have wondered why we created a user called `lamp` in the previous section; we are going to be hosting our website for this user. The first aspect in getting the user ready to host our website is to add the user to the `apache_group`. To do this, we need to run the following task:

```
- name: Add user to apache group
  user:
    name: "{{ item.name }}"
    groups: "{{ apache_group }}"
    append: yes
  with_items: "{{ users }}"
```

There are two things to point out here. The first is that we are using the `users` variable from the previous role, which is still available to use within the playbook run, and the second thing is that we have added a variable called `apache_group` to `roles/apache/defaults/main.yml`:

```
| apache_group: "apache"
```

Now that our user is in the `apache_group`, let's create what will be the document root for our website:

```
- name: create the document root for our website
  file:
    dest: "{{ document_root }}"
    state: "directory"
    mode: "0755"
    owner: "{{ users.0.name }}"
    group: "{{ apache_group }}"
```

As you can see, this is using a few new variables, along with a new way of accessing an old one. Let's address `users.0.name` first, because we have defined the `users` as a list. It is possible that more than one user could be being added during the playbook run, as we only want to create one document root and assign it to a single virtual host we are using the first user in the list which registered under the `user` variable, this is where the `0` comes in.

The `document_root` variable is also constructed using this principle; these are the two variables in the `roles/apache/defaults/main.yml` file that will help make up the full document root:

```
| web_root: "web"
| document_root: "/home/{{ users.0.name }}/{{ web_root }}"
```

This will give our document root a path of `/home/lamp/web/` on the Vagrant box, assuming we do not override any of the variable names in our main playbook.

We also need to change the permissions on the lamp user's home folder to allow us to execute scripts; to do this, the following task is called:

```
- name: set the permissions on the user folder
  file:
    dest: "/home/{{ users.0.name }}/"
    state: "directory"
    mode: "0755"
    owner: "{{ users.0.name }}"
```

Next, we need to put our Apache virtual host in place; this will serve our web page whenever we put the name of our host in a browser. To do this, we will be using a template file stored in `roles/apache/templates/vhost.conf.j2` that uses the variables we have already defined along with two more:

```
# {{ ansible_managed }}
<VirtualHost *:80>
  ServerName {{ ansible_nodename }}
  DocumentRoot {{ document_root }}
  DirectoryIndex {{ index_file }}
  <Directory {{ document_root }}>
    AllowOverride All
    Require all granted
  </Directory>
</VirtualHost>
```

The `index_file` variable in `roles/apache/default/main.yml` looks like the following:

```
| index_file: index.html
```

There is also the `ansible_nodename` variable; this is one of the variables gathered from the host machine when the `setup` module first runs. The task to deploy the template is as follows:

```
- name: copy the vhost.conf to /etc/httpd/conf.d/
  template:
    src: "vhost.conf.j2"
    dest: "/etc/httpd/conf.d/vhost.conf"
  notify: "restart httpd"
```

The task that restarts Apache can be found in `roles/apache/handlers/main.yml` and looks like the following:

```
| - name: "restart httpd"
|   service:
|     name: "httpd"
|     state: "restarted"
```

Now that we have Apache installed and configured, we need to allow Apache to use the web root, which is stored in `/home/`. To do this, we need to tweak the SELinux permissions.

Configuring SELinux

One of the packages we installed during the last section was `policycoreutils-python`. This allows us to configure SELinux using Python, and therefore Ansible.



Security-Enhanced Linux (SELinux) was developed by Red Hat and the NSA. It provides a mechanism for supporting access control security policies at the kernel level. These include mandatory access controls used by the United States Department of Defense.

By default, the Vagrant box we are using ships with SELinux enabled. Rather than simply stopping SELinux, we can just permit Apache to run outside of its default `/var/www/`. To do this, we need to add the following to our role:

- name: set the selinux allowing `httpd_t` to be permissive

```
selinux_permissive:  
  name: httpd_t  
  permissive: true
```

Now that Apache is allowed to serve content from our user directory, we can add an `index.html` file, so we have something other than the default Apache page to serve.

Copying an HTML file

The final task is to copy an `index.html` file to our web root, so we have something to serve with our newly installed Apache server. The task to do this uses the `template` module:

```
- name: copy the test HTML page to the document root
  template:
    src: "index.html.j2"
    dest: "{{ document_root }} / index.html"
    mode: "0644"
    owner: "{{ users.0.name }}"
    group: "{{ apache_group }}"
  when: html_deploy == true
```

As you can see, we are loading a template called `index.html.j2`, which contains the following content:

```
<!--{{ ansible_managed }}-->
<!doctype html>
<title>{{ html_heading }}</title>
<style>
  body { text-align: center; padding: 150px; }
  h1 { font-size: 50px; }
  body { font: 20px Helvetica, sans-serif; color: #333; }
  article { display: block; text-align: left; width: 650px;
    margin: 0 auto; }
</style>
<article>
  <h1>{{ html_heading }}</h1>
  <div>
    <p>{{ html_body }}</p>
  </div>
</article>
```

We are using two variables in our template; both of these can found in the `roles/apache/defaults/main.yml` file along with the variable:

```
html_deploy: true
html_heading: "Success !!!"
html_body: |
  This HTML page has been deployed using Ansible to
  <b>{{ ansible_nodename }}</b>. <br>
  The user is <b>{{ users.0.name }}</b> who is in the
  <b>{{ apache_group }}</b> group. <br>
  The webroot is <b>{{ document_root }}</b>, the default index file is
  <b>{{ index_file }}</b>. <br>
```

As part of the task, we have the following line:

```
|when: html_deploy == true
```

This means that the task will only be executed if `html_deploy` equals `true`. If it is anything else, then the task will be skipped. We will be looking at this later in the chapter, but for now, we want the page to be deployed, so we will keep the default value defined in the `apache/defaults/main.yml` file.

The final thing to point out before we run the role is the `html_body` variable. As you can see the content of the variable is spread over three lines. This is done using the `|` character after the variable name; this helps make your variable files readable and also allows you to start distributing items such as keys or certificates as variables, while also allowing you to encode them using vault.

Running the role

Now that the role for installing and configuring Apache is complete, we can add it to our playbook:

```
---
- hosts: boxes
  gather_facts: true
  become: yes
  become_method: sudo

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/common
    - roles/apache
```

Following on from the playbook in the previous section, we can simply rerun the following command:

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

This will work through the common role before executing the apache role. I have truncated the output for the common role in the playbook run here:

```
PLAY [boxes]
*****
TASK [Gathering Facts]
*****
ok: [box]

TASK [roles/common : update all of the installed packages]
*****
ok: [box]

TASK [roles/common : install the common packages]
*****
ok: [box]

TASK [roles/common : copy the ntp.conf to /etc/ntp.conf]
*****
ok: [box]

TASK [roles/common : add group for our users]
*****
ok: [box]
TASK [roles/common : add users to our group]
*****
ok: [box]
```

```

TASK [roles/common : add keys to our users]
*****
ok: [box]

TASK [roles/apache : install the apache packages]
*****
changed: [box] => (item=[u'httpd24u', u'httpd24u-filesystem', u'httpd24u-tools',
u'httpd24u-mod_ssl', u'openssl', u'openssl-libs'])

TASK [roles/apache : Add user to apache group]
*****
changed: [box] => (item={u'state': u'present', u'group': u'lamp', u'name': u'lamp',
u'key': u'ssh-rsa
AAAAAB3NzaC1yc2EAAAQABAAQDmuoFR01i/Yf3HATl9c3sufJvghTFgYzK/Zt29JiTqW1SQhmXhNNTh6iIE
russmckendrick@me.com'})}

TASK [roles/apache : create the document root for our website]
*****
changed: [box]

TASK [roles/apache : set the permissions on the user folder]
*****
changed: [box]

TASK [roles/apache : copy the vhost.conf to /etc/httpd/conf.d/]
*****
changed: [box]

TASK [roles/apache : set the selinux allowing httpd_t to be permissive]
*****
changed: [box]

TASK [roles/apache : copy the test HTML page to the document root]
*****
changed: [box]

RUNNING HANDLER [roles/apache : restart httpd]
*****
changed: [box]

PLAY RECAP
*****
box : ok=15 changed=8 unreachable=0 failed=0

```

Opening `http://192.168.50.4.nip.io/` in our browser should give us a page that looks like the following screenshot:



Success !!!

This HTML page has been deployed using Ansible to
192.168.50.4.nip.io.
 The user is **lamp** who is in the **apache** group.
 The webroot is **/home/lamp/web**, the default index file is **index.html**.

As you can see, the template has picked up all of the variables we defined; the source for the page looks like this:

```
<!--Ansible managed-->
<!doctype html>
<title>Success !!!</title>
<style>
  body { text-align: center; padding: 150px; }
  h1 { font-size: 50px; }
  body { font: 20px Helvetica, sans-serif; color: #333; }
  article { display: block; text-align: left; width: 650px;
    margin: 0 auto; }
</style>
<article>
  <h1>Success !!!</h1>
  <div>
    <p>This HTML page has been deployed using Ansible to
    <b>192.168.50.4.nip.io</b>.<br>
    The user is <b>lamp</b> who is in the <b>apache</b> group.<br>
    The webroot is <b>/home/lamp/web</b>, the default index file is
    <b>index.html</b>.<br></p>
  </div>
</article>
```

If we were to rerun the playbook, we should see the following results:

```
PLAY RECAP
*****
box : ok=14 changed=0 unreachable=0 failed=0
```

As you can see, there are 14 tasks that are `ok`, and nothing has changed.

MariaDB

Next, we are going to install and configure MariaDB, the database component of our LAMP stack.



MariaDB is a fork of MySQL. Its development has been led by some of the original developers of MySQL; they created the fork after concerns surrounding the licensing of MySQL after it was acquired by Oracle.

The first step is to create the files we are going to need for the role; again, we will use the `ansible-galaxy init` command to bootstrap the role files: **\$ ansible-galaxy init roles/mariadb**

Installing MariaDB

As we have been using the IUS repository for other packages in our playbook, it would make sense to install the latest version of MariaDB from there. However, there is a conflict we need to resolve first.

As part of the base Vagrant box installation, Postfix, the mail server, is installed. Postfix requires the `mariadb-libs` package as a dependency, but having this package installed is going to cause a conflict with the later version of the package we want to install. The solution to this problem is to remove the `mariadb-libs` package and, then install the packages we need, along with Postfix, which is removed when we uninstall `mariadb-libs`.

The first task in the role, which we need to add to `roles/mariadb/tasks/mail.yml`, looks like the following:

```
- name: remove the packages so that they can be replaced
  yum:
    name: "{{ item }}"
    state: "absent"
  with_items: "{{ mariadb_packages_remove }}"
```

As you may have already suspected, `mariadb_packages_remove` is defined in the `roles/mariadb/defaults/main.yml` file:

```
mariadb_packages_remove:
  - "mariadb-libs.x86_64"
```

As you can see, we are using the full package name. We need to do this because if we simply used `mariadb-libs`, then the newly installed package would be removed during each playbook run. This is bad as this task would also uninstall all of the MariaDB packages we are going to be installing next, which, if we have a live database running, would be a disaster!

To install the later version of MariaDB, we need to add the following task:

```
- name: install the mariadb packages
  yum:
    name: "{{ item }}"
    state: "installed"
  with_items: "{{ mariadb_packages }}"
```

The `mariadb_packages` variable, which again can be found in the defaults folder, looks like the following:

```
mariadb_packages:
  - "mariadb101u"
  - "mariadb101u-server"
  - "mariadb101u-config"
  - "mariadb101u-common"
  - "mariadb101u-libs"
  - "MySQL-python"
  - "postfix"
```

We are installing the packages for MariaDB, along with Postfix, which was removed during the last task. We are also installing the `MySQL-python` package, which will allow Ansible to interact with our MariaDB installation.

By default, MariaDB does not start as part of the installation process. Typically, we would use a handler to start the service as part of the playbook run, and, as we have learned from the previous sections, the handlers run at the very end of the playbook execution. This would not be a problem if we didn't need to interact with the MariaDB service to configure it. To get around this, we need to add the following task to our role:

```
- name: start mariadb
  service:
    name: "mariadb"
    state: "started"
    enabled: "yes"
```

This makes sure that MariaDB is running, as well as configuring the service to start on boot.

Configuring MariaDB

Now that MariaDB is installed and running, we make a start on getting it configured. Our default installation of MariaDB has no root password defined, so that should be the first thing we set up. We can do this using the `mysql_user` module:

```
- name: change mysql root password
  mysql_user:
    name: "{{ mariadb_root_username }}"
    host: "{{ item }}"
    password: "{{ mariadb_root_password }}"
    check_implicit_admin: "yes"
    priv: "*.*:ALL,GRANT"
    with_items: "{{ mariadb_hosts }}
```

As you can see, we have a few different variables in use; these are defined in `roles/mariadb/defaults/main.yml` as:

```
mariadb_root_username: "root"
mariadb_root_password: "Pa55W0rd123"
mariadb_hosts:
  - "127.0.0.1"
  - "::1"
  - "{{ ansible_nodename }}"
  - "%"
  - "localhost"
```

The order of the hosts in `mariadb_hosts` is important; if `localhost` is not the last host changed, then Ansible will give an error with a message about not being able to connect to MariaDB. This is because we are using the fact that MariaDB does not ship with a default root password to actually set the root password.

Now, once we have configured the root user's password, we still want to be able to connect to MySQL. I prefer to set up a `~/.my.cnf` file under the root users folder. This can be done in Ansible as follows:

```
- name: set up .my.cnf file
  template:
    src: "my.cnf.j2"
    dest: "~/.my.cnf"
```

The template file can be found at `lamp/roles/mariadb/templates/my.cnf.j2`; it contains the following content:

```
| # {{ ansible_managed }}
```

```
| [client]
```

```
| password='{{ mariadb_root_password }}'
```

Once in place, this means that the system root user—not to be confused with the root user we just set up within MariaDB—will have direct access to MariaDB without having to provide a password. Next up, we can delete the anonymous user, which is created by default. Again, we will use the `mysql_user` module for this:

```
- name: delete anonymous MySQL user
```

```
  mysql_user:
```

```
    user: ""
```

```
    host: "{{ item }}"
    state: "absent"
  with_items: "{{ mariadb_hosts }}"
```

Finally, there is a `test` database created. As we will be creating our own, let's remove this as well, this time using the `mysql_db` module:

```
- name: remove the MySQL test database
  mysql_db:
    db: "test"
    state: "absent"
```

These configuration tasks are the equivalent of running the `mysql_secure_installation` command.

Importing a sample database

Now that our MariaDB installation is complete, we should do something with it. There are a few sample databases available on GitHub. Let's look at importing the employee database provided by datacharmer. We will be using a slightly altered version of the SQL dumps, but more on that later in the section.

We are going to be using nested variables for this part of the playbook; these can be found in `mariadb/defaults/main.yml`:

```
mariadb_sample_database:  
  create_database: true  
  source_url: "https://github.com/russmckendrick/test_db/archive/master.zip"  
  path: "/tmp/test_db-master"  
  db_name: "employees"  
  db_user: "employees"  
  db_password: "employees"  
  dump_files:  
    - "employees.sql"  
    - "load_departments.dump"  
    - "load_employees.dump"  
    - "load_dept_emp.dump"  
    - "load_dept_manager.dump"  
    - "load_titles.dump"  
    - "load_salaries1.dump"  
    - "load_salaries2.dump"  
    - "load_salaries3.dump"  
    - "show_elapsed.sql"
```

When we call these variables, they will need to be prefixed with `mariadb_sample_database`. For example, whenever we need to use the `db_name` variable, we will need to use `mariadb_sample_database.db_name`. As when we copied the HTML file in the previous section, we will be adding a condition using `when` to each of the tasks, which will mean they can be skipped if required.

The first thing we will need to do is download a copy of the dump files from GitHub and uncompress them. To do this, we will be using the `unarchive` module:

```
- name: download and unarchive the sample database data  
  unarchive:  
    src: "{{ mariadb_sample_database.source_url }}"  
    dest: "/tmp"  
    remote_src: "yes"  
    when: mariadb_sample_database.create_database == true
```

We are grabbing the file from a remote location, which is the URL

`mariadb_sample_database.source_url`, and uncompressing it in `/tmp`. As we have set `remote_src` to `yes`, Ansible knows it has to download the file from a remote source. If we didn't provide a full URL, it would have attempted to have copied the file from our control host.

The next two tasks use the `mysql_db` and `mysql_user` modules to create the database and a user who has access to it:

```
- name: create the sample database
  mysql_db:
    db: "{{ mariadb_sample_database.db_name }}"
    state: "present"
  when: mariadb_sample_database.create_database == true

- name: create the user for the sample database
  mysql_user:
    name: "{{ mariadb_sample_database.db_user }}"
    password: "{{ mariadb_sample_database.db_password }}"
    priv: "{{ mariadb_sample_database.db_name }}.*:ALL"
    state: "present"
  with_items: "{{ mariadb_hosts }}"
  when: mariadb_sample_database.create_database == true
```

The final part of the playbook imports the MySQL dumps into the database; however, before we import the files, we should first check if the dumps have already been imported. If we do not perform this check every time we run the playbook, the dumps will be imported. To check whether the data has been imported already, we are going to use the `stat` module; this checks the presence of a file and gathers information on it.

If we have already imported the data, there will be a file called `employees.frm` in the `/var/lib/mysql/employees` folder, so let's check for that:

```
- name: check to see if we need to import the sample database dumps
  stat:
    path: "/var/lib/mysql/{{ mariadb_sample_database.db_name }}/{{ mariadb_sample_database.db_name }}.frm"
    register: db_imported
  when: mariadb_sample_database.create_database == true
```

Now we know whether we need to import the database dumps or not, we can proceed with the final task, which imports the database dumps listed in `mariadb_sample_database.dump_files` if the following conditions are met:

- Is the variable `db_imported` defined? If not, then we are skipping importing the sample database and should skip this task.
- Does `db_imported.stat.exists` equal `false`? If so, then the file does not exist, and

we should import the data.

The task itself uses the `mysql_db` module to import the data:

```
- name: import the sample database
  mysql_db:
    name: "{{ mariadb_sample_database.db_name }}"
    state: "import"
    target: "{{ mariadb_sample_database.path }}/{{ item }}"
    with_items: "{{ mariadb_sample_database.dump_files }}"
    when: db_imported is defined and db_imported.stat.exists == false
```

That completes the import of the sample database into our MariaDB installation; let's now run the playbook and call the role.

Running the role

Now that we have our role written, we can add it to our playbook:

```
---
- hosts: boxes
  gather_facts: true
  become: yes
  become_method: sudo

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/common
    - roles/apache
    - roles/mariadb
```

Again, we can rerun the playbook using:

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

This will work through the common and Apache roles before moving on to the MariaDB equivalent. This playbook output starts just before the the MariaDB role begins:

```
TASK [roles/apache : set the selinux allowing httpd_t to be permissive]
*****
ok: [box]

TASK [roles/apache : copy the test HTML page to the document root]
*****
ok: [box]

TASK [roles/mariadb : remove the packages so that they can be replaced]
*****
changed: [box] => (item=[u'mariadb-libs.x86_64'])

TASK [roles/mariadb : install the mariadb packages]
*****
changed: [box] => (item=[u'mariadb101u', u'mariadb101u-server', u'mariadb101u-config', u'mariadb101u-common', u'mariadb101u-libs', u'MySQL-python', u' postfix'])

TASK [roles/mariadb : start mariadb]
*****
changed: [box]

TASK [roles/mariadb : change mysql root password]
*****
changed: [box] => (item=127.0.0.1)
changed: [box] => (item=:1)
changed: [box] => (item=192.168.50.4.nip.io)
```

```

changed: [box] => (item=%)
changed: [box] => (item=localhost)

TASK [roles/mariadb : set up .my.cnf file]
*****
changed: [box]

TASK [roles/mariadb : delete anonymous MySQL user]
*****
ok: [box] => (item=127.0.0.1)
ok: [box] => (item=:1)
changed: [box] => (item=192.168.50.4.nip.io)
ok: [box] => (item=%)
changed: [box] => (item=localhost)

TASK [roles/mariadb : remove the MySQL test database]
*****
changed: [box]

TASK [roles/mariadb : download and unarchive the sample database data]
*****
changed: [box]

TASK [roles/mariadb : create the sample database]
*****
changed: [box]

TASK [roles/mariadb : create the user for the sample database]
*****
changed: [box] => (item=127.0.0.1)
ok: [box] => (item=:1)
ok: [box] => (item=192.168.50.4.nip.io)
ok: [box] => (item=%)
ok: [box] => (item=localhost)

TASK [roles/mariadb : check to see if we need to import the sample database dumps]
*****
ok: [box]

TASK [roles/mariadb : import the sample database]
*****
changed: [box] => (item=employees.sql)
changed: [box] => (item=load_departments.dump)
changed: [box] => (item=load_employees.dump)
changed: [box] => (item=load_dept_emp.dump)
changed: [box] => (item=load_dept_manager.dump)
changed: [box] => (item=load_titles.dump)
changed: [box] => (item=load_salaries1.dump)
changed: [box] => (item=load_salaries2.dump)
changed: [box] => (item=load_salaries3.dump)
changed: [box] => (item=show_elapsed.sql)

PLAY RECAP
*****
box : ok=26 changed=11 unreachable=0 failed=0

```

If we were to rerun the playbook, the final part of the playbook run returns the following:

```

TASK [roles/mariadb : download and unarchive the sample database data]
*****
ok: [box]

```

```

TASK [roles/mariadb : create the sample database]
*****
ok: [box]

TASK [roles/mariadb : create the user for the sample database]
*****
ok: [box] => (item=127.0.0.1)
ok: [box] => (item=:1)
ok: [box] => (item=192.168.50.4.nip.io)
ok: [box] => (item=%)
ok: [box] => (item=localhost)

TASK [roles/mariadb : check to see if we need to import the sample database dumps]
*****
ok: [box]

TASK [roles/mariadb : import the sample database]
*****
skipping: [box] => (item=employees.sql)
skipping: [box] => (item=load_departments.dump)
skipping: [box] => (item=load_employees.dump)
skipping: [box] => (item=load_dept_emp.dump)
skipping: [box] => (item=load_dept_manager.dump)
skipping: [box] => (item=load_titles.dump)
skipping: [box] => (item=load_salaries1.dump)
skipping: [box] => (item=load_salaries2.dump)
skipping: [box] => (item=load_salaries3.dump)
skipping: [box] => (item=show_elapsed.sql)

PLAY RECAP
*****
box : ok=25 changed=0 unreachable=0 failed=0

```

As you can see, the checks we put in place to not reimport the database dumps worked as expected. We can test our MariaDB installation using a tool such as Sequel Pro or MySQL Workbench; just use the following host and credentials to connect:

- Host: 192.168.50.4.nip.io
- Port: 3306
- Username: root
- Password: Pa55W0rd123

The following screenshot is taken from Sequel Pro and shows the `employees` table, which we imported into the `employees` database :

(MySQL 5.5.5-10.1.31-MariaDB) 192.168.50.4.nip.io/employees/employees

employees	Structure	Content	Relations	Triggers	Table Info	Query	Table History	Users	Console																																																																																																																																																																								
TABLES & VIEWS	Search: emp_no = <input type="button" value="Search"/> <input type="button" value="Filter"/> <table border="1"> <thead> <tr> <th>emp_no</th> <th>birth_date</th> <th>first_name</th> <th>last_name</th> <th>gender</th> <th>hire_date</th> </tr> </thead> <tbody> <tr><td>10001</td><td>1953-09-02</td><td>Georgi</td><td>Facello</td><td>M</td><td>1986-06-26</td></tr> <tr><td>10002</td><td>1964-06-02</td><td>Bezalel</td><td>Simmel</td><td>F</td><td>1985-11-21</td></tr> <tr><td>10003</td><td>1959-12-03</td><td>Parto</td><td>Bamford</td><td>M</td><td>1986-08-28</td></tr> <tr><td>10004</td><td>1954-05-01</td><td>Christian</td><td>Koblick</td><td>M</td><td>1986-12-01</td></tr> <tr><td>10005</td><td>1955-01-21</td><td>Kyoichi</td><td>Maliniak</td><td>M</td><td>1989-09-12</td></tr> <tr><td>10006</td><td>1953-04-20</td><td>Anneke</td><td>Preusig</td><td>F</td><td>1989-06-02</td></tr> <tr><td>10007</td><td>1957-05-23</td><td>Tsvetan</td><td>Zielinski</td><td>F</td><td>1989-02-10</td></tr> <tr><td>10008</td><td>1958-02-19</td><td>Saniya</td><td>Kalloufi</td><td>M</td><td>1994-09-15</td></tr> <tr><td>10009</td><td>1952-04-19</td><td>Sumant</td><td>Peac</td><td>F</td><td>1985-02-18</td></tr> <tr><td>10010</td><td>1963-06-01</td><td>Duangkaew</td><td>Piveteau</td><td>F</td><td>1989-08-24</td></tr> <tr><td>10011</td><td>1953-11-07</td><td>Mary</td><td>Sluis</td><td>F</td><td>1990-01-22</td></tr> <tr><td>10012</td><td>1960-10-04</td><td>Patricia</td><td>Bridgland</td><td>M</td><td>1992-12-18</td></tr> <tr><td>10013</td><td>1963-06-07</td><td>Eberhardt</td><td>Terkki</td><td>M</td><td>1985-10-20</td></tr> <tr><td>10014</td><td>1956-02-12</td><td>Berni</td><td>Genin</td><td>M</td><td>1987-03-11</td></tr> <tr><td>10015</td><td>1959-08-19</td><td>Guoxiang</td><td>Nooteboom</td><td>M</td><td>1987-07-02</td></tr> <tr><td>10016</td><td>1961-05-02</td><td>Kazuhito</td><td>Cappelletti</td><td>M</td><td>1995-01-27</td></tr> <tr><td>10017</td><td>1958-07-06</td><td>Cristinel</td><td>Bouloucos</td><td>F</td><td>1993-08-03</td></tr> <tr><td>10018</td><td>1954-06-19</td><td>Kazuhide</td><td>Peña</td><td>F</td><td>1987-04-03</td></tr> <tr><td>10019</td><td>1953-01-23</td><td>Lillian</td><td>Haddadi</td><td>M</td><td>1999-04-30</td></tr> <tr><td>10020</td><td>1952-12-24</td><td>Mayuko</td><td>Warwick</td><td>M</td><td>1991-01-26</td></tr> <tr><td>10021</td><td>1960-02-20</td><td>Ramzi</td><td>Erde</td><td>M</td><td>1988-02-10</td></tr> <tr><td>10022</td><td>1952-07-08</td><td>Shahaf</td><td>Famili</td><td>M</td><td>1995-08-22</td></tr> <tr><td>10023</td><td>1953-09-29</td><td>Bojan</td><td>Montemayor</td><td>F</td><td>1989-12-17</td></tr> <tr><td>10024</td><td>1958-09-05</td><td>Suzette</td><td>Pettey</td><td>F</td><td>1997-05-19</td></tr> <tr><td>10025</td><td>1958-10-31</td><td>Prasadram</td><td>Heyers</td><td>M</td><td>1987-08-17</td></tr> <tr><td>10026</td><td>1953-04-03</td><td>Yongqiao</td><td>Berztiss</td><td>M</td><td>1995-03-20</td></tr> <tr><td>10027</td><td>1962-07-10</td><td>Dilma</td><td>Reindl</td><td>F</td><td>1990-07-07</td></tr> </tbody> </table>									emp_no	birth_date	first_name	last_name	gender	hire_date	10001	1953-09-02	Georgi	Facello	M	1986-06-26	10002	1964-06-02	Bezalel	Simmel	F	1985-11-21	10003	1959-12-03	Parto	Bamford	M	1986-08-28	10004	1954-05-01	Christian	Koblick	M	1986-12-01	10005	1955-01-21	Kyoichi	Maliniak	M	1989-09-12	10006	1953-04-20	Anneke	Preusig	F	1989-06-02	10007	1957-05-23	Tsvetan	Zielinski	F	1989-02-10	10008	1958-02-19	Saniya	Kalloufi	M	1994-09-15	10009	1952-04-19	Sumant	Peac	F	1985-02-18	10010	1963-06-01	Duangkaew	Piveteau	F	1989-08-24	10011	1953-11-07	Mary	Sluis	F	1990-01-22	10012	1960-10-04	Patricia	Bridgland	M	1992-12-18	10013	1963-06-07	Eberhardt	Terkki	M	1985-10-20	10014	1956-02-12	Berni	Genin	M	1987-03-11	10015	1959-08-19	Guoxiang	Nooteboom	M	1987-07-02	10016	1961-05-02	Kazuhito	Cappelletti	M	1995-01-27	10017	1958-07-06	Cristinel	Bouloucos	F	1993-08-03	10018	1954-06-19	Kazuhide	Peña	F	1987-04-03	10019	1953-01-23	Lillian	Haddadi	M	1999-04-30	10020	1952-12-24	Mayuko	Warwick	M	1991-01-26	10021	1960-02-20	Ramzi	Erde	M	1988-02-10	10022	1952-07-08	Shahaf	Famili	M	1995-08-22	10023	1953-09-29	Bojan	Montemayor	F	1989-12-17	10024	1958-09-05	Suzette	Pettey	F	1997-05-19	10025	1958-10-31	Prasadram	Heyers	M	1987-08-17	10026	1953-04-03	Yongqiao	Berztiss	M	1995-03-20	10027	1962-07-10	Dilma	Reindl	F	1990-07-07
emp_no	birth_date	first_name	last_name	gender	hire_date																																																																																																																																																																												
10001	1953-09-02	Georgi	Facello	M	1986-06-26																																																																																																																																																																												
10002	1964-06-02	Bezalel	Simmel	F	1985-11-21																																																																																																																																																																												
10003	1959-12-03	Parto	Bamford	M	1986-08-28																																																																																																																																																																												
10004	1954-05-01	Christian	Koblick	M	1986-12-01																																																																																																																																																																												
10005	1955-01-21	Kyoichi	Maliniak	M	1989-09-12																																																																																																																																																																												
10006	1953-04-20	Anneke	Preusig	F	1989-06-02																																																																																																																																																																												
10007	1957-05-23	Tsvetan	Zielinski	F	1989-02-10																																																																																																																																																																												
10008	1958-02-19	Saniya	Kalloufi	M	1994-09-15																																																																																																																																																																												
10009	1952-04-19	Sumant	Peac	F	1985-02-18																																																																																																																																																																												
10010	1963-06-01	Duangkaew	Piveteau	F	1989-08-24																																																																																																																																																																												
10011	1953-11-07	Mary	Sluis	F	1990-01-22																																																																																																																																																																												
10012	1960-10-04	Patricia	Bridgland	M	1992-12-18																																																																																																																																																																												
10013	1963-06-07	Eberhardt	Terkki	M	1985-10-20																																																																																																																																																																												
10014	1956-02-12	Berni	Genin	M	1987-03-11																																																																																																																																																																												
10015	1959-08-19	Guoxiang	Nooteboom	M	1987-07-02																																																																																																																																																																												
10016	1961-05-02	Kazuhito	Cappelletti	M	1995-01-27																																																																																																																																																																												
10017	1958-07-06	Cristinel	Bouloucos	F	1993-08-03																																																																																																																																																																												
10018	1954-06-19	Kazuhide	Peña	F	1987-04-03																																																																																																																																																																												
10019	1953-01-23	Lillian	Haddadi	M	1999-04-30																																																																																																																																																																												
10020	1952-12-24	Mayuko	Warwick	M	1991-01-26																																																																																																																																																																												
10021	1960-02-20	Ramzi	Erde	M	1988-02-10																																																																																																																																																																												
10022	1952-07-08	Shahaf	Famili	M	1995-08-22																																																																																																																																																																												
10023	1953-09-29	Bojan	Montemayor	F	1989-12-17																																																																																																																																																																												
10024	1958-09-05	Suzette	Pettey	F	1997-05-19																																																																																																																																																																												
10025	1958-10-31	Prasadram	Heyers	M	1987-08-17																																																																																																																																																																												
10026	1953-04-03	Yongqiao	Berztiss	M	1995-03-20																																																																																																																																																																												
10027	1962-07-10	Dilma	Reindl	F	1990-07-07																																																																																																																																																																												
TABLE INFORMATION																																																																																																																																																																																	
<input checked="" type="checkbox"/> created: 02/03/2018 <input checked="" type="checkbox"/> engine: InnoDB <input checked="" type="checkbox"/> rows: ~298,892 <input checked="" type="checkbox"/> size: 14.5 MiB <input checked="" type="checkbox"/> encoding: latin1																																																																																																																																																																																	
<input type="button" value="+"/> <input type="button" value="*"/> <input type="button" value="C"/> <input type="button" value="▼"/> <input type="button" value="+"/> <input type="button" value="-"/> <input type="button" value="C"/> <input type="button" value="▼"/> <input type="button" value="Rows 1 - 1,000 of ~298,892 from table"/>																																																																																																																																																																																	

Now that we have MariaDB installed, configured, and some sample data imported, let's take a look at creating a role that installs PHP, the final component of our LAMP stack.

PHP

The final element of the stack we are putting together is PHP. As with the other three elements, we need to create a role using the `ansible-galaxy init` command: **\$ ansible-galaxy init roles/php**

As with the other parts of the stack, we are going to be using packages from the IUS repository; this will allow us to install the latest version of PHP, version 7.2.

Installing PHP

Like the previous three parts of the stack, we are going to start by installing the packages. As before, we are defining a variable in `roles/php/default/main.yml` that lists all of the packages we need:

```
php_packages:
  - "php72u"
  - "php72u-bcmath"
  - "php72u-cli"
  - "php72u-common"
  - "php72u-dba"
  - "php72u-fpm"
  - "php72u-fpm-nginx"
  - "php72u-gd"
  - "php72u-intl"
  - "php72u-json"
  - "php72u-mbstring"
  - "php72u-mysqli"
  - "php72u-odbc"
  - "php72u-pdo"
  - "php72u-process"
  - "php72u-snmp"
  - "php72u-soap"
  - "php72u-xml"
  - "php72u-xmlrpc"
```

This is installed by using the YUM module in `php/roles/tasks/main.yml`:

```
- name: install the php packages
  yum:
    name: "{{ item }}"
    state: "installed"
  with_items: "{{ php_packages }}"
  notify:
    - "restart php-fpm"
    - "restart httpd"
```

As you can see from this task, we are notifying two different handlers, the one for Apache and one for PHP-FPM. You may be thinking to yourself: why do we need to notify Apache?



FastCGI Process Manager (FPM) is a PHP FastCGI implementation that helps busy PHP websites run more efficiently. It also adds the ability to start PHP workers with different user and group IDs, which can listen on different ports using different `php.ini` files, allowing you to create pools of PHP workers to handle your load.

As we are installing the `php72u-fpm` package, we need to configure Apache to use the configuration put in place by the `php72u-fpm-nginx` package; if we don't, then

Apache will not load the configuration, which instructs it on how to interact with PHP-FPM.

The handler for PHP-FPM can be found in `roles/php/handlers/main.yml`, and it contains the following:

```
- name: "restart php-fpm"
  service:
    name: "php-fpm"
    state: "restarted"
    enabled: "yes"
```

That is kind of it for the PHP installation and configuration; we should now have a working PHP installation, and we can test this using a `phpinfo` file.

The phpinfo file

As with the Apache installation, we can add the option to upload a test file, in this case, a simple PHP file that calls the `php_info` function. This displays information about our PHP installation. The task to upload this file looks like the following:

- name: copy the test PHP page to the document root

copy:

```
src: "info.php"
dest: "{{ document_root }}/info.php"
mode: "0755"
owner: "{{ users.0.name }}"
group: "{{ apache_group }}"
when: php_info == true
```

As you can see, it is only being called when the following is set in `roles/php/default/main.yml`:

```
| php_info: true
```

The file we are copying to the host from our Ansible controller can be found in `roles/php/files/info.php`, and it contains the following three lines:

```
| <?php
|   phpinfo();
| ?>
```

While this demonstrates that PHP is installed and working, it isn't very interesting, so before we run our playbook, let's add a few more steps that tie all of the elements of our LAMP stack together.

Adminer

The final task for the playbook is going to be to install a PHP script called Adminer; this provides a PHP-powered interface for interacting with and managing your databases. There are three steps to install Adminer, all of which use the following nested variables that can be found in `roles/php/defaults/main.yml`:

```
adminer:
  install: true
  path: "/usr/share/adminer"
  download: "https://github.com/vrana/adminer/releases/download/v4.6.2/adminer-4.6.2-
mysql.php"
```

As you can see, we are using nest variables again, this time to tell our playbook to install the tool, where it should be installed, and also where it can be downloaded from. The first task in `roles/php/tasks/main.yml` is to create the directory where we are going to be installing Adminer:

```
- name: create the document root for adminer
  file:
    dest: "{{ adminer.path }}"
    state: "directory"
    mode: "0755"
  when: adminer.install == true
```

Now that we have somewhere on our Vagrant box to install Adminer, we should download it. This time, as we are not downloading an archive, we are using the `get_url` module:

```
- name: download adminer
  get_url:
    url: "{{ adminer.download }}"
    dest: "{{ adminer.path }}/index.php"
    mode: 0755
  when: adminer.install == true
```

As you can see, we are downloading the `adminer-4.6.2-mysql.php` file from GitHub and saving it to `/usr/share/adminer/index.php`, so how do we access it? The final part of the task uses the `template` module to upload an additional Apache configuration file to `/etc/httpd/conf.d/adminer.conf`:

```
- name: copy the vhost.conf to /etc/httpd/conf.d/
  template:
    src: "adminer.conf.j2"
    dest: "/etc/httpd/conf.d/adminer.conf"
```

```
|   when: adminer.install == true
|   notify: "restart httpd"
```

The `adminer.conf.j2` template, which should be placed in `roles/php/templates`, looks as follows:

```
# {{ ansible_managed }}
Alias /adminer "{{ adminer.path }}"
<Directory "{{ adminer.path }}">
  DirectoryIndex index.php
  AllowOverride All
  Require all granted
</Directory>
```

As you can see, it is creating an alias called `/adminer`, which then points at `index.php` in `/usr/share/adminer/`. As we are adding to the Apache configuration file, we are also notifying the `restart httpd` handler so that Apache restarts, picking up our updated configuration.

Running the role

Now that the role for the final element of our LAMP stack is complete, we can add it to our playbook. It should now look like the following:

```
---
- hosts: boxes
  gather_facts: true
  become: yes
  become_method: sudo

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/common
    - roles/apache
    - roles/mariadb
    - roles/php
```

Run it by using the following command:

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

This will deploy PHP on our Vagrant box; this output picks up as the PHP role is called:

```
TASK [roles/php : install the php packages]
*****
changed: [box] => (item=[u'php72u', u'php72u-bcmath', u'php72u-cli', u'php72u-common',
u'php72u-dba', u'php72u-fpm', u'php72u-fpm-httdp', u'php72u-gd', u'php72u-intl',
u'php72u-json', u'php72u-mbstring', u'php72u-mysqli', u'php72u-odbc', u'php72u-pdo',
u'php72u-process', u'php72u-snmp', u'php72u-soap', u'php72u-xml', u'php72u-xmlrpc'])

TASK [roles/php : copy the test PHP page to the document root]
*****
changed: [box]

TASK [roles/php : create the document root for adminer]
*****
changed: [box]

TASK [roles/php : download adminer]
*****
changed: [box]

TASK [roles/php : copy the vhost.conf to /etc/httpd/conf.d/]
*****
changed: [box]

RUNNING HANDLER [roles/common : restart ntp]
```

```
*****
changed: [box]

RUNNING HANDLER [roles/apache : restart httpd]
*****
changed: [box]

RUNNING HANDLER [roles/php : restart php-fpm]
*****
changed: [box]

PLAY RECAP
*****
box : ok=34 changed=32 unreachable=0 failed=0
```

Once installed, you should be able to access the following URLs:

- <http://192.168.50.4.nip.io/info.php>
- <http://192.168.50.4.nip.io/adminer/>

When you go to the first link, you should see something like the following page:

PHP Version 7.2.2

System	Linux 192.168.50.4.nip.io 3.10.0-693.11.6.el7.x86_64 #1 SMP Thu Jan 4 01:06:37 UTC 2018 x86_64
Build Date	Feb 1 2018 15:31:57
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc
Loaded Configuration File	/etc/php.ini
Scan this dir for additional .ini files	/etc/php.d
Additional .ini files parsed	/etc/php.d/20-bcmath.ini, /etc/php.d/20-bz2.ini, /etc/php.d/20-calendar.ini, /etc/php.d/20-ctype.ini, /etc/php.d/20-curl.ini, /etc/php.d/20-dba.ini, /etc/php.d/20-dom.ini, /etc/php.d/20-exif.ini, /etc/php.d/20 FileInfo.ini, /etc/php.d/20-ftp.ini, /etc/php.d/20-gd.ini, /etc/php.d/20-gettext.ini, /etc/php.d/20-conv.ini, /etc/php.d/20-intl.ini, /etc/php.d/20-json.ini, /etc/php.d/20-mbstring.ini, /etc/php.d/20-mysqli.ini, /etc/php.d/20-odbc.ini, /etc/php.d/20-pdo.ini, /etc/php.d/20-phar.ini, /etc/php.d/20-posix.ini, /etc/php.d/20-shmop.ini, /etc/php.d/20-simplexml.ini, /etc/php.d/20-snmp.ini, /etc/php.d/20-soap.ini, /etc/php.d/20-sockets.ini, /etc/php.d/20-sqlite3.ini, /etc/php.d/20-sysvmsg.ini, /etc/php.d/20-sysvsem.ini, /etc/php.d/20-sysvshm.ini, /etc/php.d/20-tokenizer.ini, /etc/php.d/20-xml.ini, /etc/php.d/20-xmlwriter.ini, /etc/php.d/20-xsl.ini, /etc/php.d/20-zip.ini, /etc/php.d/30-mysqli.ini, /etc/php.d/30-pdo_mysql.ini, /etc/php.d/30-pdo_odbc.ini, /etc/php.d/30-pdo_sqlite.ini, /etc/php.d/30-wdx.ini, /etc/php.d/30-xmireader.ini, /etc/php.d/30-xmlrpc.ini
PHP API	20170718
PHP Extension	20170718
Zend Extension	320170718
Zend Extension Build	API320170718.NTS
PHP Extension Build	API20170718.NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	enabled
Zend Memory Manager	enabled
Zend Multibyte Support	provided by mbstring

At the second link, once logged in using the username `root` and password `Pa55W0rd123`, you should be able to see the `employees` database:

The screenshot shows the Adminer 4.6.2 interface for MySQL. The top navigation bar includes standard OS icons, a refresh button, the IP address 192.168.50.4.nip.io, and a toolbar with various icons. The language is set to English. The main title is "MySQL » Server » Database: employees". A "Logout" button is in the top right.

The left sidebar displays the database name "employees" and a list of recent SQL queries:

- select current_dept_emp
- select departments
- select dept_emp
- select dept_emp_latest_date
- select dept_manager
- select employees
- select salaries
- select titles

The main content area is titled "Database: employees" and shows the "Tables and views" section. It includes a search bar for tables (6) and a table listing the database's schema:

Table	Engine?	Collation?	Data Length?	Index Length?	Data Free?	Auto Increment?	Rows?	Comment?
current_dept_emp	InnoDB	latin1_swedish_ci	16,384	16,384	0			?
departments	InnoDB	latin1_swedish_ci	12,075,008	5,783,552	4,194,304			~ 9
dept_emp	InnoDB	latin1_swedish_ci	15,220,736	0	4,194,304			~ 331,570
dept_emp_latest_date	InnoDB	latin1_swedish_ci	100,270,080	0	4,194,304			?
dept_manager	InnoDB	latin1_swedish_ci	16,384	16,384	0			~ 24
employees	InnoDB	latin1_swedish_ci	20,512,768	0	4,194,304			~ 299,246
salaries	InnoDB	latin1_swedish_ci	148,111,360	5,816,320	0			~ 2,838,426
titles	InnoDB	latin1_swedish_ci						~ 441,951
8 in total								

With Adminer, we have a PHP script accessing our MariaDB database; the pages are being served by Apache from our Linux Vagrant box.

Overriding variables

Before we finish, we should quickly discuss how we can override the default variables we have been setting. To do this, add the following lines to

```
group_vars/common.yml file: html_body: |
```

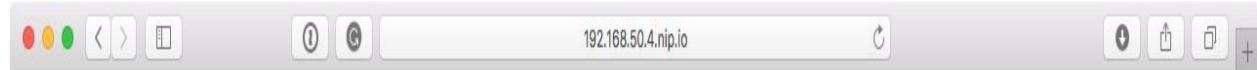
This HTML page has been deployed using Ansible to **<code>{{ ansible_nodename }</code>**.
**
**

The user is **<code>{{ users.0.name }</code>** who is in the **<code>{{ apache_group }</code>**
**
**

The webroot is **<code>{{ document_root }</code>**, the default index file is **<code>{{ index_file }</code>**.
**
**

You can access a [PHP Info file](/info.php) or [Adminer](/adminer).

Then, rerun the playbook. Once the playbook has finished, opening <http://192.168.50.4.nip.io/> will show you the following page:



Success !!!

This HTML page has been deployed using Ansible to
192.168.50.4.nip.io.

The user is **lamp** who is in the **apache** group.

The webroot is **/home/lamp/web**, the default index file is **index.html**.

You can access a [PHP Info file](#) or [Adminer](#).

As you can see, the default `index.html` page has been updated with a link to our `phpinfo` page and also Adminer. Any of the variables we have configured as default can be overridden in this way.

Summary

In this chapter, we have worked through writing a playbook that installs a LAMP stack on our CentOS 7 Vagrant box. We created four roles, one for each element of the stack, and within each of the roles we built in a little logic that can be overridden to deploy additional elements such as test HTML and PHP pages, and we also built in the option to create a test database that contains over 40,000 records.

So far, we have been installing some pretty basic packages. In the next chapter, we will be building a playbook that installs, configures, and maintains a WordPress installation.

Questions

1. Which Ansible module would you use to download and unarchive a zip file?
2. True or false: The variables found in the `roles/rolename/default/` folder override all other references of the same variable.
3. Explain how you would add a second user to our playbook
4. True or false: You can only call a single handler from a task.
5. Update the final playbook to add a second virtual host, which serves a different default HTML page.

Further reading

You can find the project pages for the third-party tools covered throughout the chapter at the following URLs:

- **CentOS:** <https://www.centos.org/>
- **Apache:** <https://httpd.apache.org/>
- **MariaDB:** <https://mariadb.org/>
- **Datacharmer test database:** https://github.com/datacharmer/test_db
- **PHP:** <https://php.net/>
- **PHP-FPM:** <https://php-fpm.org/>
- **Adminer:** <https://www.adminer.org>

Deploying WordPress

In the previous chapter, we worked on building a playbook that installs and configures a basic LAMP stack. In this chapter, we are going to be building on top of the techniques we used there in order to create a playbook that installs a LEMP stack and WordPress.

We will be covering the following topics:

- Preparing our initial playbook
- Downloading and installing the WordPress CLI
- Installing and configuring WordPress
- Logging in to your WordPress installation

Before we start, we should quickly cover what WordPress is. It is likely that at some point in the last 48 hours, you have visited a website that is powered by WordPress. It is an open source **content management system (CMS)** that is powered by PHP and MySQL and is used by around 19,545,516 websites according to the CMS usage statistics provided by BuiltWith.

Technical requirements

Again, a fresh copy of the CentOS 7 Vagrant box we have launched in the previous chapters will be used. This does mean that packages will need to be downloaded again, along with WordPress. You can find a complete copy of the playbook at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter05/lemp>.

Preinstallation tasks

As mentioned in the previous chapter, a LEMP stack is composed of the following elements:

- **Linux:** In our case, this will be CentOS 7 again
- **NGINX:** If you remember, it is pronounced as *engine-x*, and replaces Apache in our stack
- **MariaDB:** As we have seen, this will be the database component
- **PHP:** We will be using PHP 7.2 again for this

Before we install WordPress, we need to install and configure these components. Also, as this playbook is eventually going to be executed against publicly available cloud servers, we need to think about some best practices around our NGINX configuration.

Let's start by getting the initial structure of the playbook set up:

```
$ mkdir lemp lemp/group_vars  
$ touch lemp/group_vars/common.yml lemp/production lemp/site.yml lemp/Vagrantfile  
lemp/.gitignore  
$ cd lemp
```

Now that we have the basic layout, we need to put some content in the `Vagrantfile` and `.gitignore` files. `Vagrantfile` contains the following, similar to the previous chapters:

```
# -*- mode: ruby -*-  
# vi: set ft=ruby :  
  
API_VERSION = "2"  
BOX_NAME      = "centos/7"  
BOX_IP        = "192.168.50.5"  
DOMAIN        = "nip.io"  
PRIVATE_KEY   = "~/.ssh/id_rsa"  
PUBLIC_KEY    = '~/.ssh/id_rsa.pub'  
  
Vagrant.configure(API_VERSION) do |config|  
  config.vm.box = BOX_NAME  
  config.vm.network "private_network", ip: BOX_IP  
  config.vm.host_name = BOX_IP + '.' + DOMAIN  
  config.ssh.insert_key = false  
  config.ssh.private_key_path = [PRIVATE_KEY,  
    "~/.vagrant.d/insecure_private_key"]  
  config.vm.provision "file", source: PUBLIC_KEY, destination:  
    "~/.ssh/authorized_keys"
```

```
config.vm.provider "virtualbox" do |v|
  v.memory = "2024"
  v.cpus = "2"
end

config.vm.provider "vmware_fusion" do |v|
  v.vmx["memsize"] = "2024"
  v.vmx["numvcpus"] = "2"
end

end
```

As you may have spotted, we are using a different IP address for this Vagrant box; the `.gitignore` file should contain a single line:

```
| .vagrant
```

Now that we have the basics configured, we can make a start by writing the playbook to deploy and configure our initial software stack.

The stack-install command

We are going to start by creating a role called `stack-install` using `ansible-galaxy init`:

```
$ ansible-galaxy init roles/stack-install
```

This will install our initial software stack. Once installed, we hand over to a second role, which will then configure the software stack before a third role starts the WordPress installation.

So what packages do we need? WordPress has the following requirements:

- PHP 7.2 or higher
- MariaDB 10.0 or greater, or MySQL 5.6 or greater
- NGINX or Apache with the `mod_rewrite` module
- HTTPS support

We know from the previous chapter that the IUS repository can provide PHP 7.2 and MariaDB 10.1, so we will use that as the source for those packages, but what about NGINX? There are NGINX packages in the EPEL repository. However, we are going to be using the main NGINX repository so we can get the latest and greatest version.

Enabling the repositories

Let's start our playbook by enabling the three repositories we need in order to install our software stack and then, once those repositories are enabled, we should do a `yum update` to make sure that the base operating system is up to date.

The `roles/stack-install/defaults/main.yml` file requires the following content to achieve this. First, we have the locations for the RPM packages that enable EPEL and IUS:

```
repo_packages:
  - "epel-release"
  - "https://centos7.iuscommunity.org/ius-release.rpm"
```

After that, we have the following nested variable, which contains all of the information we need to use the `yum_repository` module in order to create a `.repo` file for the NGINX repository:

```
nginx_repo:
  name: "nginx"
  description: "The mainline NGINX repo"
  baseurl: "http://nginx.org/packages/mainline/centos/7/$basearch/"
  gpgcheck: "no"
  enabled: "yes"
```

Now that we have the defaults in place, we can add the tasks to the `roles/stack-install/tasks/main.yml` file; these are as follows, with the first task being already familiar as all it does is install our two packages:

```
- name: install the repo packages
  yum:
    name: "{{ item }}"
    state: "installed"
  with_items: "{{ repo_packages }}"
```

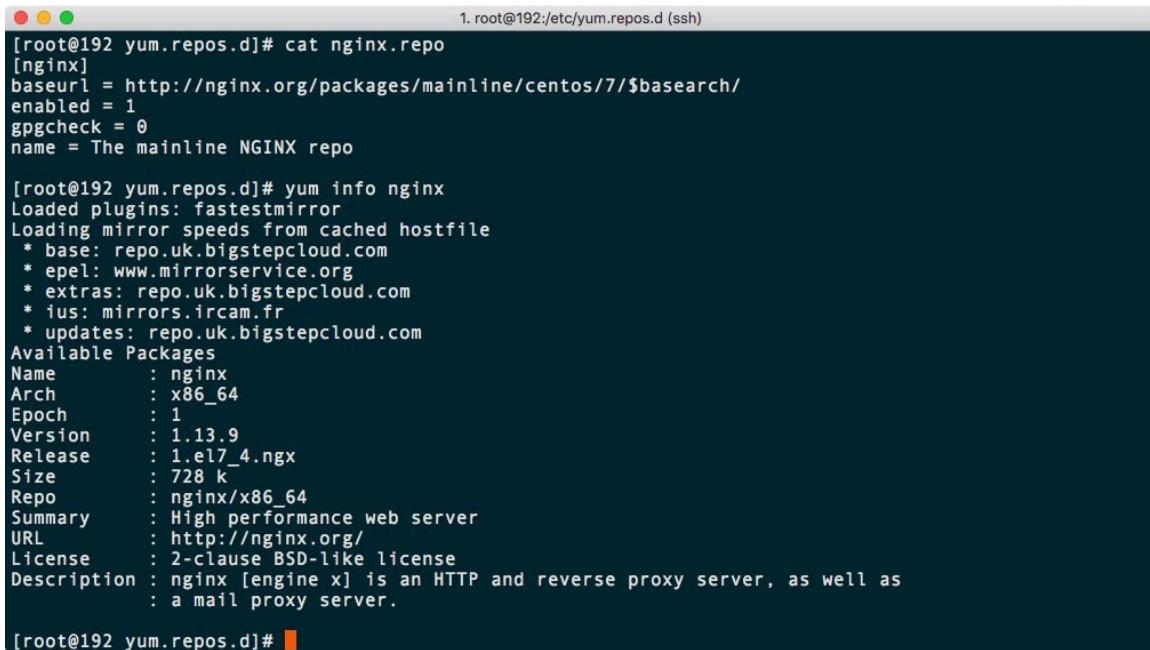
The next task creates a repository file called `nginx.repo` in `/etc/yum.repos.d/`:

```
- name: add the NGINX mainline repo
  yum_repository:
    name: "{{ nginx_repo.name }}"
    description: "{{ nginx_repo.description }}"
    baseurl: "{{ nginx_repo.baseurl }}"
    gpgcheck: "{{ nginx_repo.gpgcheck }}"
    enabled: "{{ nginx_repo.enabled }}"
```

As you can see from the following Terminal output, the content of the file is pointing toward the NGINX repository, and we can get more information on the NGINX package by running:

```
| $ yum info nginx
```

The following screenshot shows the output for the preceding command:



```
1. root@192:/etc/yum.repos.d (ssh)
[root@192 yum.repos.d]# cat nginx.repo
[nginx]
baseurl = http://nginx.org/packages/mainline/centos/7/$basearch/
enabled = 1
gpgcheck = 0
name = The mainline NGINX repo

[root@192 yum.repos.d]# yum info nginx
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
 * base: repo.uk.bigstepcloud.com
 * epel: www.mirrorservice.org
 * extras: repo.uk.bigstepcloud.com
 * ius: mirrors.ircam.fr
 * updates: repo.uk.bigstepcloud.com
Available Packages
Name        : nginx
Arch       : x86_64
Epoch      : 1
Version    : 1.13.9
Release    : 1.el7_4.ngx
Size       : 728 k
Repo       : nginx/x86_64
Summary    : High performance web server
URL        : http://nginx.org/
License    : 2-clause BSD-like license
Description : nginx [engine x] is an HTTP and reverse proxy server, as well as
              : a mail proxy server.

[root@192 yum.repos.d]#
```

The following task should also look familiar, as we used it in the previous chapter to update the installed packages:

```
- name: update all of the installed packages
  yum:
    name: "*"
    state: "latest"
    update_cache: "yes"
```

Now that we have our source repositories set up and the already installed packages updated, we can proceed with the remainder of the package installations.

Installing the packages

We are going to create four lists of packages; these are in the `roles/stack-install/defaults/main.yml` file. As per the previous chapter, we first need to uninstall a preinstalled MariaDB package, so our first list includes packages to remove:

```
packages_remove:  
  - "mariadb-libs.x86_64"
```

Next up, we have the packages needed to allow Ansible to interact with services such as SELinux and MariaDB, as well as installing the Postfix package, which, we know from the last time, is removed:

```
system_packages:  
  - "postfix"  
  - "MySQL-python"  
  - "policycoreutils-python"
```

Then, we have all of the packages that make up our core software stack:

```
stack_packages:  
  - "nginx"  
  - "mariadb101u"  
  - "mariadb101u-server"  
  - "mariadb101u-config"  
  - "mariadb101u-common"  
  - "mariadb101u-libs"  
  - "php72u"  
  - "php72u-bcmath"  
  - "php72u-cli"  
  - "php72u-common"  
  - "php72u-dba"  
  - "php72u-fpm"  
  - "php72u-fpm-nginx"  
  - "php72u-gd"  
  - "php72u-intl"  
  - "php72u-json"  
  - "php72u-mbstring"  
  - "php72u-mysqlnd"  
  - "php72u-process"  
  - "php72u-snmp"  
  - "php72u-soap"  
  - "php72u-xml"  
  - "php72u-xmlrpc"
```

Finally, we have a few nice-to-haves:

```
extra_packages:  
  - "vim-enhanced"  
  - "git"
```

```
| - "unzip"
```

The tasks to remove the packages and then install them should be placed in the `roles/stack-install/tasks/main.yml` file, starting with the task to remove the packages:

```
- name: remove the packages so that they can be replaced
  yum:
    name: "{{ item }}"
    state: "absent"
  with_items: "{{ packages_remove }}"
```

Then, we can install all of the packages in one go using the following task:

```
- name: install the stack packages
  yum:
    name: "{{ item }}"
    state: "installed"
  with_items: "{{ system_packages + stack_packages + extra_packages }}"
```

Notice how we are combining the three remaining lists of packages in a single variable. We are doing this so we do not have to repeat the `yum` task any more than we have to. It also allows us to override, say, just `extra_packages` elsewhere in the playbook and not have to repeat the entire list of required packages needed for other parts of the stack.

The stack-config role

The next role will configure the software stack we have just installed, so let's create the role:

```
| $ ansible-galaxy init roles/stack-config
```

Now that we have the files needed for the role, we can make a start on planning what we need to configure. We will need to do the following:

- Create a user for our WordPress to run under
- Configure NGINX as per the best practices on the WordPress Codex
- Configure PHP-FPM to run as the WordPress user
- Do the initial configuration for SELinux

Let's start by creating the WordPress user.

WordPress system user

The defaults for the WordPress system user, which should be placed in `roles/stack-config/defaults/main.yml`, are as follows: `wordpress_system:`

```
user: "wordpress"
group: "php-fpm"
comment: "wordpress system user"
home: "/var/www/wordpress"
state: "present"
```

We are referring to this as the system user as we will be creating a user in WordPress later in the chapter. This user's details will also be defined in Ansible, so we do not want to get the two different users confused.

The task that uses these variables, found in `roles/stack-config/tasks/main.yml`, should look like this: - name: add the wordpress user

```
user:
name: "{{ wordpress_system.user }}"
group: "{{ wordpress_system.group }}"
comment: "{{ wordpress_system.comment }}"
home: "{{ wordpress_system.home }}"
state: "{{ wordpress_system.state }}"
```

As you can see, we are not adding a key to the user this time as we don't want to be logging in to the user account to start manipulating files and other actions. This should all be done within WordPress itself or by using Ansible.

NGINX configuration

We are going to be using several template files for our NGINX configuration. The first template is called `roles/stack-config/templates/nginx-nginx.conf.j2`, and it will replace the main NGINX configuration deployed by the package installation:

```
# {{ ansible_managed }}
user nginx;
worker_processes {{ ansible_processor_count }};
error_log /var/log/nginx/error.log warn;
pid /var/run/nginx.pid;

events {
    worker_connections 1024;
}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;
    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
                    '$status $body_bytes_sent "$http_referer" '
                    '"$http_user_agent" "$http_x_forwarded_for"';
    access_log /var/log/nginx/access.log main;
    sendfile on;
    keepalive_timeout 65;
    client_max_body_size 20m;
    include /etc/nginx/conf.d/*.conf;
}
```

The content of the file itself is pretty much the same as the initial file, except that we are updating `worker_processes` so that it uses the number of processors detected by Ansible when the `setup` module runs, rather than a hardcoded value.

The task to deploy the configuration file is as you would expect and it should be placed in `roles/stack-config/tasks/main.yml`:

```
- name: copy the nginx.conf to /etc/nginx/
  template:
    src: "nginx-nginx.conf.j2"
    dest: "/etc/nginx/nginx.conf"
  notify: "restart nginx"
```

As you can see, we are notifying the `restart nginx` handler, which is stored in the following `roles/stack-config/handlers/main.yml` file:

```
- name: "restart nginx"
  service:
```

```
    name: "nginx"
    state: "restarted"
    enabled: "yes"
```

Next up, we have the default site template, `roles/stack-config/templates/nginx-confd-default.conf.j2`:

```
# {{ ansible_managed }}

upstream {{ php.upstream }} {
    server {{ php.ip }}:{{ php.port }};
}

server {
    listen 80;
    server_name {{ ansible_nodename }};
    root {{ wordpress_system.home }};
    index index.php index.html index.htm;

    include global/restrictions.conf;
    include global/wordpress_shared.conf;
}
```

To help identify where the template files will be placed on the target host, I am naming them so that the full path is in the filename. In this case, the filename is `nginx-confd-default.conf.j2` and it will be deployed to `/etc/nginx/conf.d/...`

The next two templates we are deploying are going into a folder that doesn't exist. So, we first need to create the destination folder. To do this, we need to add the following to `roles/stack-config/tasks/main.yml`:

```
- name: create the global directory in /etc/nginx/
  file:
    dest: "/etc/nginx/global/"
    state: "directory"
    mode: "0644"
```

Then, the following command will copy the files to the `global` folder:

```
- name: copy the restrictions.conf to /etc/nginx/global/
  copy:
    src: "nginx-global-restrictions.conf"
    dest: "/etc/nginx/global/restrictions.conf"
    notify: "restart nginx"

- name: copy the wordpress_shared.conf to /etc/nginx/global/
  template:
    src: "nginx-global-wordpress_shared.conf.j2"
    dest: "/etc/nginx/global/wordpress_shared.conf"
    notify: "restart nginx"
```

As we are not making any replacements in the `nginx-global-restrictions.conf` file,

we are using the `copy` module rather than `template` here; the file is stored in `roles/stack-config/files/` and has the following content:

```
# Do not log robots.txt
location = /robots.txt {
    log_not_found off;
    access_log off;
}

# If no favicon exists return a 204 (no content error)
location ~* /favicon\.(ico$ {
    try_files $uri =204;
    expires max;
    log_not_found off;
    access_log off;
}

# Deny access to htaccess files
location ~ /\. {
    deny all;
}

# Deny access to some bits wordpress leaves hanging around
location ~* /(wp-config.php|readme.html|license.txt|nginx.conf) {
    deny all;
}

# Deny access to .php files in the /wp-content/ directory (including sub-folders)
location ~* ^/wp-content/.*(php|phps)$ {
    deny all;
}

# Allow only internal access to .php files inside wp-includes directory
location ~* ^/wp-includes/.*\.(php|phps)$ {
    internal;
}

# Deny access to specific files in the /wp-content/ directory (including sub-folders)
location ~* ^/wp-content/.*(txt|md|exe)$ {
    deny all;
}

# hide content of sensitive files
location ~* \\.
(conf|engine|inc|info|install|make|module|profile|test|po|sh|.*sql|theme|tpl(\\.php)?
|xtmpl)\$|^(\\\..*|Entries.*|Repository|Root|Tag|Template)\$|\\.php_ {
    deny all;
}

# don't allow other executable file types
location ~* \\.(pl|cgi|py|sh|lua)\$ {
    deny all;
}

# hide the wordfence firewall
location ~ ^/\.user\.ini {
    deny all;
}
```

As we are setting `php.upstream` as a variable, we are using the `template` module to

make sure that our configuration contains the correct value, the file `roles/stack-config/templates/nginx-global-wordpress_shared.conf.j2` contains the following:

```
# http://wiki.nginx.org/WordPress
# This is cool because no php is touched for static content.
# Include the "?$args" part so non-default permalinks doesn't break when using
query string
location / {
    try_files $uri $uri/ /index.php?$args;
}

# Set the X-Frame-Options
add_header X-Frame-Options "SAMEORIGIN" always;
add_header X-Xss-Protection "1; mode=block" always;
add_header X-Content-Type-Options "nosniff" always;

# Do not log + cache images, css, js, etc
location ~* \.(ico|css|js|gif|jpeg|jpg|png|woff|ttf|otf|svg|woff2|eot)$ {
    expires max;
    log_not_found off;
    access_log off;
    # Send the all shebang in one fell swoop
    tcp_nodelay off;
    # Set the OS file cache
    open_file_cache max=1000 inactive=120s;
    open_file_cache_valid 45s;
    open_file_cache_min_uses 2;
    open_file_cache_errors off;
}

# Handle .php files
location ~ \.php$ {
    try_files $uri =404;
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    include /etc/nginx/fastcgi_params;
    fastcgi_connect_timeout 180s;
    fastcgi_send_timeout 180s;
    fastcgi_read_timeout 180s;
    fastcgi_intercept_errors on;
    fastcgi_max_temp_file_size 0;
    fastcgi_pass {{ php.upstream }};
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    fastcgi_index index.php;
}

# Rewrite rules for WordPress SEO by Yoast
rewrite ^/sitemap_index\.xml$ /index.php?sitemap=1 last;
rewrite ^/([^\/]+?)-sitemap([0-9]+)\?.*\.\w+$ /index.php?sitemap=$1&sitemap_n=$2
last;

# Add trailing slash to */wp-admin requests
rewrite /wp-admin\$ $scheme://\$host\$uri/ permanent;
```

The final part of the NGINX configuration is to copy the main configuration for the WordPress site. The task in `roles/stack-config/tasks/main.yml` looks as follows:

```
- name: copy the default.conf to /etc/nginx/conf.d/
  template:
    src: "nginx-confd-default.conf.j2"
    dest: "/etc/nginx/conf.d/default.conf"
```

```
|   notify: "restart nginx"
```

As we are setting a few variables, such as the path and domain name, we have the following template file:

```
# {{ ansible_managed }}
```

```
upstream php {
    server {{ php.ip }}:{{ php.port }};
}

server {
    listen 80;
    server_name {{ ansible_nodename }};
    root {{ wordpress_system.home }};
    index index.php;
    include global/restrictions.conf;
    include global/wordpress_shared.conf;
}
```

As you can see, we are using a few variables we haven't defined yet, `php.ip` and `php.port`. We are going to look at configuring PHP-FPM next.

PHP and PHP-FPM configuration

As we saw in the last section, there are a few variables defined for PHP in `roles/stack-config/defaults/main.yml`, and these are:

```
php:
  ip: "127.0.0.1"
  port: "9000"
  upstream: "php"
  ini:
    - { regexp: '^date.timezone =', replace: 'date.timezone = Europe/London' }
    - { regexp: '^expose_php = On', replace: 'expose_php = Off' }
    - { regexp: '^upload_max_filesize = 2M', replace: 'upload_max_filesize = 20M' }
```

The first configuration task is to deploy the PHP-FPM configuration; this is what the template looks like:

```
; {{ ansible_managed }}

[{{ wordpress_system.user }}]
user = {{ wordpress_system.user }}
group = {{ wordpress_system.group }}
listen = {{ php.ip }}:{{ php.port }}
listen.allowed_clients = {{ php.ip }}
pm = dynamic
pm.max_children = 50
pm.start_servers = 5
pm.min_spare_servers = 5
pm.max_spare_servers = 35
php_admin_value[error_log] = /var/log/php-fpm/{{ wordpress_system.user }}-error.log
php_admin_flag[log_errors] = on
php_value[session.save_handler] = files
php_value[session.save_path] = /var/lib/php/fpm/session
php_value[soap.wsdl_cache_dir] = /var/lib/php/fpm/wsdlcache
```

As you can see, we have a few replacements in this file. Starting at the top between the square brackets, we are defining the PHP-FPM pool name; we are using the content of the `wordpress_system.user` for this. Next up, we have the user and group we want our pool to run under; here, we are using `wordpress_system.user` and `wordpress_system.group`. Finally, we are setting the IP address and port we want our PHP-FPM pool to listen on by using the `php.ip` and `php.port` variables.

The task in `roles/stack-config/tasks/main.yml` to deploy the template looks as follows:

```
- name: copy the www.conf to /etc/php-fpm.d/
  template:
```

```
    |     src: "php-fpmd-www.conf.j2"
    |     dest: "/etc/php-fpm.d/www.conf"
    |     notify: "restart php-fpm"
```

The handler to restart PHP-FPM in `roles/stack-config/handlers/main.yml` is just:

```
  - name: "restart php-fpm"
    service:
      name: "php-fpm"
      state: "restarted"
      enabled: "yes"
```

The next task in `roles/stack-config/tasks/main.yml` uses the `lineinfile` module:

```
  - name: configure php.ini
    lineinfile:
      dest: "/etc/php.ini"
      regexp: "{{ item.regexp }}"
      line: "{{ item.replace }}"
      backup: "yes"
      backrefs: "yes"
      with_items: "{{ php.ini }}"
      notify: "restart php-fpm"
```

What we are doing here is taking the content of `php.ini` and looping through it by looking for the value defined by the `regexp` key. Once we find the value, we are replacing it with the content of the `replace` key. If there are changes to the file, we are making a `backup` first, just in case. Also, we are using `backrefs` to ensure that if there is no matching regex in the file, then it will be left unchanged; if we didn't use them, the `restart php-fpm` handler would be called every time the playbook runs, and we do not want PHP-FPM to be restarted if there is no reason to.

Starting NGINX and PHP-FPM

Now that we have our stack installed and configured, we need to start the two services rather than waiting until the end of the playbook run. If we don't do this now, our upcoming role to install WordPress will fail. The two tasks in `roles/stack-config/tasks/main.yml` are:

```
- name: start php-fpm
  service:
    name: "php-fpm"
    state: "started"

- name: start nginx
  service:
    name: "nginx"
    state: "started"
```

MariaDB Configuration

The MariaDB configuration is going to closely match that of the last chapter, minus a few of the steps, so I am not going to go into too much detail.

The default variables for this part of the role in `roles/stack-config/defaults/main.yml` are:

```
mariadb:  
  bind: "127.0.0.1"  
  server_config: "/etc/my.cnf.d/mariadb-server.cnf"  
  username: "root"  
  password: "Pa55W0rd123"  
  hosts:  
    - "127.0.0.1"  
    - ">::1"  
    - "{{ ansible_nodename }}"  
    - "localhost"
```

As you can see, we are now using a nested variable, and we have removed root access on the host wildcard, the %, as the first part of the task in `roles/stack-config/tasks/main.yml` binds MariaDB to the localhost:

```
- name: configure the mariadb bind address  
  lineinfile:  
    dest: "{{ mariadb.server_config }}"  
    regexp: "#bind-address=0.0.0.0"  
    line: "bind-address={{ mariadb.bind }}"  
    backup: "yes"  
    backrefs: "yes"
```

From there, we then start MariaDB, set the root password, configure the `~/.my.cnf` file, and then remove the anonymous user and test database:

```
- name: start mariadb  
  service:  
    name: "mariadb"  
    state: "started"  
    enabled: "yes"  
  
- name: change mysql root password  
  mysql_user:  
    name: "{{ mariadb.username }}"  
    host: "{{ item }}"  
    password: "{{ mariadb.password }}"  
    check_implicit_admin: "yes"  
    priv: "*.*:ALL,GRANT"  
    with_items: "{{ mariadb.hosts }}"  
  
- name: set up .my.cnf file
```

```
template:
  src: "my.cnf.j2"
  dest: "~/.my.cnf"

- name: delete anonymous MySQL user
  mysql_user:
    user: ""
    host: "{{ item }}"
    state: "absent"
  with_items: "{{ mariadb.hosts }}"

- name: remove the MySQL test database
  mysql_db:
    db: "test"
    state: "absent"
```

The template used for the `.my.cnf` file, which can be found in `roles/stack-config/templates/my.cnf.j2`, now looks as follows:

```
# {{ ansible_managed }}
[client]
password='{{ mariadb.password }}'
```

This means that we will not need to pass the root username and password with each database-related task from where we copied the `.my.cnf` file.

SELinux configuration

The final task of the role is to set HTTP in SELinux to be permissive; to do this, we have the following variable in `roles/stack-config/defaults/main.yml`:

```
| selinux:  
|   http_permissive: true
```

The task in `roles/stack-config/tasks/main.yml` has a condition that runs if `selinux.http_permissive` equals `true`:

```
- name: set the selinux allowing httpd_t to be permissive is required  
  selinux_permissive:  
    name: httpd_t  
    permissive: true  
  when: selinux.http_permissive == true
```

We will be looking more at SELinux in a later chapter; for now, we are just allowing all HTTP requests.

WordPress installation tasks

Now that we have the roles that prepare our target Vagrant box completed, we can proceed with the actual WordPress installation; this will be split into a few different parts, starting with downloading `wp_c1i` and setting up the database.

Before we progress, we should create the role:

```
| $ ansible-galaxy init roles/wordpress
```

WordPress CLI installation

WordPress CLI (WP-CLI) is a command-line tool used to administer your WordPress installation; we will be using it throughout the role, so the first thing our role should do is download it. To do this, we need to download the following variables in `roles/wordpress/defaults/main.yml`:

```
wp_cli:
  download: "https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar"
  path: "/usr/local/bin/wp"
```

As you probably have gathered from the two variables, we are going to be downloading the file from `wp_cli.download` and copying it to `wp_cli.path`. The task in `roles/wordpress/tasks/main.yml` to do this looks like the following:

```
- name: download wp-cli
  get_url:
    url: "{{ wp_cli.download }}"
    dest: "{{ wp_cli.path }}"
- name: update permissions of wp-cli to allow anyone to execute it
  file:
    path: "{{ wp_cli.path }}"
    mode: "0755"
```

As you can see, we are downloading the `.phar` file, moving it, and then setting permissions so that it can be executed by anyone logged in to the server—this is important as we will be running a lot of the installation commands as the `wordpress` user.

Creating the WordPress database

The next part of the role creates the database our WordPress installation will use; as per the other tasks in this chapter, it uses a nested variable, which can be found in `roles/wordpress/defaults/main.yml`:

```
wp_database:  
  name: "wordpress"  
  username: "wordpress"  
  password: "W04DPr3S5"
```

The tasks in `roles/wordpress/tasks/main.yml` to create the database and user look like the following:

```
- name: create the wordpress database  
  mysql_db:  
    db: "{{ wp_database.name }}"  
    state: "present"  
  
- name: create the user for the wordpress database  
  mysql_user:  
    name: "{{ wp_database.username }}"  
    password: "{{ wp_database.password }}"  
    priv: "{{ wp_database.name }}.*:ALL"  
    state: "present"  
  with_items: "{{ mariadb.hosts }}"
```

Notice how we are using the `mariadb.hosts` variable from the previous role. Now that we have the database created, we can make a start on downloading and installing WordPress.

Downloading, configuring, and installing WordPress

Now that we have everything in place to install WordPress, we can make a start, first by setting some default variables in `roles/wordpress/defaults/main.yml`:

```
wordpress:
  domain: "http://{{ ansible_nodename }}/"
  title: "WordPress installed by Ansible"
  username: "ansible"
  password: "password"
  email: "test@example.com"
  theme: "sydney"
  plugins:
    - "jetpack"
    - "wp-super-cache"
    - "wordpress-seo"
    - "wordfence"
    - "nginx-helper"
```

Now that we have our variables, we can start our download if we need to:

```
- name: are the wordpress files already there?
  stat:
    path: "{{ wordpress_system.home }}/index.php"
  register: wp_installed

- name: download wordpresss
  shell: "{{ wp_cli.path }} core download"
  args:
    chdir: "{{ wordpress_system.home }}"
    become_user: "{{ wordpress_system.user }}"
    become: true
  when: wp_installed.stat.exists == False
```

As you can see, the first task uses the `stat` module to check for the presence of an `index.php` in our system user's home directory, which is also the webroot. The second task uses the `shell` module to issue the `wp core download` command.

There are a few arguments we should work through before moving on to the next task. These are:

- `args` and `chdir`: You can pass additional arguments to the `shell` module using `args`. Here, we are passing `chdir`, which instructs Ansible to change to the directory we specify before running the `shell` command we provide.

- `become_user`: The user we want to run the command as. If we do not use this, the command will run as the root user.
- `become`: This instructs Ansible to execute the task as the defined user.

The next task in the playbook sets the correct permissions on the user's home directory:

```
- name: set the correct permissions on the homedir
  file:
    path: "{{ wordpress_system.home }}"
    mode: "0755"
  when: wp_installed.stat.exists == False
```

Now that WordPress is downloaded, we can start the installation. First, we need to check whether this has already been done:

```
- name: is wordpress already configured?
  stat:
    path: "{{ wordpress_system.home }}/wp-config.php"
  register: wp_configured
```

If there is no `wp-config.php` file, then the following task will be executed:

```
- name: configure wordpress
  shell: "{{ wp_cli.path }} core config --dbhost={{ mariadb.bind }} --dbname={{ wp_database.name }} --dbuser={{ wp_database.username }} -dbpass={{ wp_database.password }}"
  args:
    chdir: "{{ wordpress_system.home }}"
  become_user: "{{ wordpress_system.user }}"
  become: true
  when: wp_configured.stat.exists == False
```

Now that we have our `wp-config.php` file created, with the database credentials in place, we can install WordPress. First, we need to check whether WordPress has already been installed:

```
- name: do we need to install wordpress?
  shell: "{{ wp_cli.path }} core is-installed"
  args:
    chdir: "{{ wordpress_system.home }}"
  become_user: "{{ wordpress_system.user }}"
  become: true
  ignore_errors: yes
  register: wp_installed
```

As you can see from the presence of the `ignore_errors` option, if WordPress is not installed, this command will give us an error. We are then using this to our advantage when registering the results, as you can see from the following task:

```
- name: install wordpress if needed
  shell: "{{ wp_cli.path }} core install --url='{{ wordpress.domain }}' --title='{{ wordpress.title }}' --admin_user={{ wordpress.username }} --admin_password={{ wordpress.password }} --admin_email={{ wordpress.email }}"
  args:
    chdir: "{{ wordpress_system.home }}"
  become_user: "{{ wordpress_system.user }}"
  become: true
  when: wp_installed.rc == 1
```

Now we have a basic WordPress site installed, we can progress with installing the plugins and theme files.

WordPress plugins and theme installation

The last part of our WordPress installation is downloading and installing the plugins and theme files we defined in the `wordpress.plugins` and `wordpress.theme` variables.

Let's start with the tasks that install the plugins, so we don't end up rerunning the task that installs the plugins. When we need to, we will be building a little logic into the tasks. First of all, we run a task to see if all of the plugins are already installed:

```
- name: do we need to install the plugins?
  shell: "{{ wp_cli.path }} plugin is-installed {{ item }}"
  args:
    chdir: "{{ wordpress_system.home }}"
    become_user: "{{ wordpress_system.user }}"
    become: true
    with_items: "{{ wordpress.plugins }}"
    ignore_errors: yes
    register: wp_plugin_installed
```

If the plugins are not installed, then this task should fail, which is why we have the `ignore_errors` in there. As you can see, we are registering the results of the entire task, because if you remember we are installing several plugins, as `wp_plugin_installed`. The next two tasks take the results of `wp_plugin_installed` and use the `setfact` module to set a fact:

```
- name: set a fact if we don't need to install the plugins
  set_fact:
    wp_plugin_installed_skip: true
  when: wp_plugin_installed.failed is undefined

- name: set a fact if we need to install the plugins
  set_fact:
    wp_plugin_installed_skip: false
  when: wp_plugin_installed.failed is defined
```

As you can see, we are setting `wp_theme_installed_skip` to be `true` or `false`: if the fact is set to `false`, then the next task will loop through installing the plugins:

```
- name: install the plugins if we need to or ignore if not
  shell: "{{ wp_cli.path }} plugin install {{ item }} --activate"
  args:
```

```
    chdir: "{{ wordpress_system.home }}"
become_user: "{{ wordpress_system.user }}"
become: true
with_items: "{{ wordpress.plugins }}"
when: wp_plugin_installed_skip == false
```

If we add another plugin to the list but leave the others in place, it will show an error, causing the plugin to be installed. We are going to be using this same logic to figure out whether the theme file we have defined as `wordpress.theme` needs to be installed:

```
- name: do we need to install the theme?
shell: "{{ wp_cli.path }} theme is-installed {{ wordpress.theme }}"
args:
  chdir: "{{ wordpress_system.home }}"
become_user: "{{ wordpress_system.user }}"
become: true
ignore_errors: yes
register: wp_theme_installed

- name: set a fact if we don't need to install the theme
set_fact:
  wp_theme_installed_skip: true
when: wp_theme_installed.failed == false

- name: set a fact if we need to install the theme
set_fact:
  wp_theme_installed_skip: false
when: wp_theme_installed.failed == true

- name: install the theme if we need to or ignore if not
shell: "{{ wp_cli.path }} theme install {{ wordpress.theme }} --activate"
args:
  chdir: "{{ wordpress_system.home }}"
become_user: "{{ wordpress_system.user }}"
become: true
when: wp_theme_installed_skip == false
```

Now that we have the plugins and theme installed, we can have a go at running our playbook.

Running the WordPress playbook

To run the playbook and install WordPress, we need a few things, starting with the inventory file called `production`: `box1 ansible_host=192.168.50.5.nip.io`

```
[wordpress]
box1
```

```
[wordpress:vars]
ansible_connection=ssh
ansible_user=vagrant
ansible_private_key_file=~/ssh/id_rsa
host_key_checking=False
```

As you can see, it takes into account the updated IP address of the Vagrant box we defined at the start of the chapter. Also, we need the playbook itself; `site.yml` should look as follows: ---

```
- hosts: wordpress
gather_facts: true
become: yes
become_method: sudo

vars_files:
- group_vars/common.yml
```

```
roles:
- roles/stack-install
- roles/stack-config
- roles/wordpress
```

Now, start the Vagrant box by running one of the following two commands:

```
$ vagrant up
$ vagrant up --provider=vmware_fusion
```

Once your Vagrant box is up and running, we can start the playbook run with this command:

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

When the playbook is first executed, you should see something like the following results:

```
PLAY [wordpress]
*****
TASK [Gathering Facts]
*****
ok: [box1]

TASK [roles/stack-install : install the repo packages]
*****
changed: [box1] => (item=[u'epel-release', u'https://centos7.iuscommunity.org/ius-release.rpm'])

TASK [roles/stack-install : add the NGINX mainline repo]
*****
changed: [box1]

TASK [roles/stack-install : update all of the installed packages]
*****
changed: [box1]

TASK [roles/stack-install : remove the packages so that they can be replaced]
*****
changed: [box1] => (item=[u'mariadb-libs.x86_64'])

TASK [roles/stack-install : install the stack packages]
*****
changed: [box1] => (item=[u'postfix', u'MySQL-python', u'policycoreutils-python', u'nginx', u'mariadb101u', u'mariadb101u-server', u'mariadb101u-config', u'mariadb101u-common', u'mariadb101u-libs', u'php72u', u'php72u-bcmath', u'php72u-cli', u'php72u-common', u'php72u-dba', u'php72u-fpm', u'php72u-fpm-nginx', u'php72u-gd', u'php72u-intl', u'php72u-json', u'php72u-mbstring', u'php72u-mysqlnd', u'php72u-process', u'php72u-snmp', u'php72u-soap', u'php72u-xml', u'php72u-xmlrpc', u'vim-enhanced', u'git', u'unzip'])

TASK [roles/stack-config : add the wordpress user]
*****
changed: [box1]

TASK [roles/stack-config : copy the nginx.conf to /etc/nginx/]
*****
changed: [box1]

TASK [roles/stack-config : create the global directory in /etc/nginx/]
*****
changed: [box1]

TASK [roles/stack-config : copy the restrictions.conf to /etc/nginx/global/]
*****
changed: [box1]

TASK [roles/stack-config : copy the wordpress_shared.conf to /etc/nginx/global/]
*****
```

```
changed: [box1]

TASK [roles/stack-config : copy the default.conf to /etc/nginx/conf.d/]
*****
changed: [box1]

TASK [roles/stack-config : copy the www.conf to /etc/php-fpm.d/]
*****
changed: [box1]

TASK [roles/stack-config : configure php.ini]
*****
changed: [box1] => (item={u'regexp': u'^date.timezone =', u'replace': u'date.timezone = Europe/London'})
changed: [box1] => (item={u'regexp': u'^expose_php = On', u'replace': u'expose_php = Off'})
changed: [box1] => (item={u'regexp': u'^upload_max_filesize = 2M', u'replace': u'upload_max_filesize = 20M'})

TASK [roles/stack-config : start php-fpm]
*****
changed: [box1]

TASK [roles/stack-config : start nginx]
*****
changed: [box1]

TASK [roles/stack-config : configure the mariadb bind address]
*****
changed: [box1]

TASK [roles/stack-config : start mariadb]
*****
changed: [box1]

TASK [roles/stack-config : change mysql root password]
*****
changed: [box1] => (item=127.0.0.1)
changed: [box1] => (item=:1)
changed: [box1] => (item=192.168.50.5.nip.io)
changed: [box1] => (item=localhost)

TASK [roles/stack-config : set up .my.cnf file]
*****
changed: [box1]

TASK [roles/stack-config : delete anonymous MySQL user]
*****
ok: [box1] => (item=127.0.0.1)
ok: [box1] => (item=:1)
changed: [box1] => (item=192.168.50.5.nip.io)
changed: [box1] => (item=localhost)

TASK [roles/stack-config : remove the MySQL test database]
*****
changed: [box1]

TASK [roles/stack-config : set the selinux allowing httpd_t to be permissive is required]
*****
changed: [box1]

TASK [roles/wordpress : download wp-cli]
*****
changed: [box1]
```

```
TASK [roles/wordpress : update permissions of wp-cli to allow anyone to execute it]
*****
changed: [box1]

TASK [roles/wordpress : create the wordpress database]
*****
changed: [box1]

TASK [roles/wordpress : create the user for the wordpress database]
*****
changed: [box1] => (item=127.0.0.1)
ok: [box1] => (item=:1)
ok: [box1] => (item=192.168.50.5.nip.io)
ok: [box1] => (item=localhost)

TASK [roles/wordpress : are the wordpress files already there?]
*****
ok: [box1]

TASK [roles/wordpress : download wordpresss]
*****
changed: [box1]

TASK [roles/wordpress : set the correct permissions on the homedir]
*****
changed: [box1]

TASK [roles/wordpress : is wordpress already configured?]
*****
ok: [box1]

TASK [roles/wordpress : configure wordpress]
*****
changed: [box1]

TASK [roles/wordpress : do we need to install wordpress?]
*****
fatal: [box1]: FAILED! => {"changed": true, "cmd": "/usr/local/bin/wp core is-installed", "delta": "0:00:00.364987", "end": "2018-03-04 20:22:16.659411", "msg": "non-zero return code", "rc": 1, "start": "2018-03-04 20:22:16.294424", "stderr": "", "stderr_lines": [], "stdout": "", "stdout_lines": []}
...ignoring

TASK [roles/wordpress : install wordpress if needed]
*****
changed: [box1]

TASK [roles/wordpress : do we need to install the plugins?]
*****
failed: [box1] (item=jetpack) => {"changed": true, "cmd": "/usr/local/bin/wp plugin is-installed jetpack", "delta": "0:00:01.366121", "end": "2018-03-04 20:22:20.175418", "item": "jetpack", "msg": "non-zero return code", "rc": 1, "start": "2018-03-04 20:22:18.809297", "stderr": "", "stderr_lines": [], "stdout": "", "stdout_lines": []}
failed: [box1] (item=wp-super-cache) => {"changed": true, "cmd": "/usr/local/bin/wp plugin is-installed wp-super-cache", "delta": "0:00:00.380384", "end": "2018-03-04 20:22:21.035274", "item": "wp-super-cache", "msg": "non-zero return code", "rc": 1, "start": "2018-03-04 20:22:20.654890", "stderr": "", "stderr_lines": [], "stdout": "", "stdout_lines": []}
failed: [box1] (item=wordpress-seo) => {"changed": true, "cmd": "/usr/local/bin/wp plugin is-installed wordpress-seo", "delta": "0:00:00.354021", "end": "2018-03-04 20:22:21.852955", "item": "wordpress-seo", "msg": "non-zero return code", "rc": 1, "start": "2018-03-04 20:22:21.498934", "stderr": "", "stderr_lines": [], "stdout": "", "stdout_lines": []}
```

```

failed: [box1] (item=wordfence) => {"changed": true, "cmd": "/usr/local/bin/wp plugin
is-installed wordfence", "delta": "0:00:00.357012", "end": "2018-03-04
20:22:22.673549", "item": "wordfence", "msg": "non-zero return code", "rc": 1, "start":
"2018-03-04 20:22:22.316537", "stderr": "", "stderr_lines": [], "stdout": "",
"stdout_lines": []}
failed: [box1] (item=nginx-helper) => {"changed": true, "cmd": "/usr/local/bin/wp
plugin is-installed nginx-helper", "delta": "0:00:00.346194", "end": "2018-03-04
20:22:23.389176", "item": "nginx-helper", "msg": "non-zero return code", "rc": 1,
"start": "2018-03-04 20:22:23.042982", "stderr": "", "stderr_lines": [], "stdout": "",
"stdout_lines": []}
...ignoring

TASK [roles/wordpress : set a fact if we don't need to install the plugins]
*****
skipping: [box1]

TASK [roles/wordpress : set a fact if we need to install the plugins]
*****
ok: [box1]

TASK [roles/wordpress : install the plugins if we need to or ignore if not]
*****
changed: [box1] => (item=jetpack)
changed: [box1] => (item=wp-super-cache)
changed: [box1] => (item=wordpress-seo)
changed: [box1] => (item=wordfence)
changed: [box1] => (item=nginx-helper)

TASK [roles/wordpress : do we need to install the theme?]
*****
fatal: [box1]: FAILED! => {"changed": true, "cmd": "/usr/local/bin/wp theme is-
installed sydney", "delta": "0:00:01.451018", "end": "2018-03-04 20:23:02.227557",
"msg": "non-zero return code", "rc": 1, "start": "2018-03-04 20:23:00.776539",
"stderr": "", "stderr_lines": [], "stdout": "", "stdout_lines": []}
...ignoring

TASK [roles/wordpress : set a fact if we don't need to install the theme]
*****
skipping: [box1]

TASK [roles/wordpress : set a fact if we need to install the theme]
*****
ok: [box1]

TASK [roles/wordpress : install the theme if we need to or ignore if not]
*****
changed: [box1]

RUNNING HANDLER [roles/stack-config : restart nginx]
*****
changed: [box1]

RUNNING HANDLER [roles/stack-config : restart php-fpm]
*****
changed: [box1]

PLAY RECAP
*****
box1 : ok=42 changed=37 unreachable=0 failed=0

```

As you can see during the playbook, we have fatal errors for the check to see whether we need to install WordPress, and also for the plugin and theme checks,

as we have accounted for these in the tasks the playbook ran as expected and installed the software stack, WordPress, plugins, and theme.

Rerunning the playbook gives us the following results for the sections we errored previously:

```
TASK [roles/wordpress : do we need to install wordpress?]
*****
changed: [box1]

TASK [roles/wordpress : install wordpress if needed]
*****
skipping: [box1]

TASK [roles/wordpress : do we need to install the plugins?]
*****
changed: [box1] => (item=jetpack)
changed: [box1] => (item=wp-super-cache)
changed: [box1] => (item=wordpress-seo)
changed: [box1] => (item=wordfence)
changed: [box1] => (item=nginx-helper)

TASK [roles/wordpress : set a fact if we don't need to install the plugins]
*****
ok: [box1]

TASK [roles/wordpress : set a fact if we need to install the plugins]
*****
skipping: [box1]

TASK [roles/wordpress : install the plugins if we need to or ignore if not]
*****
skipping: [box1] => (item=jetpack)
skipping: [box1] => (item=wp-super-cache)
skipping: [box1] => (item=wordpress-seo)
skipping: [box1] => (item=wordfence)
skipping: [box1] => (item=nginx-helper)

TASK [roles/wordpress : do we need to install the theme?]
*****
changed: [box1]

TASK [roles/wordpress : set a fact if we don't need to install the theme]
*****
ok: [box1]

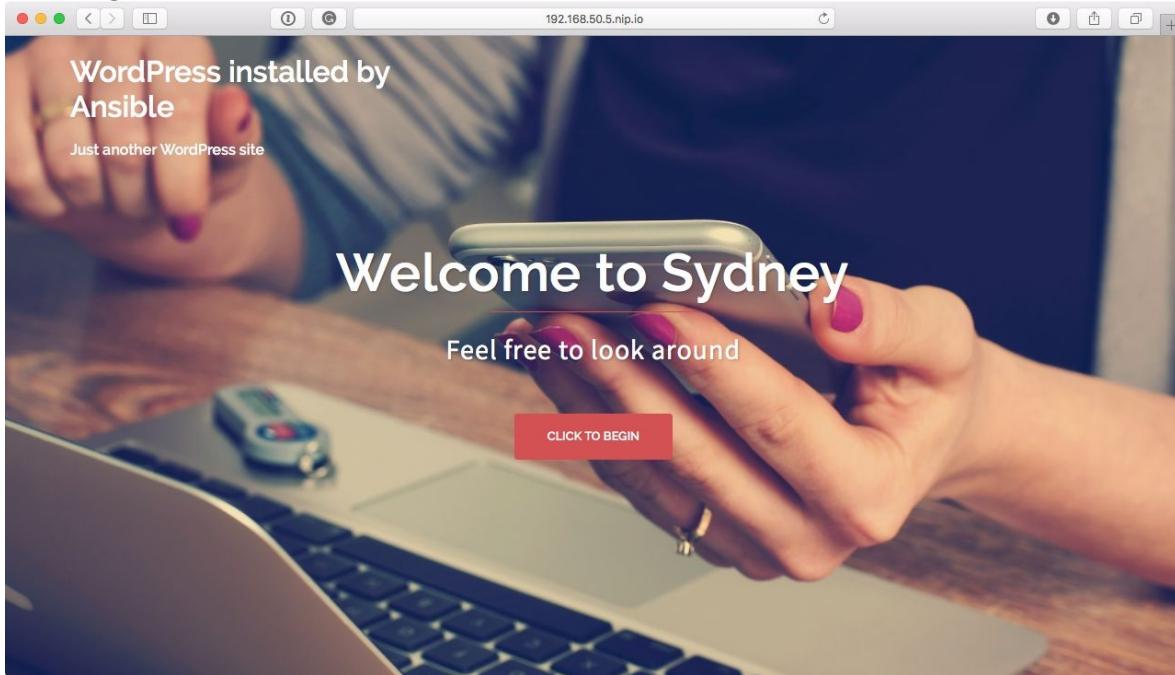
TASK [roles/wordpress : set a fact if we need to install the theme]
*****
skipping: [box1]

TASK [roles/wordpress : install the theme if we need to or ignore if not]
*****
skipping: [box1]

PLAY RECAP
*****
box1 : ok=34 changed=3 unreachable=0 failed=0
```

Now that WordPress is installed, we should be able to access in a browser by

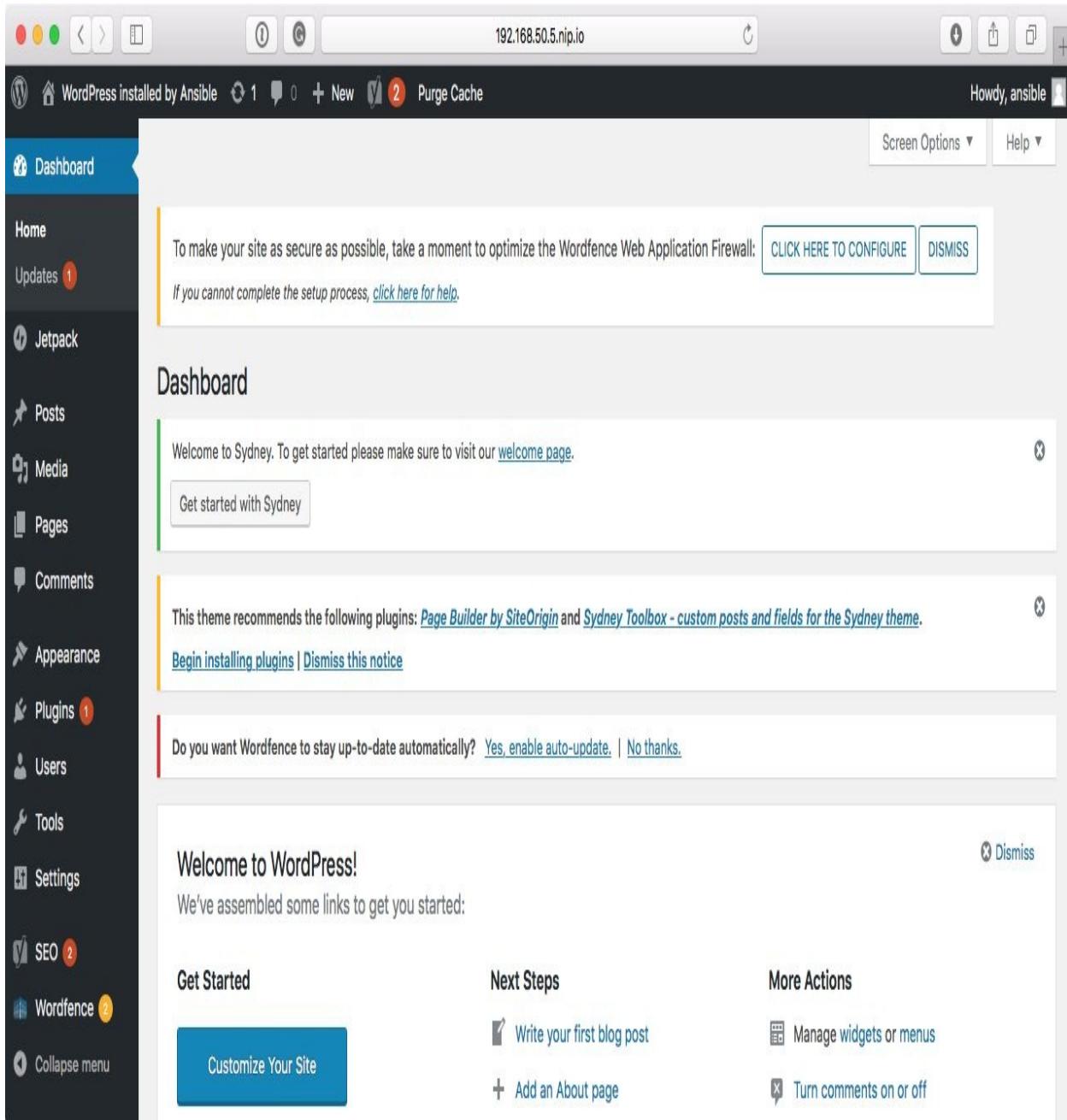
going to <http://192.168.50.5.nip.io/>. As you can see here, the theme we defined is running rather than the WordPress default theme:



Also, if you go to <http://192.168.50.5.nip.io/wp-admin/>, you should be able to log in to WordPress using the username and password we defined:

- Username: ansible
- Password: password

Once logged in, you should see a few messages about the plugins we installed during the playbook run needing to be configured:



Feel free to have a play with the WordPress installation; also, you can remove the Vagrant box by running:

```
| $ vagrant destroy
```

Then bring up a fresh copy and redeploy it using the commands at that start of this section.

Summary

In this chapter, we have reused a lot of the same principles we covered in the previous chapter and moved on to deploying a full application. What is good about this is that the process is both repeatable and just a single command.

So far, we have been targeting a CentOS 7 Vagrant box. If we ran our playbook against an Ubuntu Vagrant box, the playbook would give an error. In the next chapter, we will look at how we can target multiple operating systems using the same playbook.

Questions

1. Which fact gathered during `setup` module execution can we use to tell our playbook how many processors our target host has?
2. True or false: Using `backref` in the `lineinfile` module makes sure that no changes are applied if the regular expression is not matched.
3. Explain why we would want to build the logic into the playbook to check whether WordPress is already installed.
4. Which module do we use to define variables as part of a playbook run?
5. Which argument do we pass to the `shell` module to have the command we want to run executed in a directory of our choosing?
6. True or false: Setting MariaDB to bind to `127.0.0.1` will allow us to access it externally.
7. Change the theme of your WordPress site to one of your choice; see <https://wordpress.org/themes/> for some options.

Further reading

You can find out more information on the technologies we have covered in this chapter at the following links:

- **NGINX:** <http://nginx.org/>
- **WordPress:** <https://wordpress.org/>
- **WP-CLI:** <http://wp-cli.org>
- **CMS statistics from BuiltWith:** <https://trends.builtwith.com/cms>
- **The WordPress NGINX Codex:** <https://codex.wordpress.org/Nginx>
- **The Sydney WordPress theme:** <https://en-gb.wordpress.org/themes/sydney/>

The project pages for the plugins we installed can be found at:

- **Jetpack:** <https://en-gb.wordpress.org/plugins/jetpack/>
- **WP Super Cache:** <https://en-gb.wordpress.org/plugins/wp-super-cache/>
- **Yoast SEO:** <https://en-gb.wordpress.org/plugins/wordpress-seo/>
- **Wordfence:** <https://en-gb.wordpress.org/plugins/wordfence/>
- **NGINX Helper:** <https://wordpress.org/plugins/nginx-helper/>

Targeting Multiple Distributions

As mentioned at the end of the last chapter, so far we have been targeting a single operating system with our playbook. While this is great if we are only going to run our playbook against CentOS 7 hosts, that might not always be the case.

In this chapter, we are going to look at adapting our WordPress installation playbook to target an Ubuntu 17.04 server instance.

In this chapter, we will:

- Look at and implement core modules that are operating system-dependent
- Discuss and apply best practices for targeting multiple distributions
- See how we can target multiple hosts with an Ansible inventory

Technical requirements

In this chapter, we are going to be launching two Vagrant boxes so you will need Vagrant installed and access to the internet; the boxes themselves are around a 300 to 500 MB download each.

You will need to make a copy of the `lemp` folder from the last chapter and call it `lemp-multi` if you are going to follow along, adapting the roles as we work through them. If you are not following along, you can find a complete version of `lemp-multi` at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter06/lemp-multi>.

Launching multiple Vagrant boxes

Before we start to look at the changes we need to make to our Ansible playbook, we should look at how we are going to launch two Vagrant boxes running different operating systems side by side. It is possible to launch two Vagrant boxes from a single `Vagrantfile`; we will be using the following one:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

API_VERSION = "2"
DOMAIN      = "nip.io"
PRIVATE_KEY = "~/.ssh/id_rsa"
PUBLIC_KEY  = '~/.ssh/id_rsa.pub'
CENTOS_IP   = '192.168.50.6'
CENTOS_BOX  = 'centos/7'
UBUNTU_IP   = '192.168.50.7'
UBUNTU_BOX  = 'generic/ubuntu1704'

Vagrant.configure(API_VERSION) do |config|


  config.vm.define "centos" do |centos|
    centos.vm.box = CENTOS_BOX
    centos.vm.network "private_network", ip: CENTOS_IP
    centos.vm.host_name = CENTOS_IP + '.' + DOMAIN
    centos.ssh.insert_key = false
    centos.ssh.private_key_path = [PRIVATE_KEY,
      "~/.vagrant.d/insecure_private_key"]
    centos.vm.provision "file", source: PUBLIC_KEY, destination:
      "~/.ssh/authorized_keys"

    centos.vm.provider "virtualbox" do |v|
      v.memory = "2024"
      v.cpus = "2"
    end

    centos.vm.provider "vmware_fusion" do |v|
      v.vmx["memsize"] = "2024"
      v.vmx["numvcpus"] = "2"
    end
  end

  config.vm.define "ubuntu" do |ubuntu|
    ubuntu.vm.box = UBUNTU_BOX
    ubuntu.vm.network "private_network", ip: UBUNTU_IP
    ubuntu.vm.host_name = UBUNTU_IP + '.' + DOMAIN
    ubuntu.ssh.insert_key = false
    ubuntu.ssh.private_key_path = [PRIVATE_KEY,
      "~/.vagrant.d/insecure_private_key"]
    ubuntu.vm.provision "file", source: PUBLIC_KEY, destination:
      "~/.ssh/authorized_keys"

    ubuntu.vm.provider "virtualbox" do |v|
      v.memory = "2024"
      v.cpus = "2"
    end
  end
end
```

```

    end

    ubuntu.vm.provider "vmware_fusion" do |v|
      v.vmx["memsize"] = "2024"
      v.vmx["numvcpus"] = "2"
    end
  end
end

```

As you can see, we are defining two different boxes, one called `centos` and the other `ubuntu`, you should replace the `Vagrantfile` in the `lemp` folder you copied earlier.

We can launch the two machines using just one command; to use VirtualBox, we should run:

```
| $ vagrant up
```

Or to use VMware, we can run:

```
| $ vagrant up --provider=vmware_fusion
```

As you can see from the Terminal output here, this launched the two boxes:

```

Bringing machine 'centos' up with 'vmware_fusion' provider...
Bringing machine 'ubuntu' up with 'vmware_fusion' provider...
==> centos: Cloning VMware VM: 'centos/7'. This can take some time...
==> centos: Checking if box 'centos/7' is up to date...
==> centos: Verifying vmnet devices are healthy...
==> centos: Preparing network adapters...
==> centos: Starting the VMware VM...
==> centos: Waiting for the VM to receive an address...
==> centos: Forwarding ports...
  centos: -- 22 => 2222
==> centos: Waiting for machine to boot. This may take a few minutes...
  centos: SSH address: 127.0.0.1:2222
  centos: SSH username: vagrant
  centos: SSH auth method: private key
==> centos: Machine booted and ready!
==> centos: Setting hostname...
==> centos: Configuring network adapters within the VM...
  centos: SSH address: 127.0.0.1:2222
  centos: SSH username: vagrant
  centos: SSH auth method: private key
==> centos: Rsyncing folder: /Users/russ/lemp/ => /vagrant
==> centos: Running provisioner: file...
==> ubuntu: Cloning VMware VM: 'generic/ubuntu1704'. This can take some time...
==> ubuntu: Checking if box 'generic/ubuntu1704' is up to date...
==> ubuntu: Verifying vmnet devices are healthy...
==> ubuntu: Preparing network adapters...
==> ubuntu: Starting the VMware VM...
==> ubuntu: Waiting for the VM to receive an address...
==> ubuntu: Forwarding ports...
  ubuntu: -- 22 => 2222
==> ubuntu: Waiting for machine to boot. This may take a few minutes...

```

```
|     ubuntu: SSH address: 127.0.0.1:2222
|     ubuntu: SSH username: vagrant
|     ubuntu: SSH auth method: private key
|=> ubuntu: Machine booted and ready!
|=> ubuntu: Setting hostname...
|=> ubuntu: Configuring network adapters within the VM...
|=> ubuntu: Running provisioner: file...
```

Once the boxes are up and running, you can SSH to them using the machine name:

```
| $ vagrant ssh centos
| $ vagrant ssh ubuntu
```

Now that we have our two boxes running on two different operating systems, we can discuss the changes we need to make to our playbook. First of all, let's look at how the changes to the `vagrantfile` will affect our host inventory file, as you can see from this file:

```
| centos ansible_host=192.168.50.6.nip.io
| ubuntu ansible_host=192.168.50.7.nip.io
|
| [wordpress]
|   centos
|   ubuntu
|
| [wordpress:vars]
|   ansible_connection=ssh
|   ansible_user=vagrant
|   ansible_private_key_file=~/ssh/id_rsa
|   host_key_checking=False
```

We now have two hosts, one called `centos` and the other `ubuntu`, and we are putting them in a group called `wordpress` where we are setting some common variables. You should update your `production` file, as we will be using it in the next section.

Multi-operating system considerations

Looking at each of the core Ansible modules used in the three roles, `stack-install`, `stack-config`, and `wordpress`, we are using a few that will not work on our newly introduced Ubuntu box. Let's quickly work through each one and see what we need to take into account when targeting two very different operating systems:

- `yum`: The `yum` module is the package manager used for Red Hat-based machines such as CentOS, as Ubuntu is based on Debian, which uses `apt`. We will need to break out the parts of our playbook that uses the `yum` module to use the `apt` module instead.
- `yum_repository`: As mentioned, we will need to use an `apt` equivalent module, which is `apt_repository`.
- `user`: The `user` module works pretty much the same on both operating systems, as we are not giving our user escalated privileges. There aren't any special considerations we need to make, other than double-checking the correct group is available.
- `template`, `file`, `copy`, and `lineinfile`: All four of these modules will work as expected; the only consideration we need to make is checking that we are copying the files to the correct location on the boxes.
- `service`: The service module should be the same on both operating systems, so we should be fine.
- `mysql_user` and `mysql_db`: As you would expect, once MySQL is installed and started, both of these will work across both operating systems.
- `selinux_permissive`: SELinux is primarily for operating systems based on Red Hat, so we will need to find an alternative.
- `get_url`, `stat`, `shell`, and `set_fact`: These should all work consistently across both of our target operating systems.

Now we know which parts of our existing playbook we need to review when running on Ubuntu versus running CentOS, we can make a start on making our roles work on both of the operating systems.

Adapting the roles

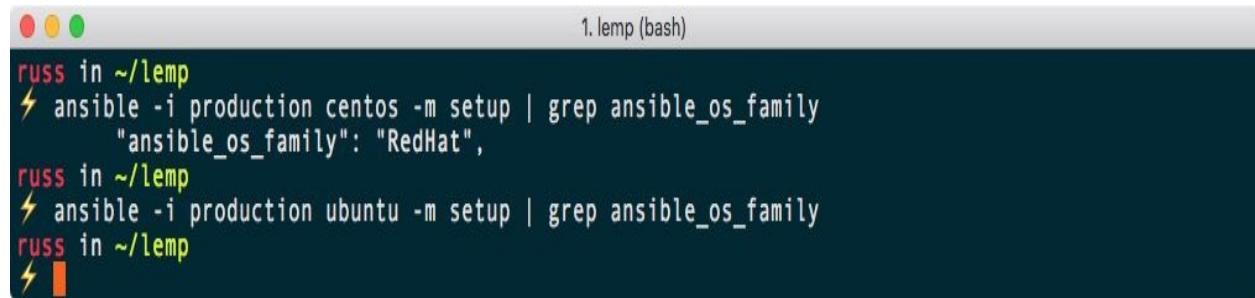
So how do we go about building the logic into our roles to only execute certain parts of the roles on different operating systems, and also as we know that package names will be different? How do we define different sets of variables per operating system?

Operating system family

We have looked at the `setup` module in previous chapters; this is the module that gathers facts about our target hosts. One of these facts is `ansible_os_family`; this tells us the type of operating system we are running. Let's check on both of our boxes:

```
$ ansible -i production centos -m setup | grep ansible_os_family
$ ansible -i production ubuntu -m setup | grep ansible_os_family
```

As you can see from the following Terminal output, the CentOS box returns Red Hat, which is expected. However, the Ubuntu box does not return any information:



```
russ in ~/lemp
$ ansible -i production centos -m setup | grep ansible_os_family
    "ansible_os_family": "RedHat",
russ in ~/lemp
$ ansible -i production ubuntu -m setup | grep ansible_os_family
russ in ~/lemp
$
```

Let's take a look at why this is. First of all, we can rerun the command, but this time minus the `grep` so we can see the full output:

```
| $ ansible -i production ubuntu -m setup
```

This should give you something like the following results:

```
1. lemp (bash)
russ in ~/lemp
⚡ ansible -i production ubuntu -m setup
ubuntu | FAILED! => {
    "changed": false,
    "module_stderr": "Shared connection to 192.168.50.7.nip.io closed.\r\n",
    "module_stdout": "/bin/sh: 1: /usr/bin/python: not found\r\n",
    "msg": "MODULE FAILURE",
    "rc": 0
}
russ in ~/lemp
⚡
```

Oh, we are getting an error. Why is it reporting there is no Python installed? Running the following will SSH into the box:

```
| $ vagrant ssh ubuntu
```

Once logged in using SSH, running `which python` will show us the path to the Python binary. As you can see, there isn't one installed as we get no path returned. So what about Python 3? Running `which python3` does return a binary:

```
1. vagrant@192: ~ (ssh)
russ in ~/lemp
⚡ vagrant ssh ubuntu
Welcome to Ubuntu 17.04 (GNU/Linux 4.10.0-42-generic x86_64)

 * Documentation: https://help.ubuntu.com
 * Management:   https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

Last login: Sat Mar 10 03:36:33 2018 from 192.168.50.1
vagrant@192:~$ which python
vagrant@192:~$ which python3
/usr/bin/python3
vagrant@192:~$
```

Let's close our SSH session by running `exit`.

What should we do about this? As we are running a version of Ansible that is later than 2.2, we can tell Ansible to use `/usr/bin/python3` rather than its default of `/usr/bin/python`. To do this, we need to update our host inventory file so that just the Ubuntu host gets the `ansible_python_interpreter` variable added along with the updated path.

There are a few ways to achieve this; however, for now, let's just update the following line in the `production` host inventory file:

```
| ubuntu ansible_host=192.168.50.7.nip.io
```

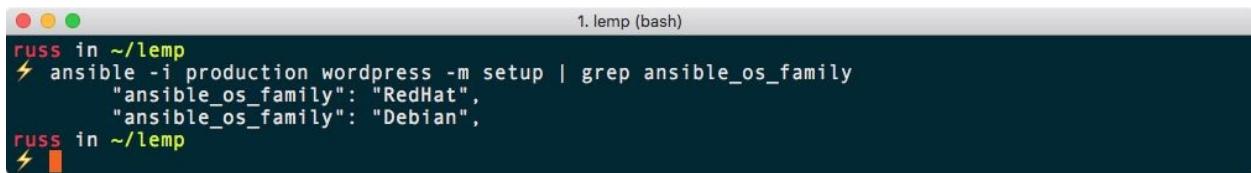
So it reads as follows:

```
| ubuntu ansible_host=192.168.50.7.nip.io ansible_python_interpreter=/usr/bin/python3
```

Once updated, we should be able to run the following command:

```
| $ ansible -i production wordpress -m setup | grep ansible_os_family
```

The following screenshot shows an output for the preceding command:



A screenshot of a terminal window titled "1. lemp (bash)". The terminal shows the command: "ansible -i production wordpress -m setup | grep ansible_os_family". The output is: "ansible_os_family": "RedHat", "ansible_os_family": "Debian",. The terminal has a dark background with light-colored text. There are three colored dots (red, yellow, green) at the top left and a small lightning bolt icon at the bottom left.

As you can see, we are targeting the host group of `wordpress`, which contains both of our boxes in it, and, as expected, we get `RedHat` for the CentOS box and the Ubuntu box is now returning `Debian`. Now that we have a way of identifying which operating system is in use on each host, we can make a start adapting the roles.

The stack-install role

As you may have already guessed the bulk of this role, it is only tasks that call `yum` related modules and we have already mentioned that this is going to be changed.

The first part of the role we are going to look at is the content of `roles/stack-install/tasks/main.yml`. At the moment, this file contains the tasks that install our desired repositories and packages using the `yum` and `yum_repository` modules.

We need to update the file, but first, take the existing content and save it as a file called `roles/stack-install/tasks/install-centos.yml`. Once you have the copied the content, update `roles/stack-install/tasks/main.yml` so it has this content:

```
---
- name: include the operating system specific variables
  include_vars: "{{ ansible_os_family }}.yml"
- name: install the stack on centos
  import_tasks: install-centos.yml
  when: ansible_os_family == 'RedHat'
- name: install the stack on ubuntu
  import_tasks: install-ubuntu.yml
  when: ansible_os_family == 'Debian'
```

As you can see, we are using the `ansible_os_family` variable to include variables and also different tasks.

The task will include one of the following files, depending on which operating system the task is being executed on:

- `roles/stack-install/vars/RedHat.yml`
- `roles/stack-install/vars/Debian.yml`

It will then include one of the following two files, which contain the tasks for the operating system:

- `install-centos.yml`
- `install-ubuntu.yml`

We already know that `install-centos.yml` contains old content of our `main.yml` file; as the package name and repository URLs will also be changing, we should move the content of `roles/stack-install/default/main.yml` to `roles/stack-install/vars/RedHat.yml`, leaving `roles/stack-install/default/main.yml` empty.

Now that we have the CentOS portion of our role defined, we can look at the Ubuntu parts, starting with the content of `roles/stack-install/vars/Debian.yml`:

```
---
repo_packages:
  - "deb [arch=amd64,i386] http://mirror.sax.uk.as61049.net/mariadb/repo/10.1/ubuntu {{ ansible_distribution_release }} main"
  - "deb http://nginx.org/packages/mainline/ubuntu/ {{ ansible_distribution_release }}"
  - "deb-src http://nginx.org/packages/mainline/ubuntu/ {{ ansible_distribution_release }}"
} nginx"

repo_keys:
  - { key_server: "keyserver.ubuntu.com", key: "0xF1656F24C74CD1D8" }

repo_keys_url:
  - "http://nginx.org/keys/nginx_signing.key"

system_packages:
  - "software-properties-common"
  - "python3-mysqldb"
  - "acl"

stack_packages:
  - "nginx"
  - "mariadb-server"
  - "php7.0"
  - "php7.0-cli"
  - "php7.0-fpm"
  - "php7.0-gd"
  - "php7.0-json"
  - "php7.0-mbstring"
  - "php7.0-mysqlnd"
  - "php7.0-soap"
  - "php7.0-xml"
  - "php7.0-xmlrpc"

extra_packages:
  - "vim"
  - "git"
  - "unzip"
```

As you can see, while we are keeping the `system_packages`, `stack_packages`, and `extra_packages` variables, we have different package names in there. We have a similar situation with `repo_packages`, where we have updated URLs, as the CentOS repositories will not work with Ubuntu. Finally, we have introduced two new variables, `repo_keys`, and `repo_keys_urls`; we will look at what these are used for shortly.

The final file we need to cover for our new role is `roles/stack-install/tasks/install-ubuntu.yml`. Like `install-centos.yml`, this contains the tasks to add the additional repositories we need and install the packages.

First of all, we need to install a few of the tools we need to continue with the rest of the tasks; these have been defined in the `system_packages` variable, so we simply have to add the following task:

```
- name: update cache and install the system packages
  apt:
    name: "{{ item }}"
    update_cache: "yes"
  with_items: "{{ system_packages }}"
```

Now that we have basic prerequisites installed, we can add the keys for the repositories we will be adding:

```
- name: add the apt keys from a key server
  apt_key:
    keyserver: "{{ item.key_server }}"
    id: "{{ item.key }}"
  with_items: "{{ repo_keys }}"

- name: add the apt keys from a URL
  apt_key:
    url: "{{ item }}"
    state: present
  with_items: "{{ repo_keys_url }}"
```

The first task adds keys from the official Ubuntu key store, and the second task downloads the keys from a URL. In our case, we are adding one key for the official MariaDB repository and one for the NGINX mainline repository; without these keys, we would not be able to add the repositories without generating an error about them not being trusted.

The task for adding repositories looks like the following; it cycles through the repository URLs in the `repo_packages` variable:

```
- name: install the repo packages
  apt_repository:
    repo: "{{ item }}"
    state: "present"
    update_cache: "yes"
  with_items: "{{ repo_packages }}"
```

The final part of the playbook installs the remaining packages:

```
- name: install the stack packages
  apt:
```

```
|   name: "{{ item }}"
|   state: "installed"
with_items: "{{ stack_packages + extra_packages }}"
```

Now we have updated our `stack-install` role, we need to do the same for the `stack-config` one.

The stack-config role

A lot of the modules we are using in this role will work fine on both of our target operating systems, so in this role, we are tweaking things like paths to configuration files and so on. Rather than list out the entire content of the `roles/stack-config/tasks/main.yml` file, I will just highlight the changes that need to be made, starting with the following task that should be right at the top of the file:

```
| - name: include the operating system specific variables  
|   include_vars: "{{ ansible_os_family }}.yml"
```

This will load in the variables that contain the paths we need to use later in the role; the content of `roles/stack-config/vars/RedHat.yml` is:

```
| ---  
| php_fpm_path: "/etc/php-fpm.d/www.conf"  
| php_ini_path: /etc/php.ini  
| php_service_name: "php-fpm"
```

And the content of `roles/stack-config/vars/Debian.yml` is:

```
| php_fpm_path: "/etc/php/7.0/fpm/pool.d/www.conf"  
| php_ini_path: "/etc/php/7.0/fpm/php.ini"  
| php_service_name: "php7.0-fpm"
```

As you can see, most of the changes we need to make are around the location of the PHP configuration files. Before we get those, we need to create the WordPress user back in our `roles/stack-config/tasks/main.yml` file. Because PHP-FPM runs under a different group by default on Ubuntu, there is no PHP-FPM group created, so let's create one, making sure we add these tasks before the `add` the `wordpress` user task:

```
| - name: add the wordpress group  
|   group:  
|     name: "{{ wordpress_system.group }}"  
|     state: "{{ wordpress_system.state }}"
```

Next up, there is no `/var/www/` folder created on Ubuntu, so we will need to create the folder:

```
| - name: create the global directory in /etc/nginx/
```

```
file:
  dest: "/var/www/"
  state: "directory"
  mode: "0755"
```

Both the group and folder are already there on the CentOS box, so these tasks should just say `ok`. Once they have been created, the user will be created without errors on both boxes with no changes to the `add the wordpress user` task.

All of the tasks that deploy the NGINX configuration will work without any changes, so we can move on to the PHP configuration:

```
- name: copy the www.conf to /etc/php-fpm.d/
  template:
    src: "php-fpmd-www.conf.j2"
    dest: "{{ php_fpm_path }}"
    notify: "restart php-fpm"

- name: configure php.ini
  lineinfile:
    dest: "{{ php_ini_path }}"
    regexp: "{{ item.regexp }}"
    line: "{{ item.replace }}"
    backup: "yes"
    backrefs: "yes"
    with_items: "{{ php.ini }}"
    notify: "restart php-fpm"
```

As you can see, both of these tasks have been updated to include the paths relevant to the operating system the playbook is currently targeting.

The `restart php-fpm` handler has also been updated as the PHP-FPM service on the two operating systems has a different name; this task should replace the existing one in `roles/stack-config/handlers/main.yml`:

```
- name: "restart php-fpm"
  service:
    name: "{{ php_service_name }}"
    state: "restarted"
    enabled: "yes"
```

Likewise, back in `roles/stack-config/tasks/main.yml` the task that starts PHP-FPM should be updated as per this task:

```
- name: start php-fpm
  service:
    name: "{{ php_service_name }}"
    state: "started"
```

The next two changes are to make the following tasks only run on CentOS

boxes:

```
- name: configure the mariadb bind address
  lineinfile:
    dest: "{{ mariadb.server_config }}"
    regexp: "#bind-address=0.0.0.0"
    line: "bind-address={{ mariadb.bind }}"
    backup: "yes"
    backrefs: "yes"
  when: ansible_os_family == 'RedHat'
```

This is because the default configuration on Ubuntu for MariaDB does not contain `bind-address`, so we are skipping it; the next and final task is as follows:

```
- name: set the selinux allowing httpd_t to be permissive is required
  selinux_permissive:
    name: httpd_t
    permissive: true
  when: selinux.http_permissive == true and ansible_os_family == 'RedHat'
```

We are skipping this on the Ubuntu box because SELinux is not installed and does not work with Ubuntu.

The wordpress role

There a few small changes to the `wordpress` role; the first change is an update to `roles/wordpress/defaults/main.yml`:

```
wordpress:  
domain: "http://{{ wordpress_domain }}/"  
title: "WordPress installed by Ansible on {{ os_family }}"
```

As you can see, we have updated the `wordpress.domain` to include the `wordpress_domain` variable, and `wordpress.title` now includes the `os_family` variable; we are setting both of these in the `roles/wordpress/tasks/main.yml` file by adding the following task:

```
- name: set a fact for the wordpress domain  
set_fact:  
wordpress_domain: "{{ ansible_ssh_host }}"  
os_family: "{{ ansible_os_family }}"
```

The reason we are doing this here is that Vagrant does not correctly set the hostname of our Ubuntu box to the fully qualified domain name, such as `192.168.50.7.nip.io`, so we are using the host we are SSHing to, which is defined in the `production` inventory hosts file. The remainder of this role remains as is.

Running the playbook

There are no changes to our `site.yml` file, meaning that we just need to run the following command to start the playbook run: **\$ ansible-playbook -i production site.yml**

This will run through the playbook, giving the following output; please note that I have trimmed a few parts of the playbook output:

```
PLAY [wordpress]
TASK [Gathering Facts]
ok: [centos]
ok: [ubuntu]

TASK [roles/stack-install : include the operating system specific variables]
ok: [centos]
ok: [ubuntu]

TASK [roles/stack-install : install the repo packages]
skipping: [ubuntu] => (item=[])
changed: [centos] => (item=[u'epel-release', u'https://centos7.iuscommunity.org/ius-release.rpm'])

TASK [roles/stack-install : add the NGINX mainline repo]
skipping: [ubuntu]
changed: [centos]

TASK [roles/stack-install : update all of the installed packages]
skipping: [ubuntu]
changed: [centos]

TASK [roles/stack-install : remove the packages so that they can be replaced]
skipping: [ubuntu]
changed: [centos] => (item=[u'mariadb-libs.x86_64'])

TASK [roles/stack-install : install the stack packages]
skipping: [ubuntu] => (item=[])
changed: [centos] => (item=[u'postfix', u'MySQL-python', u'policycoreutils-python',
u'nginx', u'mariadb101u', u'mariadb101u-server', u'mariadb101u-config', u'mariadb101u-common',
u'mariadb101u-libs', u'php72u', u'php72u-bcmath', u'php72u-cli', u'php72u-common',
u'php72u-dba', u'php72u-fpm', u'php72u-fpm-nginx', u'php72u-gd', u'php72u-intl',
u'php72u-json', u'php72u-mbstring', u'php72u-mysqld', u'php72u-process',
u'php72u-snmp', u'php72u-soap', u'php72u-xml', u'php72u-xmlrpc', u'vim-enhanced',
u'git', u'unzip'])

TASK [roles/stack-install : update cache and install the system packages]
skipping: [centos] => (item=[])
changed: [ubuntu] => (item=[u'software-properties-common', u'python3-mysqldb', u'acl'])

TASK [roles/stack-install : add the apt keys from a key server]
skipping: [centos]
changed: [ubuntu] => (item={u'key_server': u'keyserver.ubuntu.com', u'key': u'0xF1656F24C74CD1D8'})
```

```
TASK [roles/stack-install : add the apt keys from a URL]
skipping: [centos]
changed: [ubuntu] => (item=http://nginx.org/keys/nginx_signing.key)

TASK [roles/stack-install : install the repo packages]
skipping: [centos] => (item=epel-release)
skipping: [centos] => (item=https://centos7.iuscommunity.org/ius-release.rpm)
changed: [ubuntu] => (item=deb [arch=amd64,i386]
http://mirror.sax.uk.as61049.net/mariadb/repo/10.1/ubuntu zesty main)
changed: [ubuntu] => (item=deb http://nginx.org/packages/mainline/ubuntu/ zesty nginx)
changed: [ubuntu] => (item=deb-src http://nginx.org/packages/mainline/ubuntu/ zesty
nginx)

TASK [roles/stack-install : install the stack packages]
skipping: [centos] => (item[])
changed: [ubuntu] => (item=[u'nginx', u'mariadb-server', u'php7.0', u'php7.0-cli',
u'php7.0-fpm', u'php7.0-gd', u'php7.0-json', u'php7.0-mbstring', u'php7.0-mysqlnd',
u'php7.0-soap', u'php7.0-xml', u'php7.0-xmlrpc', u'vim', u'git', u'unzip'])

TASK [roles/stack-config : include the operating system specific variables]
ok: [centos]
ok: [ubuntu]

TASK [roles/stack-config : add the wordpress group]
ok: [centos]

TASK [roles/stack-config : create the global directory in /etc/nginx/]
changed: [ubuntu]
ok: [centos]

TASK [roles/stack-config : add the wordpress user]
changed: [centos]
changed: [ubuntu]

TASK [roles/stack-config : copy the nginx.conf to /etc/nginx/]
changed: [ubuntu]
changed: [centos]

TASK [roles/stack-config : create the global directory in /etc/nginx/]
changed: [ubuntu]
changed: [centos]

TASK [roles/stack-config : copy the restrictions.conf to /etc/nginx/global/]
changed: [ubuntu]
changed: [centos]

TASK [roles/stack-config : copy the wordpress_shared.conf to /etc/nginx/global/]
changed: [ubuntu]
changed: [centos]

TASK [roles/stack-config : copy the default.conf to /etc/nginx/conf.d/]
changed: [ubuntu]
changed: [centos]

TASK [roles/stack-config : copy the www.conf to /etc/php-fpm.d/]
changed: [ubuntu]
changed: [centos]

TASK [roles/stack-config : configure php.ini]
changed: [ubuntu] => (item={u'regexp': u'^;date.timezone =', u'replace':
u'date.timezone = Europe/London'})
changed: [centos] => (item={u'regexp': u'^;date.timezone =', u'replace':
u'date.timezone = Europe/London'})
```

```
ok: [ubuntu] => (item={u'regexp': u'^expose_php = On', u'replace': u'expose_php = Off'})
changed: [centos] => (item={u'regexp': u'^expose_php = On', u'replace': u'expose_php = Off'})
changed: [ubuntu] => (item={u'regexp': u'^upload_max_filesize = 2M', u'replace': u'upload_max_filesize = 20M'})
changed: [centos] => (item={u'regexp': u'^upload_max_filesize = 2M', u'replace': u'upload_max_filesize = 20M'})

TASK [roles/stack-config : start php-fpm]
changed: [ubuntu]
changed: [centos]

TASK [roles/stack-config : start nginx]
changed: [ubuntu]
changed: [centos]

TASK [roles/stack-config : configure the mariadb bind address]
skipping: [ubuntu]
changed: [centos]

TASK [roles/stack-config : start mariadb]
ok: [ubuntu]
changed: [centos]

TASK [roles/stack-config : change mysql root password]
changed: [centos] => (item=127.0.0.1)
changed: [ubuntu] => (item=127.0.0.1)
changed: [centos] => (item=:1)
changed: [ubuntu] => (item=:1)
changed: [ubuntu] => (item=192)
changed: [centos] => (item=192.168.50.6.nip.io)
changed: [ubuntu] => (item=localhost)
changed: [centos] => (item=localhost)

TASK [roles/stack-config : set up .my.cnf file]
changed: [ubuntu]
changed: [centos]

TASK [roles/stack-config : delete anonymous MySQL user]
ok: [ubuntu] => (item=127.0.0.1)
ok: [centos] => (item=127.0.0.1)
ok: [ubuntu] => (item=:1)
ok: [centos] => (item=:1)
ok: [ubuntu] => (item=192)
changed: [centos] => (item=192.168.50.6.nip.io)
ok: [ubuntu] => (item=localhost)
changed: [centos] => (item=localhost)

TASK [roles/stack-config : remove the MySQL test database]
ok: [ubuntu]
changed: [centos]

TASK [roles/stack-config : set the selinux allowing httpd_t to be permissive is required]
skipping: [ubuntu]
changed: [centos]

TASK [roles/wordpress : set a fact for the wordpress domain]
ok: [centos]
ok: [ubuntu]

TASK [roles/wordpress : download wp-cli]
changed: [ubuntu]
```

```
changed: [centos]

TASK [roles/wordpress : update permissions of wp-cli to allow anyone to execute it]
changed: [ubuntu]
changed: [centos]

TASK [roles/wordpress : create the wordpress database]
changed: [ubuntu]
changed: [centos]

TASK [roles/wordpress : create the user for the wordpress database]
changed: [ubuntu] => (item=127.0.0.1)
changed: [centos] => (item=127.0.0.1)
ok: [ubuntu] => (item=:1)
ok: [centos] => (item=:1)
ok: [ubuntu] => (item=192)
ok: [centos] => (item=192.168.50.6.nip.io)
ok: [ubuntu] => (item=localhost)
ok: [centos] => (item=localhost)

TASK [roles/wordpress : are the wordpress files already there?]
ok: [ubuntu]
ok: [centos]

TASK [roles/wordpress : download wordpress]
changed: [ubuntu]
changed: [centos]

TASK [roles/wordpress : set the correct permissions on the homedir]
ok: [ubuntu]
changed: [centos]

TASK [roles/wordpress : is wordpress already configured?]
ok: [centos]
ok: [ubuntu]

TASK [roles/wordpress : configure wordpress]
changed: [ubuntu]
changed: [centos]

TASK [roles/wordpress : do we need to install wordpress?]
fatal: [ubuntu]: FAILED! =>
...ignoring
fatal: [centos]: FAILED! =>
...ignoring

TASK [roles/wordpress : install wordpress if needed]
changed: [ubuntu]
changed: [centos]

TASK [roles/wordpress : do we need to install the plugins?]
failed: [ubuntu] (item=jetpack) =>
failed: [ubuntu] (item=wp-super-cache) =>
failed: [ubuntu] (item=wordpress-seo) =>
failed: [centos] (item=jetpack) =>
failed: [ubuntu] (item=wordfence) =>
failed: [centos] (item=wp-super-cache) =>
failed: [ubuntu] (item=nginx-helper) =>
failed: [centos] (item=wordpress-seo) =>
failed: [centos] (item=wordfence) =>
failed: [centos] (item=nginx-helper) =>

TASK [roles/wordpress : set a fact if we don't need to install the plugins]
skipping: [centos]
```

```

skipping: [ubuntu]

TASK [roles/wordpress : set a fact if we need to install the plugins]
ok: [centos]
ok: [ubuntu]

TASK [roles/wordpress : install the plugins if we need to or ignore if not]
changed: [centos] => (item=jetpack)
changed: [ubuntu] => (item=jetpack)
changed: [ubuntu] => (item=wp-super-cache)
changed: [centos] => (item=wp-super-cache)
changed: [ubuntu] => (item=wordpress-seo)
changed: [centos] => (item=wordpress-seo)
changed: [ubuntu] => (item=wordfence)
changed: [centos] => (item=wordfence)
changed: [ubuntu] => (item=nginx-helper)
changed: [centos] => (item=nginx-helper)

TASK [roles/wordpress : do we need to install the theme?]
fatal: [centos]: FAILED! =>
fatal: [ubuntu]: FAILED! =>

TASK [roles/wordpress : set a fact if we don't need to install the theme]
skipping: [centos]
skipping: [ubuntu]

TASK [roles/wordpress : set a fact if we need to install the theme]
ok: [centos]
ok: [ubuntu]

TASK [roles/wordpress : install the theme if we need to or ignore if not]
changed: [centos]
changed: [ubuntu]

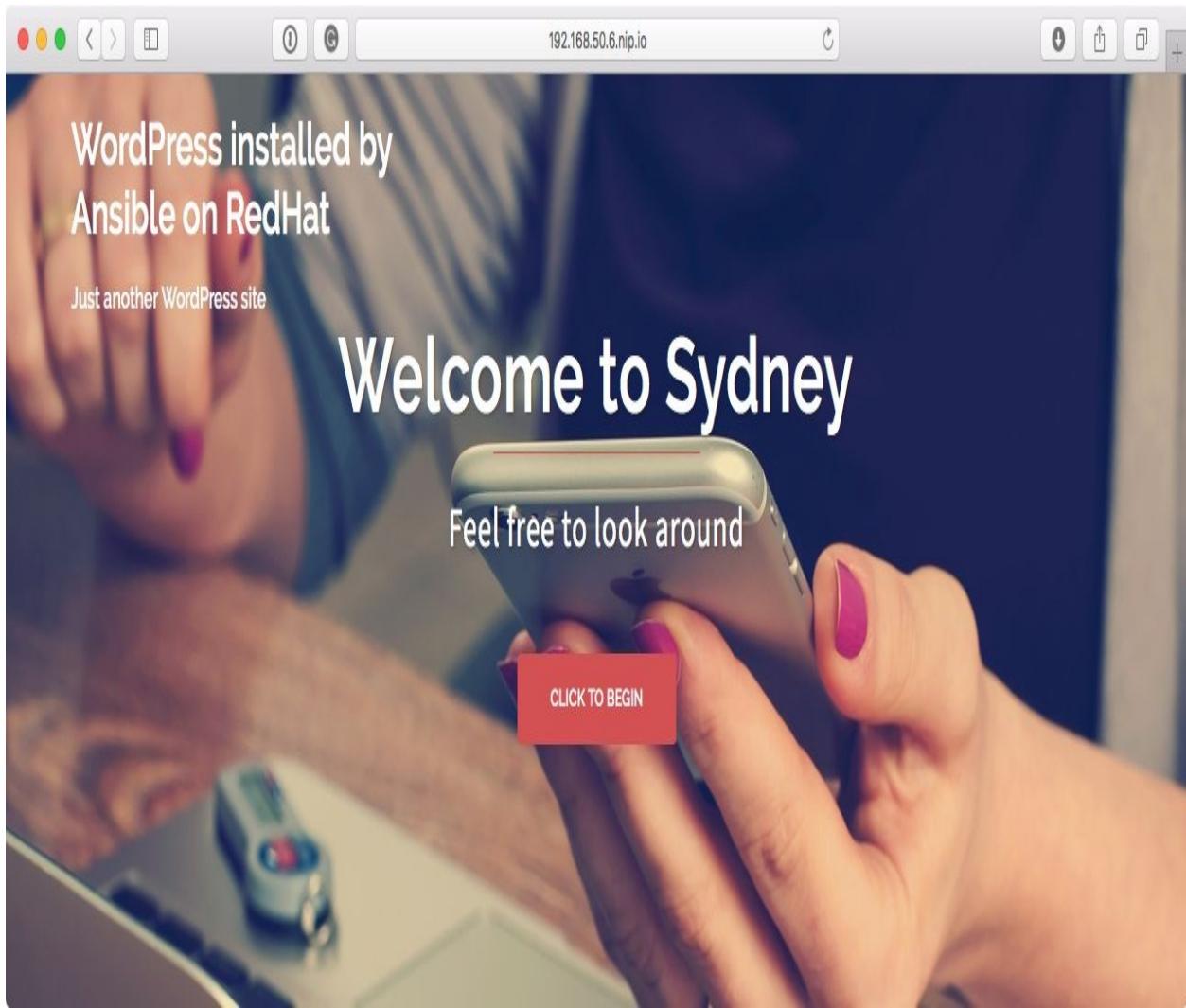
RUNNING HANDLER [roles/stack-config : restart nginx]
changed: [ubuntu]
changed: [centos]

RUNNING HANDLER [roles/stack-config : restart php-fpm]
changed: [ubuntu]
changed: [centos]

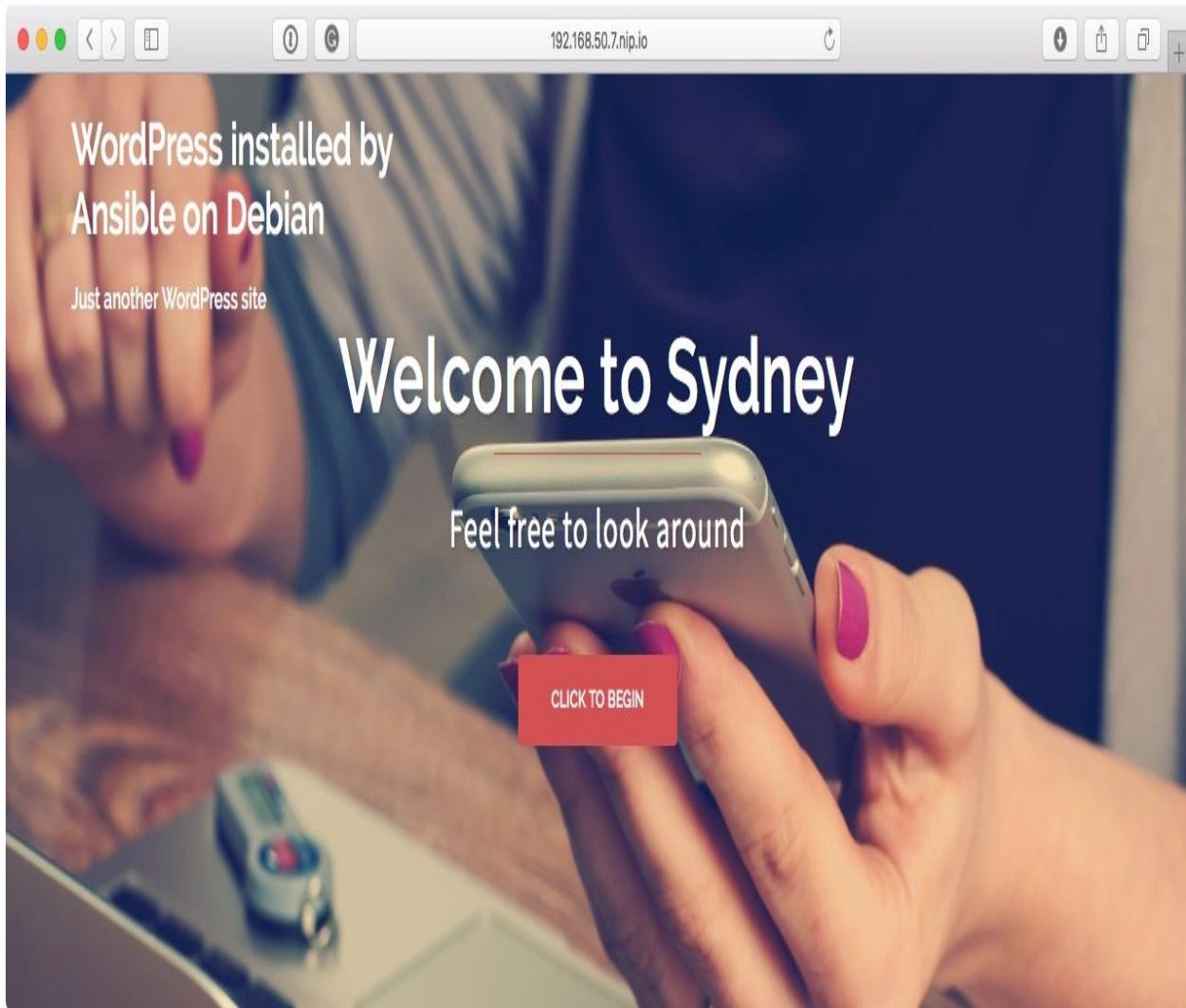
PLAY RECAP
centos : ok=47 changed=37 unreachable=0 failed=0
ubuntu : ok=45 changed=33 unreachable=0 failed=0

```

Once the playbook has finished, you should be able to access <http://192.168.50.6.nip.io/> in your browser, and you should see WordPress showing that it is installed on a Red Hat-based operating system:



Going to `http://192.168.50.7.nip.io/` will show the same theme, but it should state that it is running in a Debian-based operating system, as in this screenshot:



You can try rerunning your playbook to see what results are returned, and also you can remove the Vagrant boxes by running:

```
| $ vagrant destroy
```

You will be asked if you want to remove each machine one at a time; just answer yes to each of the two prompts.

Summary

In this chapter, we have adapted our WordPress installation playbook to target multiple operating systems. We did this by using Ansible's inbuilt auditing module to determine which operating system the playbook is running against and running just the tasks that will work on the target operating system.

In the next chapter, we are going to make a start at looking at some core Ansible modules that deal with networking.

Questions

1. True or false: We need to double-check every task in our playbook so it will work on both operating systems.
2. Which configuration option allows us to define the path to Python, Ansible will use?
3. Explain why we need to make changes to the tasks that configured and interact with the PHP-FPM service.
4. True or false: The package names for each of the operating systems correspond exactly.
5. Update the playbook so that a different theme is installed on each of the different hosts.

Further reading

You can find out more about the Ubuntu operating system
at <https://www.ubuntu.com>.

The Core Network Modules

In this chapter, we will take a look at the core network modules that ship with Ansible. Due to the requirements of these modules, we will only be touching upon the functionality these modules provide with some use cases and examples.

The following topics will be covered in this chapter:

- The core network modules
- Interacting with a server's local firewall
- Interacting with a network device

Technical requirements

In this chapter, we will be launching a Vagrant box running a software firewall. You will need Vagrant installed and access to the internet; the Vagrant box is around a 400-MB download. A complete version playbook we will be working through in this chapter can be accessed at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter07/vyos>.

Manufacturer and device support

So far, we have been looking at modules that interact with servers. In our case, they have all been running locally. In later chapters, we will be communicating with remotely hosted servers. Before we start interacting with remote servers, we should cover the core network modules.

These modules have all been designed to interact with and manage the configuration of various network devices, from traditional top-of-rack switches and fully virtualized network infrastructure, to firewalls and load balancers. There are quite a lot of very different devices supported by Ansible, from open source virtual appliances all the way through to solutions that can potentially cost over \$500,000, depending on your configuration.

The modules

I have listed each of the devices and operating systems here. For each of them, there is a shorthand name, which is highlighted in bold. Each of these is the prefix for the module. For example, in the first device, there is a module called **a10_server**, which is used to manage **Server Load Balancer (SLB)** objects using the aXAPIv2 API.

A10 Networks

The **A10** modules support A10 Networks AX, SoftAX, Thunder, and vThunder devices. These are all application delivery platforms, which provide load balancing. Among other features, the half-dozen modules allow you to manage load balancing and virtual hosts on both the physical and virtual devices.

Cisco Application Centric Infrastructure (ACI)

The 50+ **ACI** modules are used to manage all aspects of Cisco's ACI, which is to be expected of Cisco's next generation API-driven networking stack.

Cisco AireOS

The two **AireOS** modules allow you to interact with the Cisco Wireless LAN Controllers running AireOS. One of the modules allows you to run commands directly on the devices, and the other is for managing the configuration.

Apstra Operating System (AOS)

The dozen or so **AOS** modules are all marked as deprecated, as they don't support AOS 2.1 or later. The modules will be replaced before they are removed in an upcoming Ansible release, Version 2.9 to be exact.

Aruba Mobility Controller

There are just two **Aruba** modules. These allow you to manage the configuration and execute commands on the Aruba Mobility Controllers from Hewlett Packard.

Cisco Adaptive Security Appliance (ASA)

With the three **ASA** modules, you can manage access lists, as well as run commands and manage the configuration of both the physical and virtual Cisco ASA-powered devices.

Avi Networks

At the time of writing, there are 65 **Avi** modules, which allow you to interact with all aspects of the Avi application services platform, including the load-balancing and **web application firewall (WAF)** features.

Big Switch Networks

There are three Big Switch Network modules. One, the **Big Cloud Fabric (BCF)**, allows you to create and delete BCF switches. The other two modules allow you to create **Big Monitoring Fabric (Big Mon)** service chains and policies.

Citrix Netscaler

There is currently a single deprecated **Netscaler** module. It is due to be removed for Ansible 2.8. This gives you plenty of time to move over to the new modules. The single module has been replaced by 14 others, which allow you to manage a lot more of the features in the load balancer and the security devices from Citrix.

Huawei CloudEngine (CE)

There are over 65 **CE** modules that allow you to manage all aspects of these powerful switches from Huawei, including BGP, access control lists, MTU, static routes, VXLANs, and even SNMP configuration.

Arista CloudVision (CV)

There is a single module that allows you to provision an Arista **CV** server port using a configlet.

Lenovo CNOS

There are over 15 modules that allow you to manage devices running the **CNOS** operating system from Lenovo; they allow you to manage everything from BGP and port aggregation to VLAG, VLANs, and even factory reset devices.

Cumulus Linux (CL)

Of the eight **CL**, seven of them have been deprecated in favor of a single module that communicates with your Cumulus Linux-powered device using the **Network Command Line Utility (NCLU)**.

Dell operating system 10 (DellOS10)

DellOS10 has three modules, which allow you to execute commands, manage the configuration, and gather facts on devices running Dell's networking operating system. There are also modules for **Dell operating system 6 (DellOS6)** and **Dell operating system 9 (DellOS9)**.

Ubiquiti EdgeOS

There are modules for **EdgeOS** that allow you to manage the configuration, execute ad hoc commands, and gather facts on devices that are running EdgeOS, such as the Ubiquiti EdgeRouter.

Lenovo Enterprise Networking Operating System (ENOS)

There are three modules for the Lenovo **ENOS**. Like other devices, these allow you to gather facts, execute commands, and manage the configuration.

Arista EOS

There are sixteen modules that allow you to manage your devices running **EOS**. These modules let you configure interfaces, VLANs, VRFs, users, link aggregation, static routes, and even logging. There is also a module that allows you to gather facts from each device.

F5 BIG-IP

There are 65 modules, all prefixed with **BIG-IP**, that allow you to manage all aspects of your F5 BIG-IP Application Delivery Controller.

FortiGate FortiManager

There is a single module that allows you to add, edit, delete, and execute scripts against your FortiGate devices, using **FortiManager (fmgr)**.

FortiGate FortiOS

As part of the core network modules, there are three that enable you to manage the addresses, configuration, and IPv4 policy objects on your FortiGate **FortiOS**-powered devices.

illumos

illumos is a fork of the OpenSolaris operating system. It has several powerful networking features that make it the perfect candidate for deploying as a self-built router or firewall. There are three prefixes used: `d1adm`, `f1owadm`, and `i1padm`. These modules allow you to manage the interfaces, NetFlow, and tunnels. Also, as illumos is a fork of OpenSolaris, your playbook should work on OpenSolaris-based operating systems.

Cisco IOS and IOS XR

There are around 25 modules that allow you to manage your Cisco **IOS** and **IOS XR**-powered devices. With them, you can gather facts on your devices, as well as configure users, interfaces, logging, banners, and more.

Brocade IronWare

There are the usual three modules that assist you in managing your Brocade **IronWare**-powered devices; you can configure, run ad hoc commands, and gather facts.

Juniper Junos

There are 20 modules that enable you to interact with Juniper devices running **Junos** from within your playbooks. These range from the standard command, configuration, and fact-gathering modules to ones that allow you to install packages and copy files to your devices.

Nokia NetAct

There is a single module that allows you to upload and apply your Nokia **NetAct**-powered core and radio networks.

Pluribus Networks Netvisor OS

There are over ten modules that allow you to manage your **Pluribus Networks (PN)** Netvisor OS-powered devices, from creating clusters and routers, to running commands on your white-box switches.

Cisco Network Services Orchestrator (NSO)

There are a handful of modules that allow you to interact with your Cisco NSO-managed devices. You can execute NSO actions, query data from your installation, and verify your configuration alongside service synchronization and configuration.

Nokia Nuage Networks Virtualized Services Platform (VSP)

There is a single module that allows you to manage enterprises on your Nokia **Nuage** Networks VSP.

Cisco NX-OS (NXOS)

As you can imagine, there are a lot of modules for managing devices running Cisco **NXOS**—over 70. A few of them are in the process of being deprecated. With that many modules, you get quite large coverage of all the functions of this powerful network-operating system.

Mellanox ONYX

There over a dozen modules that allow you to interact with the **ONYX**, the switch operating system from Mellanox. You can manage BGP, L2 and L3 interfaces, and also LDAP.

Ordnance

There are two modules for the **Ordnance** Router as a Service; they allow you to apply configuration changes and also gather facts.

Open vSwitch (OVS)

There are three modules that allow you to manage bridges, ports, and databases on your **OVS** virtual switches.

Palo Alto Networks PAN-OS

There are over 20 modules that let you configure, manage, and audit your Palo Alto Networks devices running PAN-OS (**panos**). Currently, there are a few modules that are being deprecated; they will stop being distributed as a core module from Ansible 2.5.

Radware

A small number of modules, which have recently been introduced, allow you to manage your Radware devices via a **vDirect** server.

Nokia Networks Service Router Operating System (SROS)

There are three modules that allow you to run commands against, configure, and roll-back changes to your Nokia Networks **SROS** devices.

VyOS

There are a dozen or so modules that allow you to manage most aspects of your **VyOS** open source Linux-based router and firewall. We are going to take a look at VyOS in the next section.

System

There are also several generic **net** modules that allow you to manage interfaces, Layer2 and Layer3 configuration, NETCONF, routing, and also LLDP services on Linux-based network devices.

Interacting with a network device

As already mentioned at the start of the chapter, we are going to use Vagrant to launch a network device, and then run a playbook to apply a basic configuration. The device we will be launching is a VyOS. While the device will be a full VyOS installation, we will be applying only a test configuration to give you an idea of how the modules we mentioned in the previous section can be used.



There is a full copy of this playbook in the GitHub repository that accompanies this title.

Launching the network device

To do this, we are going to use a VyOS Vagrant box. If you are following along, we first need to create a folder called `vyos`. This will hold our playbook and `vagrantfile`. To create the folder structure and blank files needed for the playbook, run the following commands:

```
$ mkdir vyos vyos/group_vars vyos/roles  
$ ansible-galaxy init vyos/roles/vyos-firewall  
$ touch vyos/Vagrantfile  
$ touch vyos/production  
$ touch vyos/site.yml  
$ touch vyos/group_vars/common.yml  
$ touch vyos/roles/vyos-firewall/templates/firewall.j2
```

Copy the following code into the empty `Vagrantfile` we created:

```
# -*- mode: ruby -*-  
# vi: set ft=ruby :  
  
API_VERSION = "2"  
BOX_NAME = "russmckendrick/vyos"  
BOX_IP = "192.168.50.10"  
DOMAIN = "nip.io"  
PRIVATE_KEY = "~/.ssh/id_rsa"  
PUBLIC_KEY = '~/.ssh/id_rsa.pub'  
  
Vagrant.configure(API_VERSION) do |config|  
  config.vm.box = BOX_NAME  
  config.vm.network "private_network", ip: BOX_IP  
  config.vm.host_name = BOX_IP + '.' + DOMAIN  
  config.ssh.insert_key = false  
  config.ssh.private_key_path = [PRIVATE_KEY, "~/.vagrant.d/insecure_private_key"]  
  config.vm.provision "file", source: PUBLIC_KEY, destination: "~/.ssh/authorized_keys"  
  
  config.vm.provider "virtualbox" do |v|  
    v.memory = "2048"  
    v.cpus = "2"  
  end  
  
  config.vm.provider "vmware_fusion" do |v|  
    v.vmx["memsize"] = "2048"  
  
    v.vmx["numvcpus"] = "2"  
  end  
end
```

As you can see, the `vagrantfile` doesn't look that much different from the others we have been using in previous chapters. Let's now take a look at the `vyos_firewall` role. There are some differences in the approach to executing and

writing the role, which we should discuss before we launch anything.

The VyOS role

Before we go into the tasks, let's take a look at the variables we will be using. First of all, the content of `roles/vyos-firewall/defaults/main.yml`:

```
---
motd_asciiart: |
-----
VyOS Ansible Managed Firewall
-----
vyos_nameservers:
- 8.8.8.8
- 8.8.4.4
```

Here, we are setting just two key values. The first, `motd_asciiart`, is a multiline banner that will be configured to be displayed whenever we log in to the VyOS device. We are setting the variable as a multiline by using the `|` after declaring the key. The next key, `vyos_nameservers`, is a list of DNS resolvers to use. Here, we are using Google's Public DNS resolvers.

There are some other variables used in the playbook; these can be found in `group_vars/common.yml` as shown in this code:

```
---
vyos:
  host: "192.168.50.10.nip.io"
  username: "vagrant"
  backup: "yes"
  inside:
    interface: "172.16.20.1/24"
    subnet: "172.16.20.0/24"

whitelist_ips:
- 172.16.20.2

rules:
  - { action: 'set', source_address: '0.0.0.0/0', source_port: '80',
    destination_port: '80', destination_address: '172.16.20.11', protocol: 'tcp',
    description: 'NAT port 80 to 172.16.10.11', rule_number: '10' }
  - { action: 'set', source_address: '0.0.0.0/0', source_port: '443',
    destination_port: '443', destination_address: '172.16.20.11', protocol: 'tcp',
    description: 'NAT port 443 to 172.16.10.11', rule_number: '20' }
  - { action: 'set', source_address: '123.123.123.123/32', source_port: '222',
    destination_port: '22', destination_address: '172.16.20.11', protocol: 'tcp',
    description: 'NAT port 443 to 172.16.10.11', rule_number: '30' }
```

As you can see, these are the bulk of the variables that could change depending on where our playbook is being run. To start off with, we are setting the details of our device and its basic configuration in a nested variable called `vyos`. You might have noticed that we are passing the details of the IP address and username for our VyOS device in here, rather than our host inventory file.

In fact, our host inventory file, called `production`, should just contain the following line of code:

```
| localhost
```

What this means is that when our playbook is executed, it will not be executed against our VyOS device. Instead, the playbook will target our Ansible controller, and the module will then target the VyOS device. This approach is common among all the core network modules. As we have already discussed, Ansible is an agentless platform; it requires only an SSH or WinRM connection by default.

However, not every networking device has SSH or WinRM access; some may have only web-based APIs, while others may use a proprietary access method. Others, such as VyOS, may appear to have SSH access; however, you are SSHing into a custom shell that is designed to run only a few firewall commands. For this reason, most of the core network modules manage their connection and communication away from the host inventory file.

The remainder of the variables in the `group_vars/common.yml` file set up some basic firewall rules, which we will look at shortly.

The task for the role, which can be found at `roles/vyos-firewall/tasks/main.yml`, contains four parts. First of all, we are using the `vyos_config` module to set the hostname. Take a look at this code:

```
- name: set the hostname correctly
  vyos_config:
    provider:
      host: "{{ vyos.host }}"
      username: "{{ vyos.username }}"
    lines:
      - "set system host-name {{ vyos.host }}"
```

As you can see, we are passing the details of our VyOS device using the `provider` option; then we are passing a single `vyos` command to set the hostname. The

`vyos_config` module also accepts template files, which we will be using in a moment to fully configure our device.

The next task configures the DNS resolvers using the `vyos_system` module. Take a look at this code:

```
- name: configure name servers
  vyos_system:
    provider:
      host: "{{ vyos.host }}"
      username: "{{ vyos.username }}"
      name_server: "{{ item }}"
    with_items: "{{ vyos_nameservers }}"
```

Next up, we are going to set the **message of the day (MOTD)** using the `vyos_banner` module. Take a look at this code:

```
- name: configure the motd
  vyos_banner:
    provider:
      host: "{{ vyos.host }}"
      username: "{{ vyos.username }}"
    banner: "post-login"
    state: "present"
    text: "{{ motd_asciart }}"
```

Finally, we are going to apply our main firewall configuration using the following task:

```
- name: backup and load from file
  vyos_config:
    provider:
      host: "{{ vyos.host }}"
      username: "{{ vyos.username }}"
    src: "firewall.j2"
    backup: "{{ vyos.backup }}"
    save: "yes"
```

Rather than providing commands using `lines`, this time we are giving the name of a template file using `src`. We are also instructing the module to make a backup of the current configuration; this will be stored in the `roles/vyos-firewall/backup` folder, which is created when the playbook runs.

The template can be found at `roles/vyos-firewall/templates/firewall.j2`. This template contains the following code:

```
set firewall all-ping 'enable'
set firewall broadcast-ping 'disable'
set firewall ipv6-receive-redirects 'disable'
set firewall ipv6-src-route 'disable'
```

```

set firewall ip-src-route 'disable'
set firewall log-martians 'enable'
set firewall receive-redirects 'disable'
set firewall send-redirects 'enable'
set firewall source-validation 'disable'
set firewall state-policy established action 'accept'
set firewall state-policy related action 'accept'
set firewall syn-cookies 'enable'
set firewall name OUTSIDE-IN default-action 'drop'
set firewall name OUTSIDE-IN description 'deny traffic from internet'
{% for item in whitelist_ips %}
set firewall group address-group SSH-ACCESS address {{ item }}
{% endfor %}
set firewall name OUTSIDE-LOCAL rule 310 source group address-group SSH-ACCESS
set firewall name OUTSIDE-LOCAL default-action 'drop'
set firewall name OUTSIDE-LOCAL rule 310 action 'accept'
set firewall name OUTSIDE-LOCAL rule 310 destination port '22'
set firewall name OUTSIDE-LOCAL rule 310 protocol 'tcp'
set firewall name OUTSIDE-LOCAL rule 900 action 'accept'
set firewall name OUTSIDE-LOCAL rule 900 description 'allow icmp'
set firewall name OUTSIDE-LOCAL rule 900 protocol 'icmp'
set firewall receive-redirects 'disable'
set firewall send-redirects 'enable'
set firewall source-validation 'disable'
set firewall state-policy established action 'accept'
set firewall state-policy related action 'accept'
set firewall syn-cookies 'enable'
set interfaces ethernet eth0 firewall in name 'OUTSIDE-IN'
set interfaces ethernet eth0 firewall local name 'OUTSIDE-LOCAL'
set interfaces ethernet eth1 address '{{ vyos.inside.interface }}'
set interfaces ethernet eth1 description 'INSIDE'
set interfaces ethernet eth1 duplex 'auto'
set interfaces ethernet eth1 speed 'auto'
set nat source rule 100 outbound-interface 'eth0'
set nat source rule 100 source address '{{ vyos.inside.subnet }}'
set nat source rule 100 translation address 'masquerade'
{% for item in rules if item.action == "set" %}
{{ item.action }} nat destination rule {{ item.rule_number }} description '{{ item.description }}'
{{ item.action }} nat destination rule {{ item.rule_number }} destination port '{{ item.source_port }}'
{{ item.action }} nat destination rule {{ item.rule_number }} translation port '{{ item.destination_port }}'
{{ item.action }} nat destination rule {{ item.rule_number }} inbound-interface 'eth0'
{{ item.action }} nat destination rule {{ item.rule_number }} protocol '{{ item.protocol }}'
{{ item.action }} nat destination rule {{ item.rule_number }} translation address '{{ item.destination_address }}'
{{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }} action 'accept'
{{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }} source address '{{ item.source_address }}'
{{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }} destination address '{{ item.destination_address }}'
{{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }} destination port '{{ item.destination_port }}'
{{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }} protocol '{{ item.protocol }}'
{{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }} state new 'enable'
{% endfor %}
{% for item in rules if item.action == "delete" %}
{{ item.action }} nat destination rule {{ item.rule_number }}
{{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }}
{% endfor %}

```

There are a lot of commands in the template, the bulk of which are just applying some basic settings on the device. The ones we are interested in are the three `for` loops. The first loop is as follows:

```
|{% for item in whitelist_ips %}  
|  set firewall group address-group SSH-ACCESS address {{ item }}  
|{% endfor %}
```

This will simply loop through each of the IP addresses we have provided in the `whitelist_ips` variable, in a similar way to how we have been using `with_items` in previous playbooks. This is better demonstrated by the next loop, which takes the variables from the `firewall` variable and creates both the NAT and firewall rules. Take a look at this code:

```
|{% for item in rules if item.action == "set" %}  
|  {{ item.action }} nat destination rule {{ item.rule_number }} description '{{  
|    item.description }}'  
|  {{ item.action }} nat destination rule {{ item.rule_number }} destination port '{{  
|    item.source_port }}'  
|  {{ item.action }} nat destination rule {{ item.rule_number }} translation port '{{  
|    item.destination_port }}'  
|  {{ item.action }} nat destination rule {{ item.rule_number }} inbound-interface 'eth0'  
|  {{ item.action }} nat destination rule {{ item.rule_number }} protocol '{{  
|    item.protocol }}'  
|  {{ item.action }} nat destination rule {{ item.rule_number }} translation address '{{  
|    item.destination_address }}'  
|  {{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }} action 'accept'  
|  {{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }} source address  
|    '{{ item.source_address }}'  
|  {{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }} destination  
|    address '{{ item.destination_address }}'  
|  {{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }} destination port  
|    '{{ item.destination_port }}'  
|  {{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }} protocol '{{  
|    item.protocol }}'  
|  {{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }} state new  
|    'enable'  
|{% endfor %}
```

As you can see, the rule is included only if we have set `action` to `set` in the variable; the final loop takes care of any rules where we have set the `action` to `delete`, as shown in this code:

```
|{% for item in rules if item.action == "delete" %}  
|  {{ item.action }} nat destination rule {{ item.rule_number }}  
|  {{ item.action }} firewall name OUTSIDE-IN rule {{ item.rule_number }}  
|{% endfor %}
```

If you have been following along, you should have content in all of the files we initially created, apart from the `site.yml` file. This should contain the following code:

```
---
```

```
- hosts: localhost
  connection: local
  gather_facts: false

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/vyos-firewall
```

Now that we have all of the parts of our playbook together, we are able to launch the VyOS Vagrant box and run the playbook.

Running the playbook

To launch the Vagrant box, make sure you are in the `vyos` folder we created at the start of this section and run one of the following two commands to start the box with your chosen hypervisor: `$ vagrant up`
`$ vagrant up --provider=vmware_fusion`

Once your Vagrant box has launched, you can run the playbook using the following command:

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

The output for this playbook run should look something like the following:

```
PLAY [localhost]
*****
TASK [roles/vyos-firewall : set the hostname correctly]
*****
changed: [localhost]

TASK [roles/vyos-firewall : configure name servers]
*****
changed: [localhost] => (item=8.8.8)
changed: [localhost] => (item=8.8.4.4)

TASK [roles/vyos-firewall : configure the motd]
*****
changed: [localhost]

TASK [roles/vyos-firewall : backup and load from file]
*****
changed: [localhost]

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

Once complete, you should be able to SSH into your VyOS device by running the following code:

```
| $ vagrant ssh
```

You should see that the login banner has been updated to the one we defined, as shown in the following screenshot:

```
2. vagrant@192: ~ (ssh)
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter07/vyos on master*
⚡ vagrant ssh
-----
VyOS Ansible Managed Firewall
-----
Last login: Mon Mar 26 15:41:49 2018 from 172.16.20.2
vagrant@192:~$
```

While logged in, you should be able to see the VyOS configuration by running the following command:

```
| $ show config
```

You should be able to spot all of the changes we have made with the playbook run, as shown in the next screenshot:

```
2. vagrant@192: ~ (ssh)
firewall {
    all-ping enable
    broadcast-ping disable
    group {
        address-group SSH-ACCESS {
            address 172.16.20.2
        }
    }
    ipv6-receive-redirects disable
    ipv6-src-route disable
    ip-src-route disable
    log-martians enable
    name OUTSIDE-IN {
        default-action drop
        description "deny traffic from internet"
        rule 10 {
            action accept
            destination {
                address 172.16.20.11
                port 80
            }
            protocol tcp
    }
}
```

To stop viewing the configuration, press **Q**. You can type `exit` to leave the SSH session. You can remove the VyOS Vagrant box by running the following: \$

vagrant destroy

As mentioned at the start of this section of the chapter, this exercise has not been about configuring a fully functional VyOS installation using Ansible; instead, it gives a practical example of how you could potentially go about configuring a network device using Ansible modules, which both effect change and also apply configuration using a template.

Summary

In this chapter, we have taken a brief look at the various network modules provided as part of the Ansible core set of modules. We also applied a configuration to a virtual VyOS device to get a feel for how the network modules differ from the modules we have covered in previous chapters.

In the next chapter, we are going to look at launching cloud-based server instances using Ansible and then applying some of our playbooks to them.

Questions

1. True or False: You have to use `with_items` with a `for` loop within a template.
2. Which character is used to split your variable over multiple lines?
3. True or False: When using the VyOS module, we do not need to pass details of our device in the host inventory file.
4. Can you roll back your VyOS configuration to the earliest backup you have stored?

Further reading

There are details on each of the devices and technologies currently supported by the core network modules at each of the following links:

- **A10 Networks:** <https://www.a10networks.com/>
- **Cisco ACI:** https://www.cisco.com/c/en_uk/solutions/data-center-virtualization/application-centric-infrastructure/index.html
- **Cisco AireOS:** <https://www.cisco.com/c/en/us/products/wireless/wireless-lan-controller/index.html>
- **AOS:** <http://www.apstra.com/products-aos/>
- **Aruba Mobility Controller:** <http://www.arubanetworks.com/en-gb/products/networking/controllers/>
- **Cisco ASA:** <https://www.cisco.com/c/en/us/products/security/adaptive-security-appliance-asa-software/index.html>
- **Avi Networks:** <https://avinetworks.com/>
- **Big Switch Networks:** <https://www.bigswitch.com>
- **Citrix Netscaler:** <https://www.citrix.com/products/netscaler-adc/>
- **Huawei CloudEngine:** <http://e.huawei.com/uk/products/enterprise-networking-switches/data-center-switches>
- **Arista CloudVision:** <https://www.arista.com/en/products/eos/eos-cloudvision>
- **Lenovo CNOS and ENOS:** <https://www3.lenovo.com/gb/en/data-center/networking/-software/c/networking-software/>
- **Cumulus Linux:** <https://cumulusnetworks.com/products/cumulus-linux/>
- **Dell operating system 10:** <http://www.dell.com/en-us/work/shop/povw/open-platform-software/>
- **Ubiquiti EdgeOS:** <https://www.ubnt.com/edgemax/edgerouter/>
- **Arista EOS:** <https://www.arista.com/en/products/eos>
- **F5 BIG-IP:** <https://f5.com/products/big-ip>
- **FortiGate FortiManager:** <https://www.fortinet.com/products/management/fortimanager.html>
- **FortiGate FortiOS:** <https://www.fortinet.com/products/fortigate/fortios.html>
- **illumos:** <http://www.illumos.org/>
- **Cisco IOS:** <https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-software->

[releases-listing.html](#)

- **Cisco IOS XR:** <https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-xr-software/index.html>
- **Brocade IronWare:** <https://www.broadcom.com/>
- **Juniper Junos:** <https://www.juniper.net/uk/en/products-services/nos/junos/>
- **Nokia NetAct:** <https://networks.nokia.com/solutions/netact>
- **Pluribus Networks Netvisor OS:** <https://www.pluribusnetworks.com/products/white-box-os/>
- **Cisco NSO:** <https://www.cisco.com/c/en/us/solutions/service-provider/solutions-cloud-providers/network-services-orchestrator-solutions.html>
- **Nokia Nuage Networks VSP:** <http://www.nuagenetworks.net/products/virtualized-services-platform/>
- **Cisco NX-OS:** <https://www.cisco.com/c/en/us/products/ios-nx-os-software/nx-os/index.html>
- **Mellanox ONYX:** http://www.mellanox.com/page/mlnx_onyx?mtag=onyx_software
- **Ordnance:** <https://ordnance.co/>
- **Open vSwitch:** <https://www.openvswitch.org/>
- **Palo Alto Networks PAN-OS:** <https://www.paloaltonetworks.com/documentation/80/pan-os>
- **Radware:** <https://www.radware.com>
- **Nokia Networks Service Router Operating System:** <https://networks.nokia.com/products/sros>
- **VyOS:** <https://vyos.io/>

Moving to the Cloud

In this chapter, we will move from using our local virtual machine to using Ansible to launch instances in a public cloud provider. For this chapter, we will be using DigitalOcean, and we are targeting this provider as it allows us to simply launch virtual machines and interact with them, without having too much configuration overhead.

We will then look at adapting our WordPress playbook so that it interacts with the newly launched instance.

In this chapter, we will cover the following topics:

- A quick introduction to DigitalOcean
- Launching instances in DigitalOcean
- How to switch between running Ansible locally and remotely so we can deploy WordPress

Technical requirements

In this chapter, we are going to be launching instances in a public cloud, so if you are following along you will need an account with DigitalOcean. As with other chapters, complete versions of the playbooks can be found in the repository in the `Chapter08` folder at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter08>.

Interacting with DigitalOcean

DigitalOcean was founded in 2011 and has grown from a typical virtual private server host with a single data center to a developer-friendly cloud service provider with data centers around the world. Netcraft noted that in December 2012, DigitalOcean hosted around 100 web-facing servers; in March 2018 that number exceeds 400,000, making DigitalOcean the third largest host of web-facing instances.

What makes DigitalOcean popular among developers, apart from its prices, is its performance; DigitalOcean was one of the first hosting companies to offer all **solid-state drives (SSD)** for its instance storage. It is simple-to-use web-based control panel, alongside the ability to launch instances from its command-line interface, and also a powerful API, which allows you to launch instances (which DigitalOcean calls Droplets) from within your applications, and also tools such as Ansible.

You can sign up for an account at <https://www.digitalocean.com/>. Once you have signed up, the first thing that I recommend you do before proceeding with anything else is to configure two-factor authentication on your account.



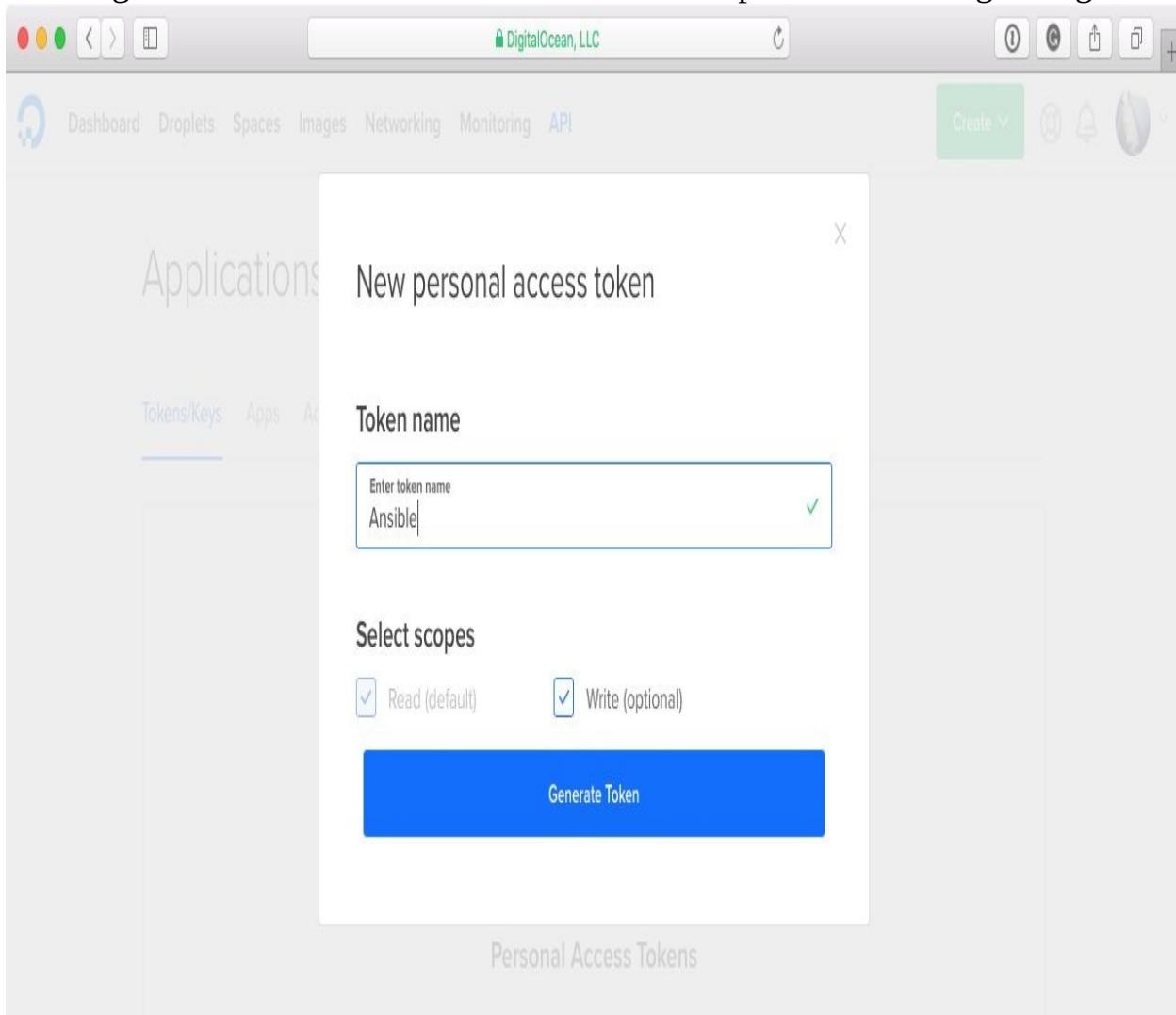
Two-factor authentication (2FA) or multi-factor authentication (MFA) adds an additional level of authentication to your account. Typically, this is achieved by sending a code via SMS to a device that has been associated with your account or by linking the account to a third-party authentication application such as Google or Microsoft Authenticator, which is running on your smartphone. Accounts that are tied to one of these services require you to typically enter a six-digit number, which is rotated every 30 seconds.

You can configure 2FA by going to your settings in the DigitalOcean control panel and then clicking on Security, which can be found in the left-hand-side menu; once there, follow the onscreen instructions to enable 2FA on your account.

Generating a personal access token

For our playbook to be able to be able to launch a Droplet in our DigitalOcean account, we will need to generate a personal access token to interact with the DigitalOcean API. To do this, click on the API link, which can be found in the menu at the top of the DigitalOcean web-based control panel.

Clicking on the Generate New Token button will open the following dialog box:



As you can see, I have named my token `Ansible` so it can be easily identified. Clicking the Generate Token button will create a token; it will only be displayed once so please make sure you make a note of it.



Anyone who has a copy of your personal access token will be able to launch resources in your DigitalOcean account; please ensure that you keep it safe and do not publish your token anywhere.

I have blurred out my token in the following screenshot, but it should give you an idea of what you will see after generating your personal access token:

The screenshot shows a web browser window with the DigitalOcean logo at the top. Below the header, there's a toolbar with various icons. The main content area is titled "Personal access tokens" and displays a table of generated tokens. A "Generate New Token" button is visible in the top right corner of the table area. The table has columns for "Name", "Scope", and "Created". One row is shown, named "Ansible", with "Scope" set to "READ WRITE" and "Created" as "Just now". The token itself is blurred out. At the bottom of the table, there's a note: "Personal access tokens function like a combined name and password for API authentication."

Name	Scope	Created	More
Ansible	READ WRITE	Just now	More ▾

While we now have our token, there is one more thing we need to configure before we start our playbook.

Installing dopy

One of the modules we will be using requires a Python module called `dopy`; it acts a wrapper for the DigitalOcean API and can be installed using the following `pip` command: **\$ sudo pip install dopy**

Once `dopy` is installed, we can start writing the playbook.

Launching a Droplet

As per previous playbooks we have written, you can create the skeleton structure by running the following commands:

```
$ mkdir digitalocean digitalocean/group_vars digitalocean/roles  
$ ansible-galaxy init digitalocean/roles/droplet  
$ touch digitalocean/production digitalocean/site.yml  
digitalocean/group_vars/common.yml
```

There are two tasks we need to complete to launch our Droplet; the first is that we need to make sure that a copy of our public SSH key is uploaded to DigitalOcean so that we can inject it into our Droplet launched during the second task.

Before we proceed with looking at the role which launches the Droplet we should figure what we are going to do with the personal access token required by the playbook to access the DigitalOcean API. For this we are going to use Ansible Vault to encode just the token; run the following command, making sure you replace the contents of `encrypt_string` with your own token:

```
ansible-vault \  
  encrypt_string 'pLgVbM2hsWiLFWbemyD4Nru3a2yYwAKm2xbL6WmPBtzqvnMTrVTXYuabWbp7vArQ' \  
  --name 'do_token'
```

 *The tokens used throughout these chapters are randomly generated; please make sure you replace them with your own.*

The following screenshot shows the output for the preceding commands:

```

1. digitalocean (bash)
russ in ~/digitalocean
⚡ ansible-vault \
→ encrypt_string 'pLgVbM2hswiLFWbemyD4Nru3a2yYwAKm2xbL6WmPBtzqvnMTrVTXYuabWbp7vArQ' \
→ --name 'do_token'
New Vault password:
Confirm New Vault password:
do_token: !vault |
$ANSIBLE_VAULT;1.1;AES256
61343630316236306532316531633239636661383630323030303861346363376665623762343031
6334316635333833373362303839646262376136343836330a64353665336439353836661633139
3732663334663633396333613531353432613834646133316330633033643230643938333663137
6337356363323064660a386264643131666132633835383630323738356662653163343436396134
33356638326137366332343162656639623632646337633366366436363439643933653062303539
633939643436363536646336356364636331613832663730393063666132613135626539613533
65653233343335663265646663646437363639663932363432303531383831333837383031343434
65336163643635623038
Encryption successful
russ in ~/digitalocean
⚡

```

As you can see, this returns the encrypted token, so place the encrypted token in the `group_vars/common.yml` file. While we are populating variables, let's take a look at what the content of `roles/droplet/defaults/main.yml` should look like:

```

---
# defaults file for digitalocean/roles/droplet

key:
  name: "Ansible"
  path: "~/.ssh/id_rsa.pub"

droplet:
  name: "AnsibleDroplet"
  region: "lon1"
  size: "s-1vcpu-2gb"
  image: "centos-7-x64"
  timeout: "500"

```

There are two collections of key values; the first deals with the SSH key, which the playbook will be uploading, and the second contains the information for launching the Droplet. The defaults for our initial playbook run will launch a 1-CPU core, 2 GB RAM, 50-GB HDD CentOS 7 Droplet in the DigitalOcean London data center.

The tasks for the launching the Droplet, which should be in `roles/droplet/tasks/main.yml`, contain two separate parts; the first part deals with uploading the SSH key, which needs to be there so we can launch the Droplet

using it:

```
- name: "upload SSH key to DigitalOcean"
  digital_ocean_sshkey:
    oauth_token: "{{ do_token }}"
    name: "{{ key.name }}"
    ssh_pub_key: "{{ item }}"
    state: present
  with_file: "{{ key.path }}"
```

As you can see, this task uses the token we encrypted with Ansible Vault; we are also using the `with_file` directive to copy the contents of the key file, which is at `~/.ssh/id_rsa.pub`. This task will do one of three things depending on what you already have in your DigitalOcean account:

- If the key does not exist, it will upload it
- If a key matches the fingerprint of `~/.ssh/id_rsa.pub` but has a different name, then it will rename that key
- If the key and name match, nothing will be uploaded or changed

Now that we know that we have our key uploaded, we need to know its unique ID. To find this out, we should gather facts on all of the keys that are configured in our DigitalOcean account by running the following task:

```
- name: "gather facts on all of the SSH keys in DigitalOcean"
  digital_ocean_sshkey_facts:
    oauth_token: "{{ do_token }}"
```

This will return a JSON array named `ssh_keys` which contains the name of the key, the fingerprint of the key, the contents of the key itself, and also the key's unique ID; this information is returned for each of the keys that are configured in our DigitalOcean account. As we need to know the ID of just one of those keys, we need to manipulate the results to filter the list down to just the single key we uploaded, and then set the ID as a variable.

As we know, we have a JSON array of potential keys stored in the `ssh_keys` value; for me, this looked like the following:

```
{
  "fingerprint": "9e:ad:42:e9:86:01:3c:5f:de:11:60:11:e0:11:9e:11",
  "id": 2663259,
  "name": "Work",
  "public_key": "ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEAvg2cUTYCHnGcwHYjVh3vu09T6UwLEyXEKDnv3039KStLpQV3H7Pvh0IpAbY7C
russ@work"
},
```

```
{
  "fingerprint": "7d:ce:56:5f:af:45:71:ab:af:fe:77:c2:9f:90:bc:cf",
  "id": 19646265,
  "name": "Ansible",
  "public_key": "ssh-rsa
AAAAB3NzaC1yc2EAAQABAAQDmuoFR01i/Yf3HATl9c3sufJvghTFgYzK/Zt29JiTqW1SQhmXhNNTh6iI6
russ@mckendrick.io"
}
```

You may have noticed I have already executed the playbook and uploaded my key so that I can walk through this task with you. We now need to find the key called `key.name`, which in our case is `Ansible`, and then return just the ID. To do this, we are going to add the following task:

```
- name: "set the SSH key ID as a fact"
  set_fact:
    pubkey: "{{ item.id }}"
  with_items: "{{ ssh_keys | json_query(key_query) }}"
  vars:
    key_query: "[?name=='{{ key.name }}']"
```

As you can see, we are using the `set_fact` module to create a key-value pair called `pubkey`; we are using the ID of an item, and to make sure we return just one item, we are applying a JSON query to our array. This query makes sure that only the JSON that contains the `key.name` is returned in the `with_items` list; from here we can take the `id` of the single item, which allows us to move on to the second part, which is launching the Droplet.

Now that we know the ID of the SSH key we want to launch our Droplet with, we can move on to the second part of the role. The following task launches the Droplet:

```
- name: "launch the droplet"
  digital_ocean:
    state: "present"
    command: "droplet"
    name: "{{ droplet.name }}"
    unique_name: "yes"
    api_token: "{{ do_token }}"
    size_id: "{{ droplet.size }}"
    region_id: "{{ droplet.region }}"
    image_id: "{{ droplet.image }}"
    ssh_key_ids: [ "{{ pubkey }}" ]
    wait_timeout: "{{ droplet.timeout }}"
  register: droplet
```

This uses the `digital_ocean` module to launch the Droplet. Most of the items are self-explanatory; however, there is one important option we have to set a value for and that is `unique_name`. By default, `unique_name` is set to `no`, and what this means

is that if we were to run our playbook a second time, a new Droplet with exactly the same details as the first Droplet we launched would be created; run it a third time and a third Droplet will be created. Setting `unique_name` to `yes` will mean that only a single Droplet with whatever the value of `droplet.name` is active at one time.

As you can see, we are registering the output of the task as a value. Some details about the Droplet are returned as part of the task execution; the IP address of the Droplet is one of them, so we can use this to set a fact and then print a message with the IP address:

```
- name: "set the droplet IP address as a fact"
  set_fact:
    droplet_ip: "{{ droplet.droplet.ip_address }}"
- name: "print the IP address of the droplet"
  debug:
    msg: "The IP of the droplet is {{ droplet_ip }}"
```

That completes the basic playbook and once we have updated the `site.yml` file, we can run it. This should contain the following:

```
---
- hosts: localhost
  connection: local
  gather_facts: false

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/droplet
```

As you can see, we are just using `localhost` so there is no need to call the host inventory file.

Running the playbook

As we have a value that is encrypted using the Vault, we need to run the following command to run the playbook:

```
| $ ansible-playbook --vault-id @prompt site.yml
```

This will prompt for the passphrase you set to encrypt the Vault. Once you have entered the passphrase, the play will run:

```
PLAY [localhost]
*****
TASK [roles/droplet : upload SSH key to DigitalOcean]
*****
changed: [localhost] => (item=ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQDmuoFR01i/Yf3HATl9c3sufJvghTFgYzK/Zt29JiTqW1SQhmXhNNTh6iIE
russ@mckendrick.io)

TASK [roles/droplet : gather facts on all of the SSH keys in DigitalOcean]
*****
ok: [localhost]

TASK [roles/droplet : set the SSH key ID as a fact]
*****
ok: [localhost] => (item={'public_key': 'ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQDmuoFR01i/Yf3HATl9c3sufJvghTFgYzK/Zt29JiTqW1SQhmXhNNTh6iIE
russ@mckendrick.io', 'fingerprint':
'u'7d:ce:5f:af:45:71:ab:af:fe:77:c2:9f:90:bc:cf', 'id': 19646265, 'name':
'Ansible'})
```

```
TASK [roles/droplet : launch the droplet]
*****
changed: [localhost]
```

```
TASK [roles/droplet : set the droplet IP address as a fact]
*****
ok: [localhost]
```

```
TASK [roles/droplet : print the IP address of the droplet]
*****
ok: [localhost] => {
    "msg": "The IP of the droplet is 159.65.27.87"
}
```

```
PLAY RECAP
*****
localhost : ok=6 changed=2 unreachable=0 failed=0
```

As you can see, this uploaded my key and launched a Droplet that has an IP address of 159.65.27.87 (this IP is no longer in use by this Droplet). This is reflected in the DigitalOcean control panel, where we can see the key that has

been added:

The screenshot shows the 'Two-factor authentication' section of the DigitalOcean account settings. It displays two methods: 'App' (Default method) and 'SMS'. Both are shown as 'Enabled'. Below this is the 'SSH keys' section, which lists two keys: 'Ansible' and 'Work'. Each key has a 'Name', 'Fingerprint', and a 'More' dropdown.

Name	Fingerprint	More
Ansible	7d:ce:56:5f:af:45:71:ab:af:fe:77:c2:9f:90:bc:cf	More
Work	[REDACTED]	More

You can also see the Droplet on the Droplets page:

The screenshot shows the 'Droplets' page. At the top, there's a search bar labeled 'Search by Droplet name'. Below it, a navigation bar includes 'Dashboard', 'Droplets' (which is active), 'Spaces', 'Images', 'Networking', 'Monitoring', and 'API'. On the right, there are 'Create' and user profile icons. The main table lists one droplet:

Name	IP Address	Created	Tags
AnsibleDroplet	159.65.27.87	2 minutes ago	More

Also, you can SSH into the Droplet using the `root` username:



1. digitalocean (bash)

```
russ in ~/digitalocean
⚡ ssh root@159.65.27.87
The authenticity of host '159.65.27.87 (159.65.27.87)' can't be established.
ECDSA key fingerprint is SHA256:khZ2jieAs3PpU7ygXkzqj41TLbCfKflLtybXp4mGtD4.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '159.65.27.87' (ECDSA) to the list of known hosts.
[root@AnsibleDroplet ~]# cat /etc/*release
CentOS Linux release 7.4.1708 (Core)
NAME="CentOS Linux"
VERSION="7 (Core)"
ID="centos"
ID_LIKE="rhel fedora"
VERSION_ID="7"
PRETTY_NAME="CentOS Linux 7 (Core)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:centos:centos:7"
HOME_URL="https://www.centos.org/"
BUG_REPORT_URL="https://bugs.centos.org/"
```

As you can see, launching and interacting with DigitalOcean is relatively simple. Before we progress to the next section, destroy your instance from within the DigitalOcean control panel.

WordPress on DigitalOcean

Now we have a playbook that launches a Droplet, we are going to make a slight adaptation and install WordPress on the Droplet we launch. To do this, make a copy of the folder that holds the playbook we just ran and call it `digitalocean-wordpress`. Copy the three roles, `stack-install`, `stack-config`, and `wordpress`, from the `Chapter06/lemp-multi/roles` folder.

The host inventory

The first file we are going to change is the host inventory file called production; this needs to be updated to be the following: [droplets]

```
[digitalocean:children]
droplets
```

```
[digitalocean:vars]
ansible_ssh_user=root
ansible_ssh_private_key_file=~/ssh/id_rsa
host_key_checking=False
ansible_python_interpreter=/usr/bin/python
```

What we have here is an empty host group called `droplets`, and then we set some global variables for the Droplet we are launching. Don't worry about adding the actual host just yet; we will be adding it during the playbook run.

Variables

We are going to be overriding a few of the default variables. To do this, update the `group_vars/common.yml` file to read like this, making sure you update the `do_token` value to reflect your own:

```
do_token: !vault |
    $ANSIBLE_VAULT;1.1;AES256
63376236316336633631353131313363666463363834524609643522613230653265373236353664
3665376337396131343337313863393366393945225734573336238353862383432373831393839
323162626539633338366133323666393303939306634373930306666326232337613937623533
3461626330663363330a303538393836613835313166383030636134623530323932303266373134
35616339376138636530346632345734563457326532376233323930383535303563323634336162
3138663564663636333439364383633346636616664386539393162333062343964326561343861
33613265616632656465643664376536653334653532336335306230363834523454245337626631
    3323730636562616631

droplet:
  name: "WordPress"
  region: "lon1"
  size: "s-1vcpu-2gb"
  image: "centos-7-x64"
  timeout: "500"

wordpress:
  domain: "http://{{ hostvars['localhost'].droplet_ip }}/"
  title: "WordPress installed by Ansible on {{ os_family }} host in DigitalOcean"
  username: "ansible"
  password: "AnsiblePasswordForDigitalOcean"
  email: "test@example.com"
  theme: "sydney"
  plugins:
    - "jetpack"
    - "wp-super-cache"
    - "wordpress-seo"
    - "wordfence"
    - "nginx-helper"
```

As you can see, the majority of the values are their default values; the four values we are changing are:

- `droplet.name`: This is a simple update to the name so we can easily spot our

instance in the DigitalOcean control panel.

- `wordpress.domain`: This is the important change here. As you can see, we are using the `droplet_ip` variable we set on our Ansible controller. To make the variable available to our WordPress host, we are telling Ansible to use the variable from localhost. If we hadn't done this, then the variable would not have been set; we will look at why in the next section.
- `wordpress.title`: A slight tweak to the title our WordPress site is configured to reflect where it is hosted.
- `wordpress.password`: Changing the password so it is more complex, as we are launching on a publicly available IP address.

The playbook

The next file we are going to change is the `site.yml` one. This file needs to be updated to run the roles both locally and also against the Droplet we have launched:

```
---
- name: Launch the droplet in DigitalOcean
  hosts: localhost
  connection: local
  gather_facts: True

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/droplet

- name: Install WordPress on the droplet
  hosts: digitalocean
  gather_facts: true

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/stack-install
    - roles/stack-config
    - roles/wordpress
```

Our updated `site.yml` file contains two different plays: the first one runs on our Ansible controller and interacts with the DigitalOcean API to launch a Droplet, the second play then connects to the hosts in the `digitalocean` group to install WordPress. So how does Ansible know the IP address of the host to connect to?

The droplet role

We need to make one change, the `droplet` role, which can be found at `roles/droplet/tasks/main.yml`; this change takes the dynamically assigned IP address and adds it to our `droplets` group of hosts. To do this, replace the following task:

```
- name: "print the IP address of the droplet"
  debug:
    msg: "The IP of the droplet is {{ droplet_ip }}"
```

Replace it with the following task:

```
- name: add our droplet to a host group for use in the next step
  add_host:
    name: "{{ droplet_ip }}"
    ansible_ssh_host: "{{ droplet_ip }}"
    groups: "droplets"
```

As you can see, this takes the `droplet_ip` variable and adds a host using the `add_host` module to the group.

Running the playbook

Now that we have all of the pieces of the playbook together, we can launch a Droplet and install WordPress by running: **\$ ansible-playbook -i production --vault-id @prompt site.yml**

It will take a little while to launch the Droplet and perform the installation; at the end of it you should have the IP address listed in the play overview as the IP address is used as the name of our Droplet host. Here is the end of my playbook run: **RUNNING HANDLER [roles/stack-config : restart nginx]**

```
*****
changed: [165.227.228.104]
```

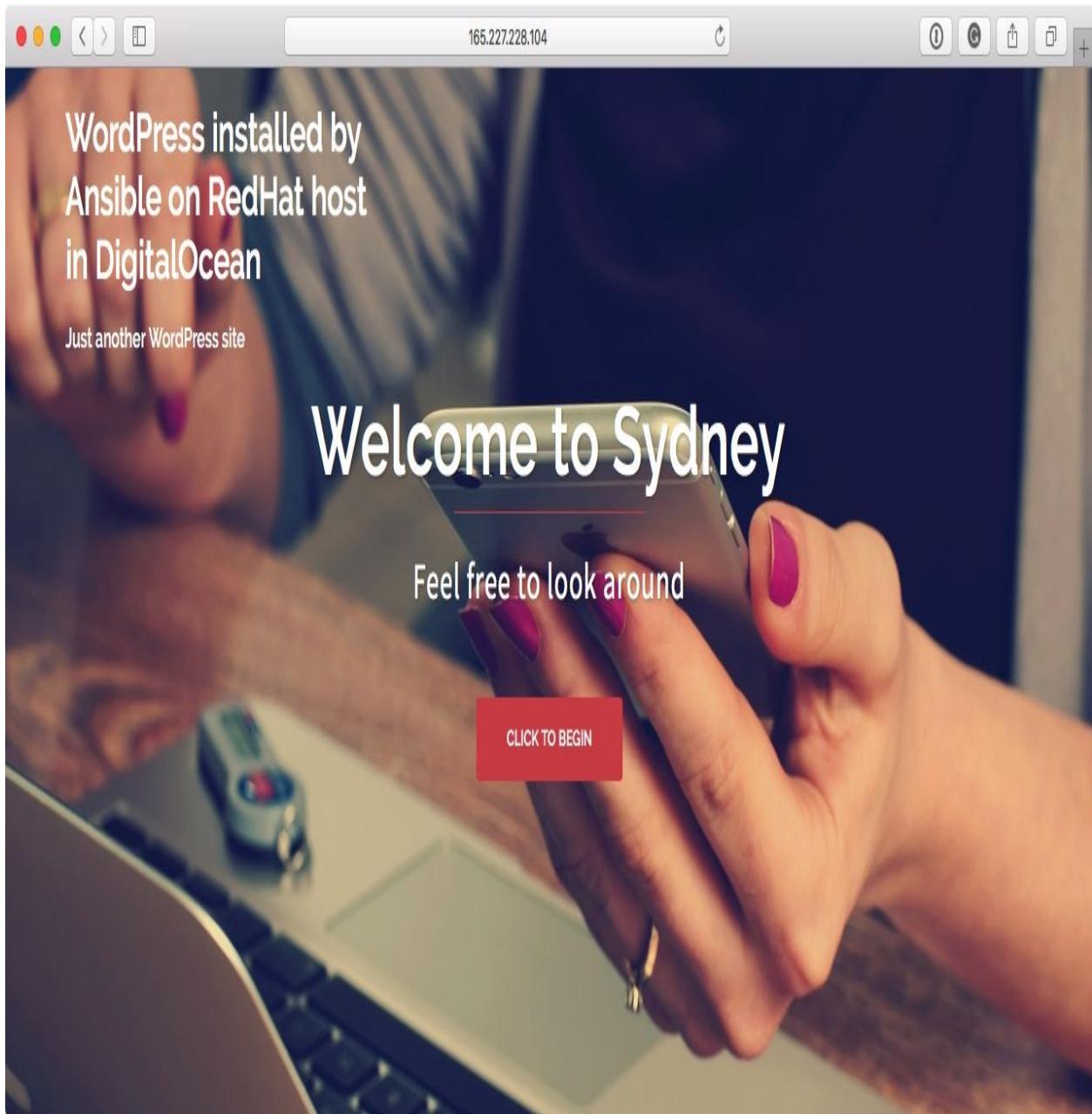
RUNNING HANDLER [roles/stack-config : restart php-fpm]

```
*****
changed: [165.227.228.104]
```

PLAY RECAP

```
*****
165.227.228.104 : ok=47 changed=37 unreachable=0 failed=0
localhost : ok=7 changed=1 unreachable=0 failed=0
```

Going to the IP address in your browser should present you with something like the following page:



You should be able to log in using the new password we set in the `common.yml` file. Have a play with the WordPress installation; when you are ready, destroy the Droplet from within the DigitalOcean control panel. But remember: leaving the Droplet running will incur a cost.

Summary

In this chapter, we launched our first instances in a public cloud using one of the Ansible cloud modules; as you have seen, the process was relatively straightforward and we managed to launch compute resource in the cloud and then install WordPress on it, without making any changes to the roles we covered in [Chapter 5, Deploying WordPress](#).

In the next chapter, we are going to expand on some of the techniques we have covered in this chapter and return to networking, but unlike the last chapter, where we covered networking devices, we will be looking at networking in public clouds.

Questions

1. What is the name of the Python module we need to install to support the `digital_ocean` module?
2. True or false: You should always encrypt sensitive values such as the DigitalOcean personal access token.
3. Which filter are we using to find the ID of the SSH key we need to launch our Droplet with?
4. State and explain why we used the `unique_name` option in the `digital_ocean` task.
5. What is the correct syntax for accessing variables from another Ansible host?
6. True or false: The `add_server` module is used to add our Droplet to the host group.
7. Try launching an Ubuntu Droplet and install WordPress on it; the image ID to use is `ubuntu-16-04-x64` and don't forget to change the `ansible_python_interpreter` value.

Further reading

You can read more details on the Netcraft statistics on DigitalOcean at <http://trends.netcraft.com/www.digitalocean.com/>.

Building Out a Cloud Network

Now that we have launched servers in DigitalOcean, we will move on to starting to look at launching services within **Amazon Web Services (AWS)**.

Before we launch instances, we will need to create a network for them to be hosted in. This is called a VPC, and there are a few different elements we will need to bring together in a playbook to create one, which we will then be able to use for our instances.

In this chapter, we will:

- Get an introduction to AWS
- Cover what it is we are trying to achieve and why
- Create a VPC, subnets, and routes—networking and routing
- Create security groups—firewall
- Create an **Elastic Load Balancing (ELB)**—load balancer

Technical requirements

In this chapter, we are going to use AWS; you will need administrator access to be able to create the roles needed to allow Ansible to interact with your account. As with other chapters, you can find the complete playbooks in the `chapter09` folder in the accompanying GitHub repository at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter09/vpc>.

An introduction to AWS

AWS has been around since 2002; it started by offering a few services which were not linked in any way—it progressed in this form until early 2006 when it was relaunched. The relaunched AWS brought together three services:

- **Amazon Elastic Compute Cloud (Amazon EC2):** This is the AWS compute service
- **Amazon Simple Storage Service (Amazon S3):** Amazon's scalable object storage accessible service
- **Amazon Simple Queue Service (Amazon SQS):** This service provides a message queue, primarily for web applications

Since 2006 it has grown from three unique services to over 160, covering over 15 primary areas such as:

- Compute
- Storage
- Database
- Networking and content delivery
- Machine learning
- Analytics
- Security, identity, and compliance
- Internet of Things

At its earnings call in February 2018, it was revealed that AWS had \$17.46 billion in revenue in 2017 which accounted for 10% of Amazon's total revenue; not bad for a service which originally offered to share idle compute time.

At the time of writing, AWS spans 18 geographic regions, which host a total of 54 availability zones: <https://aws.amazon.com/about-aws/global-infrastructure/>.

So what makes AWS so successful? Not only its coverage, but its approach to putting out its services. Andy Jassy, AWS CEO, has been quoted as saying: "Our mission is to enable any developer or any company to be able to build all their technology applications on top of our infrastructure technology platform."

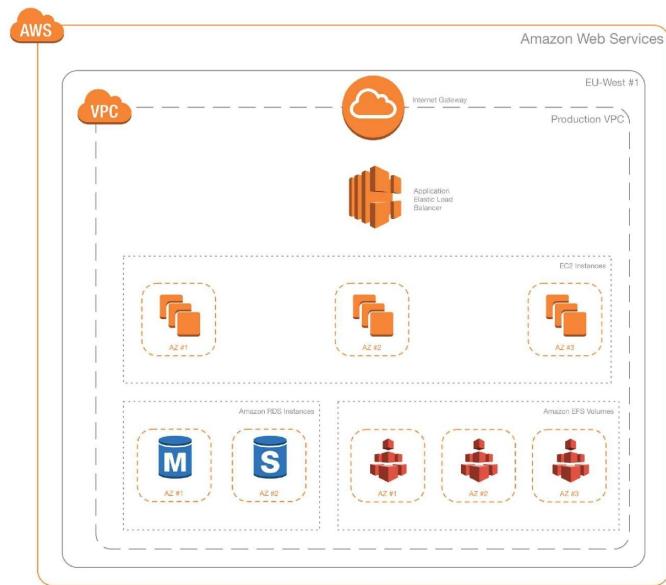
As an individual, you have access to the same APIs, service, regions, tools, and also pricing models as large multi-national companies and also Amazon themselves as they consume their services. This really gives you the freedom to start small and scale massively. For example, Amazon EC2 instances start from around \$4.50 per month for a t2.nano (1 vCPU, 0.5G) all the way up to over \$19,000 per month for an x1e.32xlarge (128 vCPU, 3,904 GB RAM, two 1920 GB SSD storage)—as you can see, there are instance types for every workload imaginable.

Both of these instances and most services are billed as pay-as-you-go modules, from per-second billing for EC2 instances or pay per GB per month for the storage you are using.

Amazon Virtual Private Cloud overview

In this chapter, we are going to be concentrating on launching an **Amazon Virtual Private Cloud (Amazon VPC)**; this is the networking layer which will house the compute and other Amazon services we will be launching in the next chapter.

An overview of the VPC our playbook is going to launch is as follows:



As you can see, we are going to be launching our VPC into the **EU-West #1** (Ireland) region; we will be spanning all three availability zones for our EC2 instances and also the **Application Elastic Load Balancer**. We will only be using two availability zones for our Amazon **Relational Database Service (RDS)** instance and also two zones for the **Amazon Elastic File System (Amazon EFS)** volumes.

This all means our Ansible playbook needs to create/configure the following:

- One Amazon VPC
- Three subnets for EC2 instances

- Two subnets for Amazon RDS instances
- Two subnets for Amazon EFS volumes
- Three subnets for the Application Load Balancer
- One internet gateway

We will also need to configure the following:

- One route to allow access through the internet gateway
- One security group which allows everyone to access to port 80 (HTTP) and 443 (HTTPS) on the Application Load Balancer
- One security group which allows trusted source access to port 22 (SSH) on the EC2 instances
- One security group which allows access to port 80 (HTTP) from the Application Load Balancer to the EC2 instances
- One security group which allows access to port 3306 (MySQL) on the Amazon RDS instances from the EC2 instances
- One security group which allows access to port 2049 (NFS) on the Amazon EFS volumes from the EC2 instances

This will give us our basic network, allowing restrictive access to everything but the Application Load Balancer which we want to be publicly available. Before we start creating an Ansible playbook which deploys the network, we need to get an AWS API access key and secret.

Creating an access key and secret

It is more than possible to create an access key and secret key for your own AWS user that would give Ansible full access to your AWS account.

Because of this, we are going to look at creating a user for Ansible which only has permission to access the parts of AWS we know that Ansible will need to interact with for the tasks we are covering in this chapter. We will be giving Ansible full access to the following services:

- Amazon VPC
- Amazon EC2
- Amazon RDS
- Amazon EFS

To do this, log in to the AWS console, which can be found at <https://console.aws.amazon.com/>. Once logged in, click on Services, which can be found in the menu at the very top of the screen. In the menu which opens, enter IAM into the search box and then click on what should be the only result, IAM Manage User Access and Encryption Keys. This will take you to a page that looks something similar to the following:

The screenshot shows the AWS Identity and Access Management (IAM) console. At the top, there's a navigation bar with the AWS logo, 'Services' dropdown, 'Resource Groups' dropdown, user 'Russell Mckendrick', 'Global' dropdown, and 'Support' dropdown. Below the navigation is a search bar labeled 'Search IAM'. On the left, a sidebar menu lists 'Dashboard', 'Groups', 'Users', 'Roles', 'Policies', 'Identity providers', 'Account settings', 'Credential report', and 'Encryption keys'. The main content area has a title 'Welcome to Identity and Access Management'. It includes a 'IAM users sign-in link' (a blue button), 'IAM Resources' section showing 'Users: 0', 'Roles: 0', 'Groups: 0', and 'Identity Providers: 0', and a 'Customer Managed Policies: 1' link. A 'Security Status' section shows a progress bar at '3 out of 5 complete.' with five items: 'Delete your root access keys' (green checkmark), 'Activate MFA on your root account' (green checkmark), 'Create individual IAM users' (orange warning icon), 'Use groups to assign permissions' (orange warning icon), and 'Apply an IAM password policy' (green checkmark). To the right, there's a 'Feature Spotlight' box titled 'Introduction to AWS IAM' with a video player showing 0:00 / 2:16. Below it are links to 'Additional Information' such as 'IAM best practices', 'IAM documentation', 'Web Identity Federation Playground', 'Policy Simulator', and 'Videos, IAM release history and additional resources'. At the bottom, there are 'Feedback' and 'English (US)' buttons, and a footer with copyright information: '© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.' and links to 'Privacy Policy' and 'Terms of Use'.

On the IAM page, click on Groups in the left-hand side menu; we are going to be creating a group which will have the permissions assigned to it, then we will create a user and assign it to our group.

Once you are on the Groups page, click on the Create New Group button. This process has three main steps, the first of which is setting the name of the group. In the space provided, enter the group name `Ansible` and then click on the Next Step button.

The next step is where we attach the policies; we will be using the Amazon-

supplied ones. Select AmazonEC2FullAccess, AmazonVPCFullAccess, AmazonRDSFullAccess, and AmazonElasticFileSystemFullAccess; once all four have been selected, click on the Next Step button.

You should now be on a page which is giving you an overview of the options you have selected; it should look something like the following:

The screenshot shows the AWS IAM interface. At the top, there's a navigation bar with icons for Home, Services (selected), Resource Groups, and a user dropdown for Russell Mckendrick. Below the navigation is a toolbar with various buttons. The main content area has a left sidebar titled "Create New Group Wizard" with steps: Step 1 : Group Name, Step 2 : Attach Policy, and Step 3 : Review (which is currently selected). The main panel is titled "Review" and contains the following information:

Group Name	Ansible	Edit Group Name
Policies	arn:aws:iam::aws:policy/AmazonEC2FullAccess arn:aws:iam::aws:policy/AmazonVPCFullAccess arn:aws:iam::aws:policy/AmazonRDSFullAccess	Edit Policies

At the bottom right of the main panel are three buttons: "Cancel", "Previous", and a blue "Create Group" button.

When you are happy with your selection, click on the Create Group button and then click on Users in the left-hand side menu.

Once on the Users page, click on Add user, and this will take you to a page where you can configure your desired username and also what type of user you want. Enter the following information:

- User name: Enter `Ansible` in here
- AWS access type: Check the box next to where it says Programmatic access; our `Ansible` user does not need AWS Management Console access so leave that option unchecked

You should now be able to click on the Next: Permissions button; this will take you to the page where you set the permissions for your user. As we have already created the group, select the `Ansible` group from the list and then click on Next: Review which will take you to an overview of the options you have entered. If

you are happy with them, then click on the Create user button.

This will take you to a page that looks like the following (I have blurred the access key ID on purpose):

The screenshot shows the AWS Management Console interface. At the top, there's a toolbar with various icons. Below it is a navigation bar with 'Services', 'Resource Groups', and other account-related links. On the right side of the navigation bar, there's a dropdown for 'Russell Mckendrick', 'Global', and 'Support'. Below the navigation bar, there are four numbered tabs (1, 2, 3, 4) with tab 4 being active. The main content area has a title 'Add user' and a 'Success' message: 'You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.' It also includes a link to the AWS Management Console sign-in page. At the bottom left, there's a 'Download .csv' button. A table lists the created user 'Ansible' with columns for User, Access key ID, and Secret access key. The Access key ID is blurred. The Secret access key is shown as a series of asterisks followed by a 'Show' button. The footer contains links for Feedback, English (US), and legal notices: © 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use.

As you can see, the success message tells you that this is the last time you will be able to download the credentials, meaning that you will not be able to see the secret access key again. Either click on the Show button and make a note of the key or click the Download .csv button; you will not have the opportunity to recover the secret access key, only to allow it to expire and generate a new one.

Now that we have an access key ID and secret access key for a user with the permissions we need to launch our VPC using Ansible, we can make a start at writing the playbook.

The VPC playbook

The first thing we need to discuss is how we can pass our access key ID and also the secret access key to Ansible in a safe and secure way. As I will be sharing the final playbooks in a public repository on GitHub, I do not want to share my AWS keys with the world as that could get expensive! Typically, if it were a private repository, I would use Ansible Vault to encrypt the keys and include them in there with other potentially sensitive data such as deployment keys and so on.

In this case, I don't want to include any encrypted information in the repository as it would mean that people would need to unencrypt it, edit the values, and then re-encrypt it. Luckily, the AWS modules provided by Ansible allows you to set two environment variables on your Ansible controller; those variables will then be read as part of the playbook execution.

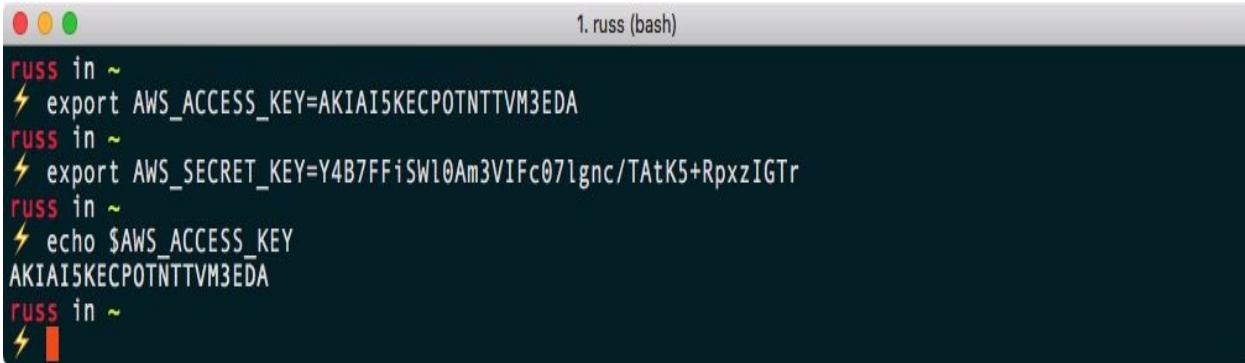
To set the variables, run the following commands to make sure that you replace the content with your own access key and secret (the information listed as follows is just placeholder values):

```
| $ export AWS_ACCESS_KEY=AKIAI5KECPOTNTTVM3EDA  
| $ export AWS_SECRET_KEY=Y4B7FFiSWl0Am3VIFc07lgnc/TAtK5+RpxzIGTr
```

Once set, you can view the contents by running:

```
| $ echo $AWS_ACCESS_KEY
```

As you can see from the output, this will display the content of the `AWS_ACCESS_KEY` variable:



```
russ in ~
⚡ export AWS_ACCESS_KEY=AKIAI5KECPOTNTTVM3EDA
russ in ~
⚡ export AWS_SECRET_KEY=Y4B7FFiSWl0Am3VIFc07lgnc/TAtK5+RpxzIGTr
russ in ~
⚡ echo $AWS_ACCESS_KEY
AKIAI5KECPOTNTTVM3EDA
russ in ~
⚡
```

Now that we have a way to pass our credentials to Ansible, we can create the playbook structure by running the following commands:

```
$ mkdir vpc vpc/group_vars vpc/roles
$ touch vpc/production vpc/site.yml vpc/group_vars/common.yml
$ cd vpc
```

Now that we have the basics in place, we can make a start at creating the roles; unlike previous chapters, we are going to be running the playbook after we have added each role so we can discuss in more detail what has happened.

The VPC role

The first role we are going to create is the one which creates the VPC itself. Everything we are going to configure/create in the upcoming roles needs to be hosted within a VPC, so it needs to be created and then we need to gather some information on it so we can proceed with the rest of the playbook.

To bootstrap the role, run the following command from within your working folder:

```
| $ ansible-galaxy init roles/vpc
```

Now that we have the files for the role, open `roles/vpc/tasks/main.yml` and enter the following:

```
- name: ensure that the VPC is present
  ec2_vpc_net:
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}"
    state: present
    cidr_block: "{{ vpc_cidr_block }}"
    resource_tags: { "Name" : "{{ environment_name }}", "Environment" : "{{ environment_name }}" }
  register: vpc_info

# - name: print the information we have registered
#   debug:
#     msg: "{{ vpc_info }}"
```

As you can see, we are using an Ansible module called `ec2_vpc_net`; this module replaces one called `ec2_vpc` which was deprecated and removed in Ansible 2.5.

We are using three variables within the task; the first two, `ec2_region` and `environment_name`, should be placed in `group_vars/common.yml` as we will be using them in most of the of the roles we will be creating:

```
| environment_name: "my-vpc"
| ec2_region: "eu-west-1"
```

Both variables are self-explanatory: the first is the name we will be using to reference the various elements we will be launching in AWS and the second lets Ansible know where we would like the VPC to be created.

The third variable, `vpc_cidr_block`, should be placed in the `roles/vpc/defaults/main.yml` file:

```
| vpc_cidr_block: "10.0.0.0/16"
```

This defines the CIDR we want to use; `10.0.0.0/16` means that we would like to reserve 10.0.0.1 to 10.0.255.254 which gives us a range of around 65,534 usable IP address, which should be more than enough for our tests.

At the end of the first task, we are using the `register` flag to take all of the content which has been captured during the creation of the VPC and register that as a variable. We are then using the `debug` module to print this content to the screen.

Now that we have our first role, we can add some content to our `site.yml` file:

```
- name: Create and configure an Amazon VPC
  hosts: localhost
  connection: local
  gather_facts: True

  vars_files:
    - group_vars/common.yml
    - group_vars/firewall.yml
    - group_vars/secrets.yml
    - group_vars/words.yml
    - group_vars/keys.yml

  roles:
    - roles/vpc
```

Then run the playbook using:

```
| $ ansible-playbook site.yml
```

This should give you something like the following output:

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the
implicit
localhost does not match 'all'

PLAY [Create and configure an Amazon VPC]
*****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [roles/vpc : ensure that the VPC is present]
*****
changed: [localhost]

TASK [roles/vpc : print the information we have registered]
```

```
*****
ok: [localhost] => {
    "msg": {
        "changed": true,
        "failed": false,
        "vpc": {
            "cidr_block": "10.0.0.0/16",
            "cidr_block_association_set": [
                {
                    "association_id": "vpc-cidr-assoc-1eee5575",
                    "cidr_block": "10.0.0.0/16",
                    "cidr_block_state": {
                        "state": "associated"
                    }
                }
            ],
            "classic_link_enabled": false,
            "dhcp_options_id": "dopt-44851321",
            "id": "vpc-ccef75aa",
            "instance_tenancy": "default",
            "is_default": false,
            "state": "available",
            "tags": {
                "Environment": "my-vpc",
                "Name": "my-vpc"
            }
        }
    }
}
PLAY RECAP
*****
localhost : ok=3 changed=1 unreachable=0 failed=0
```

Checking the VPC section of the AWS console should show you that the VPC has been created, and the information should match what has been captured by Ansible:

The screenshot shows the AWS VPC console interface. At the top, there's a navigation bar with the AWS logo, 'Services', 'Resource Groups', and user information ('Russell Mckendrick', 'Ireland', 'Support'). Below the navigation bar is a search bar with placeholder text 'Search VPCs and their properties'. A table lists two VPCs:

Name	VPC ID	State	IPv4 CIDR	IPv6 CIDR	DHCP options set	Route table	Network ACL	Tenancy
vpc-205dff47	vpc-205dff47	available	172.16.0.0/16		dopt-44851321	rtb-a3a19bc4	acl-b999eade	Default
my-vpc	vpc-ccef75aa	available	10.0.0.0/16		dopt-44851321	rtb-49e85430	acl-a98d9acf	Default

Below the table, the 'my-vpc' VPC is selected. The 'Summary' tab is active, showing the following details:

VPC ID: vpc-ccef75aa my-vpc	Network ACL: acl-a98d9acf
State: available	Tenancy: Default
IPv4 CIDR: 10.0.0.0/16	DNS resolution: yes
IPv6 CIDR:	DNS hostnames: yes
DHCP options set: dopt-44851321	ClassicLink DNS Support: no
Route table: rtb-49e85430	
ClassicLink: Disabled	

At the bottom of the page, there are links for 'Feedback', 'English (US)', and copyright information: '© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use'.

If you rerun the playbook, you will notice that, rather than creating the VPC again, Ansible recognizes that there is already a VPC called `my-vpc` and it discovers the information on the already existing VPC and populates the `vpc_info` variable. This is useful as we will be using the information gathered in the next role.

The subnets role

Now that we have our VPC, we can start to populate it. The first thing we are going to configure is the 10 subnets. If you recall, we need the following:

- Three EC2 instances
- Three ELB instances
- Two RDS instances
- Two EFS instances

Create the role by running the following command from your working directory:

```
| $ ansible-galaxy init roles/subnets
```

Now, in `roles/subnets/defaults/main.yml`, enter the following:

```
the_subnets:
  - { use: 'ec2', az: 'a', subnet: '10.0.10.0/24' }
  - { use: 'ec2', az: 'b', subnet: '10.0.11.0/24' }
  - { use: 'ec2', az: 'c', subnet: '10.0.12.0/24' }
  - { use: 'elb', az: 'a', subnet: '10.0.20.0/24' }
  - { use: 'elb', az: 'b', subnet: '10.0.21.0/24' }
  - { use: 'elb', az: 'c', subnet: '10.0.22.0/24' }
  - { use: 'rds', az: 'a', subnet: '10.0.30.0/24' }
  - { use: 'rds', az: 'b', subnet: '10.0.31.0/24' }
  - { use: 'efs', az: 'b', subnet: '10.0.40.0/24' }
  - { use: 'efs', az: 'c', subnet: '10.0.41.0/24' }
```

As you can see, we have a list of variables containing what the subnet is being used for (ec2, elb, rds, or efs), which availability zone the subnet should be created in (a, b, or c), and then the subnet itself. Here we are using a /24 for each of the availability zones.

Grouping the subnets like this should remove some of the repetition when it comes to creating the subnets. However, it doesn't remove it all, as we can see from the content of `roles/subnets/tasks/main.yml`:

```
- name: ensure that the subnets are present
  ec2_vpc_subnet:
    region: "{{ ec2_region }}"
    state: present
    vpc_id: "{{ vpc_info.vpc.id }}"
    cidr: "{{ item.subnet }}"
    az: "{{ ec2_region }}{{ item.az }}"
    resource_tags:
```

```

    "Name" : "{{ environment_name }}_{{ item.use }}_{{ ec2_region }}{{ item.az }}"
    "Environment" : "{{ environment_name }}"
    "Use" : "{{ item.use }}"
with_items: "{{ the_subnets }}"

```

The task starts off pretty simple: here we are using the `ec2_vpc_subnet` module to create subnets by looping through the `the_subnets` variable. As you can see, we are using the variable we registered in the previous role to correctly deploy the subnets into our VPC; this is `vpc_info.vpc.id`.

You may have noticed that we are not registering the results of this task; this is because, if we did, we would have had information on all ten subnets. Instead, we want to break this information down based on what the subnet is being used for. To find this information out, we can use the `ec2_vpc_subnet_facts` module to gather information based on our filtering using the `Environment` and `use` tag we set when creating the subnets:

```

- name: gather information about the ec2 subnets
  ec2_vpc_subnet_facts:
    region: "{{ ec2_region }}"
    filters:
      "tag:Use": "ec2"
      "tag:Environment": "{{ environment_name }}"
  register: subnets_ec2

- name: gather information about the elb subnets
  ec2_vpc_subnet_facts:
    region: "{{ ec2_region }}"
    filters:
      "tag:Use": "elb"
      "tag:Environment": "{{ environment_name }}"
  register: subnets_elb

- name: gather information about the rds subnets
  ec2_vpc_subnet_facts:
    region: "{{ ec2_region }}"
    filters:
      "tag:Use": "rds"
      "tag:Environment": "{{ environment_name }}"
  register: subnets_rds

- name: gather information about the efs subnets
  ec2_vpc_subnet_facts:
    region: "{{ ec2_region }}"
    filters:
      "tag:Use": "efs"
      "tag:Environment": "{{ environment_name }}"
  register: subnets_efs

```

As you can see, here we are filtering the use of and registering four different sets of information: `subnets_ec2`, `subnets_elb`, `subnets_rds`, and `subnets_efs`. We are not quite there yet, however, because we only want to know the subnet IDs rather than all

of the information about each of the subnets.

To do this, we need to use the `set_fact` module and some Jinja2 filtering:

```
- name: register just the IDs for each of the subnets
  set_fact:
    subnet_ec2_ids: "{{ subnets_ec2.subnets | map(attribute='id') | list }}"
    subnet_elb_ids: "{{ subnets_elb.subnets | map(attribute='id') | list }}"
    subnet_rds_ids: "{{ subnets_rds.subnets | map(attribute='id') | list }}"
    subnet_efs_ids: "{{ subnets_efs.subnets | map(attribute='id') | list }}"
```

Finally, we can print out all of the IDs to the screen in one big list by joining the variables together:

```
# - name: print all the ids we have registered
#   debug:
#     msg: "{{ subnet_ec2_ids + subnet_elb_ids + subnet_rds_ids
#           + subnet_efs_ids }}"
```

Now that we have all of the parts of our role together, let's run it. Update the `site.yml` file so it looks like the following:

```
- name: Create and configure an Amazon VPC
  hosts: localhost
  connection: local
  gather_facts: True

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/vpc
    - roles/subnets
```

Then run the playbook using:

```
| $ ansible-playbook site.yml
```

Before running the playbook, I commented out the `debug` task in the VPC role. Your output should look something like the output that follows; you may have noticed that the VPC role returns an `ok` as our VPC is there:

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the
implicit localhost does not match 'all'

PLAY [Create and configure an Amazon VPC]
*****
TASK [Gathering Facts]
*****
ok: [localhost]
```

```
TASK [roles/vpc : ensure that the VPC is present]
*****
ok: [localhost]

TASK [roles/subnets : ensure that the subnets are present]
*****
changed: [localhost] => (item={'subnet': u'10.0.10.0/24', 'use': u'ec2', 'az': u'a'})
changed: [localhost] => (item={'subnet': u'10.0.11.0/24', 'use': u'ec2', 'az': u'b'})
changed: [localhost] => (item={'subnet': u'10.0.12.0/24', 'use': u'ec2', 'az': u'c'})
changed: [localhost] => (item={'subnet': u'10.0.20.0/24', 'use': u'elb', 'az': u'a'})
changed: [localhost] => (item={'subnet': u'10.0.21.0/24', 'use': u'elb', 'az': u'b'})
changed: [localhost] => (item={'subnet': u'10.0.22.0/24', 'use': u'elb', 'az': u'c'})
changed: [localhost] => (item={'subnet': u'10.0.30.0/24', 'use': u'rds', 'az': u'a'})
changed: [localhost] => (item={'subnet': u'10.0.31.0/24', 'use': u'rds', 'az': u'b'})
changed: [localhost] => (item={'subnet': u'10.0.40.0/24', 'use': u'efs', 'az': u'b'})
changed: [localhost] => (item={'subnet': u'10.0.41.0/24', 'use': u'efs', 'az': u'c'})

TASK [roles/subnets : gather information about the ec2 subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the elb subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the rds subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the efs subnets]
*****
ok: [localhost]

TASK [roles/subnets : register just the IDs for each of the subnets]
*****
ok: [localhost]

TASK [roles/subnets : print all the ids we have registered]
*****
ok: [localhost] => {
  "msg": [
    "subnet-2951e761",
    "subnet-24ea4a42",
    "subnet-fce80ba6",
    "subnet-6744f22f",
    "subnet-64eb083e",
    "subnet-51f15137",
    "subnet-154ef85d",
    "subnet-19e9497f",
    "subnet-4340f60b",
    "subnet-5aea0900"
  ]
}
```

```

}

PLAY RECAP
*****
localhost : ok=9 changed=1 unreachable=0 failed=0

```

The only change recorded is the addition of the subnets; if we were to run it again, then this would also return an `ok` as the subnets exist. As you can also see, we have ten subnet IDs returned and this is also reflected in the AWS console:

The screenshot shows the AWS Management Console interface for managing VPC subnets. The URL in the address bar is `eu-west-1.console.aws.amazon.com`. The top navigation bar includes the AWS logo, Services dropdown, Resource Groups dropdown, and user information (Russell Mckendrick, Ireland, Support). Below the navigation is a search bar with the query `my-vpc`. The main content area displays a table titled "Subnet Actions" showing 10 subnets. The columns are: Name, Subnet ID, State, VPC, IPv4 CIDR, Available IPv4 Addresses, and Availability Zone. The subnets listed are:

Name	Subnet ID	State	VPC	IPv4 CIDR	Available IPv4 Addresses	Availability Zone
my-vpc_ec2_eu-west-1a	subnet-24ea4a42	available	vpc-cccf75aa my-vpc	10.0.10.0/24	251	eu-west-1a
my-vpc_ec2_eu-west-1b	subnet-2951e761	available	vpc-cccf75aa my-vpc	10.0.11.0/24	251	eu-west-1b
my-vpc_ec2_eu-west-1c	subnet-fce80ba6	available	vpc-cccf75aa my-vpc	10.0.12.0/24	251	eu-west-1c
my-vpc_efs_eu-west-1b	subnet-4340f60b	available	vpc-cccf75aa my-vpc	10.0.40.0/24	251	eu-west-1b
my-vpc_efs_eu-west-1c	subnet-5aea0900	available	vpc-cccf75aa my-vpc	10.0.41.0/24	251	eu-west-1c
my-vpc_elb_eu-west-1a	subnet-51f15137	available	vpc-cccf75aa my-vpc	10.0.20.0/24	251	eu-west-1a
my-vpc_elb_eu-west-1b	subnet-6744f22f	available	vpc-cccf75aa my-vpc	10.0.21.0/24	251	eu-west-1b
my-vpc_elb_eu-west-1c	subnet-64eb083e	available	vpc-cccf75aa my-vpc	10.0.22.0/24	251	eu-west-1c
my-vpc_rds_eu-west-1a	subnet-19e9497f	available	vpc-cccf75aa my-vpc	10.0.30.0/24	251	eu-west-1a
my-vpc_rds_eu-west-1b	subnet-154ef85d	available	vpc-cccf75aa my-vpc	10.0.31.0/24	251	eu-west-1b

Below the table, there is a message: "Select a subnet above". At the bottom of the page are links for Feedback, English (US), Copyright notice (© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.), Privacy Policy, and Terms of Use.

Now that we have our subnets, we need to make sure that the EC2 instances can route to the internet.

The internet gateway role

While the internet gateway role is going to only variables which we have defined in `common.yml`, and through gathering information in previous tasks, we should continue to bootstrap the `roles` folder as we have been doing:

```
| $ ansible-galaxy init roles/gateway
```

We are going to be using two modules in the role; the first, `ec2_vpc_igw`, creates the internet gateway and tags it:

```
- name: ensure that there is an internet gateway
  ec2_vpc_igw:
    region: "{{ ec2_region }}"
    vpc_id: "{{ vpc_info.vpc.id }}"
    state: present
    tags:
      "Name": "{{ environment_name }}_internet_gateway"
      "Environment": "{{ environment_name }}"
      "Use": "gateway"
  register: igw_info
```

We then print the information we have registered about the internet gateway to the screen:

```
# - name: print the information we have registered
#   debug:
#     msg: "{{ igw_info }}"
```

Before finally using the second module, `ec2_vpc_route_table`, we create a route which sends all traffic destined for `0.0.0.0/0` to the newly created internet gateway for just the EC2 subnets using the list of IDs we created in the previous role:

```
- name: check that we can route through internet gateway
  ec2_vpc_route_table:
    region: "{{ ec2_region }}"
    vpc_id: "{{ vpc_info.vpc.id }}"
    subnets: "{{ subnet_ec2_ids + subnet_elb_ids }}"
    routes:
      - dest: 0.0.0.0/0
        gateway_id: "{{ igw_info.gateway_id }}"
    resource_tags:
      "Name": "{{ environment_name }}_outbound"
      "Environment": "{{ environment_name }}"
```

Add the role of the `site.yml` file:

```

- name: Create and configure an Amazon VPC
  hosts: localhost
  connection: local
  gather_facts: True

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/vpc
    - roles/subnets
    - roles/gateway

```

Then run the playbook:

```
| $ ansible-playbook site.yml
```

At this point, as we have run the playbook three times now, I should quickly mention the `WARNING`. This is because we are not using an inventory file, as we have defined `localhost` at the top of our `site.yml` file. You should receive something like the following output; again I have commented out the debug tasks from previous roles:

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the
implicit localhost does not match 'all'

PLAY [Create and configure an Amazon VPC]
*****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [roles/vpc : ensure that the VPC is present]
*****
ok: [localhost]

TASK [roles/subnets : ensure that the subnets are present]
*****
ok: [localhost] => (item={u'subnet': u'10.0.10.0/24', u'use': u'ec2', u'az': u'a'})
ok: [localhost] => (item={u'subnet': u'10.0.11.0/24', u'use': u'ec2', u'az': u'b'})
ok: [localhost] => (item={u'subnet': u'10.0.12.0/24', u'use': u'ec2', u'az': u'c'})
ok: [localhost] => (item={u'subnet': u'10.0.20.0/24', u'use': u'elb', u'az': u'a'})
ok: [localhost] => (item={u'subnet': u'10.0.21.0/24', u'use': u'elb', u'az': u'b'})
ok: [localhost] => (item={u'subnet': u'10.0.22.0/24', u'use': u'elb', u'az': u'c'})
ok: [localhost] => (item={u'subnet': u'10.0.30.0/24', u'use': u'rds', u'az': u'a'})
ok: [localhost] => (item={u'subnet': u'10.0.31.0/24', u'use': u'rds', u'az': u'b'})
ok: [localhost] => (item={u'subnet': u'10.0.40.0/24', u'use': u'efs', u'az': u'b'})
ok: [localhost] => (item={u'subnet': u'10.0.41.0/24', u'use': u'efs', u'az': u'c'})

TASK [roles/subnets : gather information about the ec2 subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the elb subnets]
*****
ok: [localhost]
```

```

TASK [roles/subnets : gather information about the rds subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the efs subnets]
*****
ok: [localhost]

TASK [roles/subnets : register just the IDs for each of the subnets]
*****
ok: [localhost]

TASK [roles/gateway : ensure that there is an internet gateway]
*****
changed: [localhost]

TASK [roles/gateway : print the information we have registered]
*****
ok: [localhost] => {
  "msg": {
    "changed": true,
    "failed": false,
    "gateway_id": "igw-a74235c0",
    "tags": [
      "Environment": "my-vpc",
      "Name": "my-vpc_internet_gateway",
      "Use": "gateway"
    ],
    "vpc_id": "vpc-ccef75aa"
  }
}

TASK [roles/gateway : check that we can route through internet gateway]
*****
changed: [localhost]

PLAY RECAP
*****
localhost : ok=11 changed=2 unreachable=0 failed=0

```

Back to the AWS console. You should be able to view the internet gateway:

The screenshot shows the AWS VPC Dashboard. On the left sidebar, under the 'Internet Gateways' section, there is a link to 'Route Tables'. The main content area displays a table of Internet Gateways. The table has columns for Name, ID, State, and VPC. There are two entries:

Name	ID	State	VPC
	igw-655b1401	attached	vpc-205dff47
my-vpc_inte...	igw-a74235c0	attached	vpc-ccef75aa my...

Below the table, a section titled 'Internet gateway: igw-a74235c0' is expanded, showing its details:

Description	Tags
ID: igw-a74235c0	Attached VPC ID: vpc-ccef75aa my-vpc
State: attached	

At the bottom of the page, there are links for 'Feedback', 'English (US)', and copyright information: '© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.' followed by 'Privacy Policy' and 'Terms of Use'.

In the preceding screenshot, you can see the default VPC internet gateway and also the one we created using Ansible. You can also see the route table we created:

The screenshot shows the AWS VPC Dashboard. On the left sidebar, under 'Route Tables', the 'Summary' tab is selected. A search bar at the top of the main content area contains the text 'my-vpc'. The results table shows two route tables:

Name	Route Table ID	Explicitly Associated With	Main	VPC
my-vpc_outbound	rtb-9dd569e4	3 Subnets	No	vpc-ccef75aa my-vpc
	rtb-49e85430	0 Subnets	Yes	vpc-ccef75aa my-vpc

Below the table, the route table 'rtb-9dd569e4 | my-vpc_outbound' is selected. At the bottom of the page, there are links for 'Feedback', 'English (US)', and copyright information.

Here you can see the route Ansible configured along with the default route that was created when we created the VPC. This default route is set to be the main route and allows routing between all of the subnets we added in the previous role.

Next up, we need to add some security groups to our VPC.

The security group role

We have a few different aims with this role. The first is simple enough: create a security group which opens port `80` and `443` to the world, or `0.0.0.0/0` in IP terms. The second aim is to create a rule that allows SSH access, but only to us, and the third is to make sure that only our EC2 instances can connect to RDS and EFS.

The first aim is easy as `0.0.0.0/0` is a known quantity, the others not so much. Our IP could often change so we don't want to hardcode it. Also, we haven't launched any EC2 instances, so we don't know their IP addresses.

Let's bootstrap the role and create the first set of rules:

```
| $ ansible-galaxy init roles/securitygroups
```

We are going to be using the `ec2_group` module to create our first group in `roles/securitygroups/tasks/main.yml`:

```
- name: provision elb security group
  ec2_group:
    region: "{{ ec2_region }}"
    vpc_id: "{{ vpc_info.vpc.id }}"
    name: "{{ environment_name }}-elb"
    description: "opens port 80 and 443 to the world"
    tags:
      "Name": "{{ environment_name }}-elb"
      "Environment": "{{ environment_name }}"
    rules:
      - proto: "tcp"
        from_port: "80"
        to_port: "80"
        cidr_ip: "0.0.0.0/0"
        rule_desc: "allow all on port 80"
      - proto: "tcp"
        from_port: "443"
        to_port: "443"
        cidr_ip: "0.0.0.0/0"
        rule_desc: "allow all on port 443"
  register: sg_elb
```

Here we are creating a rule called `my-vpc-elb`, tagging it, and then opening up ports `80` and `443` to `0.0.0.0/0`. As you can see, adding a rule is easy when you know the source IP address is straightforward. Let's now look at adding the rule for the EC2 instances; this one is a little different.

First of all, we do not want to give everyone access to SSH on our instances so we need to know the IP address of our Ansible controller. To do this we will use the `ipify_facts` module.



ipify is a free web API which simply returns the current public IP address of the device you use to query the API.

As you can see from the tasks that follow, we are making an API call to ipify and then setting a fact which contains the IP address before printing the IP address to the screen:

```
- name: find out your current public IP address using https://ipify.org/
  ipify_facts:
    register: public_ip

- name: set your public ip as a fact
  set_fact:
    your_public_ip: "{{ public_ip.ansible_facts.ipify_public_ip }}/32"

# - name: print your public ip address
#   debug:
#     msg: "Your public IP address is {{ your_public_ip }}"
```

Now that we know what IP address to allow access to port 22, we can create a rule called `my-vpc-ec2`:

```
- name: provision ec2 security group
  ec2_group:
    region: "{{ ec2_region }}"
    vpc_id: "{{ vpc_info.vpc.id }}"
    name: "{{ environment_name }}-ec2"
    description: "opens port 22 to a trusted IP and port 80 to the elb group"
    tags:
      "Name": "{{ environment_name }}-ec2"
      "Environment": "{{ environment_name }}"
    rules:
      - proto: "tcp"
        from_port: "22"
        to_port: "22"
        cidr_ip: "{{ your_public_ip }}"
        rule_desc: "allow {{ your_public_ip }} access to port 22"
      - proto: "tcp"
        from_port: "80"
        to_port: "80"
        group_id: "{{ sg_elb.group_id }}"
        rule_desc: "allow {{ sg_elb.group_id }} access to port 80"
  register: sg_ec2
```

There is also the second rule in the `my-vpc-ec2` security group; this rule allows access to port 80 from any source which has the `my-vpc-elb` security group attached, which in our case will be just the ELBs. This means that the only way anyone can access port 80 on our EC2 instances is through the ELB.

We are going to use this same principle to create the RDS and EFS groups, this time only allowing access to ports 3306 and 2049 respectively to any instances in the `my-vpc-ec2` security group:

```
- name: provision rds security group
  ec2_group:
    region: "{{ ec2_region }}"
    vpc_id: "{{ vpc_info.vpc.id }}"
    name: "{{ environment_name }}-rds"
    description: "opens port 3306 to the ec2 instances"
    tags:
      "Name": "{{ environment_name }}-rds"
      "Environment": "{{ environment_name }}"
    rules:
      - proto: "tcp"
        from_port: "3306"
        to_port: "3306"
        group_id: "{{ sg_ec2.group_id }}"
        rule_desc: "allow {{ sg_ec2.group_id }} access to port 3306"
  register: sg_rds

- name: provision efs security group
  ec2_group:
    region: "{{ ec2_region }}"
    vpc_id: "{{ vpc_info.vpc.id }}"
    name: "{{ environment_name }}-efs"
    description: "opens port 2049 to the ec2 instances"
    tags:
      "Name": "{{ environment_name }}-efs"
      "Environment": "{{ environment_name }}"
    rules:
      - proto: "tcp"
        from_port: "2049"
        to_port: "2049"
        group_id: "{{ sg_ec2.group_id }}"
        rule_desc: "allow {{ sg_ec2.group_id }} access to port 2049"
  register: sg_efs
```

Now that we have our main groups created, let's add a `debug` task to print the security group IDs to the screen:

```
# - name: print all the ids we have registered
#   debug:
#     msg: "ELB = {{ sg_elb.group_id }}, EC2 = {{ sg_ec2.group_id }}, RDS = {{ sg_rds.group_id }} and EFS = {{ sg_efs.group_id }}"
```

Now that we have our full role, we can run the playbook. Remember to add `- roles/securitygroups` to the `site.yml` file:

```
| $ ansible-playbook site.yml
```

Again, I have commented out any output from the `debug` module outside the `securitygroups` role:

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the
implicit localhost does not match 'all'

PLAY [Create and configure an Amazon VPC]
*****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [roles/vpc : ensure that the VPC is present]
*****
ok: [localhost]

TASK [roles/subnets : ensure that the subnets are present]
*****
ok: [localhost] => (item={u'subnet': u'10.0.10.0/24', u'use': u'ec2', u'az': u'a'})
ok: [localhost] => (item={u'subnet': u'10.0.11.0/24', u'use': u'ec2', u'az': u'b'})
ok: [localhost] => (item={u'subnet': u'10.0.12.0/24', u'use': u'ec2', u'az': u'c'})
ok: [localhost] => (item={u'subnet': u'10.0.20.0/24', u'use': u'elb', u'az': u'a'})
ok: [localhost] => (item={u'subnet': u'10.0.21.0/24', u'use': u'elb', u'az': u'b'})
ok: [localhost] => (item={u'subnet': u'10.0.22.0/24', u'use': u'elb', u'az': u'c'})
ok: [localhost] => (item={u'subnet': u'10.0.30.0/24', u'use': u'rds', u'az': u'a'})
ok: [localhost] => (item={u'subnet': u'10.0.31.0/24', u'use': u'rds', u'az': u'b'})
ok: [localhost] => (item={u'subnet': u'10.0.40.0/24', u'use': u'efs', u'az': u'b'})
ok: [localhost] => (item={u'subnet': u'10.0.41.0/24', u'use': u'efs', u'az': u'c'})

TASK [roles/subnets : gather information about the ec2 subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the elb subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the rds subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the efs subnets]
*****
ok: [localhost]

TASK [roles/subnets : register just the IDs for each of the subnets]
*****
ok: [localhost]

TASK [roles/gateway : ensure that there is an internet gateway]
*****
ok: [localhost]

TASK [roles/gateway : check that we can route through internet gateway]
*****
ok: [localhost]

TASK [roles/securitygroups : provision elb security group]
*****
changed: [localhost]

TASK [roles/securitygroups : find out your current public IP address using
https://ipify.org/] **
ok: [localhost]
```

```
TASK [roles/securitygroups : set your public ip as a fact]
*****
ok: [localhost]

TASK [roles/securitygroups : print your public ip address]
*****
ok: [localhost] => {
    "msg": "Your public IP address is 109.153.155.197/32"
}

TASK [roles/securitygroups : provision ec2 security group]
*****
changed: [localhost]

TASK [roles/securitygroups : provision rds security group]
*****
changed: [localhost]

TASK [roles/securitygroups : provision efs security group]
*****
changed: [localhost]

TASK [roles/securitygroups : print all the ids we have registered]
*****
ok: [localhost] => {
    "msg": "ELB = sg-97778eea, EC2 = sg-fa778e87, RDS = sg-8e7089f3 and EFS = sg-7b718806"
}

PLAY RECAP
*****
localhost : ok=18 changed=4 unreachable=0 failed=0
```

You can view the groups that Ansible has created in the AWS console. In the following screenshot, you can see the `my-vpc-ec2` security group:

The screenshot shows the AWS VPC Dashboard. On the left sidebar, under 'Virtual Private Cloud', the 'Your VPCs' section is selected. In the main content area, the 'Create Security Group' button is highlighted. A search bar at the top right contains the text 'my-vpc'. The main table lists five security groups:

Name tag	Group ID	Group Name	VPC	Description
my-vpc-efs	sg-7b718806	my-vpc-efs	vpc-ccef75aa my-vpc	opens port 2049 to the ec2 instances
	sg-7df60f00	default	vpc-ccef75aa my-vpc	default VPC security group
my-vpc-rds	sg-8e7089f3	my-vpc-rds	vpc-ccef75aa my-vpc	opens port 3306 to the ec2 instances
my-vpc-elb	sg-97778eea	my-vpc-elb	vpc-ccef75aa my-vpc	opens port 80 and 443 to the world
my-vpc-ec2	sg-fa778e87	my-vpc-ec2	vpc-ccef75aa my-vpc	opens port 22 to a trusted IP and port 80...

The 'my-vpc-ec2' row is selected. Below the table, the 'Inbound Rules' tab is active. The 'Edit' button is highlighted. The table shows two rules:

Type	Protocol	Port Range	Source	Description
HTTP (80)	TCP (6)	80	sg-97778eea	allow sg-97778eea ac...
SSH (22)	TCP (6)	22	109.153.155.197/32	allow 109.153.155.19...

At the bottom, there are links for Feedback, English (US), and a footer with copyright information and links to Privacy Policy and Terms of Use.

Now that we have our basic VPC configured, we can start to launch services in it, starting with an Application Load Balancer.

The ELB role

The final role we are going to look at in this chapter is one which launches an Application Load Balancer. Well, it creates a target group which is then attached to an Application Load Balancer. The load balancer we will be creating with this role is basic; we will be going into a lot more detail in the later chapter.

Like the other roles, we first need to bootstrap the files:

```
| $ ansible-galaxy init roles/elb
```

Now open `roles/elb/tasks/main.yml` and use the `elb_target_group` module to create the target group:

```
- name: provision the target group
  elb_target_group:
    name: "{{ environment_name }}-target-group"
    region: "{{ ec2_region }}"
    protocol: "http"
    port: "80"
    deregistration_delay_timeout: "15"
    vpc_id: "{{ vpc_info.vpc.id }}"
    state: "present"
    modify_targets: "false"
```

As you can see, we are creating the target group in our VPC and calling it `my-vpc-target-group`. Now we have the target group, we can launch the Application Elastic Balancer using the `elb_application_lb` module:

```
- name: provision an application elastic load balancer
  elb_application_lb:
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}-elb"
    security_groups: "{{ sg_elb.group_id }}"
    subnets: "{{ subnet_elb_ids }}"
    listeners:
      - Protocol: "HTTP"
        Port: "80"
        DefaultActions:
          - Type: "forward"
            TargetGroupName: "{{ environment_name }}-target-group"
    state: present
  register: loadbalancer
```

Here, we are provisioning an Application Load Balancer called `my-vpc-elb` in our VPC; we are passing the IDs of the ELB subnets we created using the

`subnet_elb_ids`. We are also adding the ELB security group to the load balancer using the `sg_elb.group_id` and configuring a listener on port `80`, which forwards traffic to the `my-vpc-target-group`.

The final part of the task prints the information we have captured about the ELB:

```
# - name: print the information on the load balancer we have registered
#   debug:
#     msg: "{{ loadbalancer }}"
```

That completes our final role; update the `site.yml` file so it looks as follows:

```
- name: Create and configure an Amazon VPC
hosts: localhost
connection: local
gather_facts: True

vars_files:
  - group_vars/common.yml

roles:
  - roles/vpc
  - roles/subnets
  - roles/gateway
  - roles/securitygroups
  - roles/elb
```

We can now run our playbook for the final time by running:

```
| $ ansible-playbook site.yml
```

You can probably guess that the output of the playbook run is going to look as follows:

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the
implicit localhost does not match 'all'

PLAY [Create and configure an Amazon VPC]
*****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [roles/vpc : ensure that the VPC is present]
*****
ok: [localhost]

TASK [roles/subnets : ensure that the subnets are present]
*****
ok: [localhost] => (item={'subnet': '10.0.10.0/24', 'use': 'ec2', 'az': 'a'})
ok: [localhost] => (item={'subnet': '10.0.11.0/24', 'use': 'ec2', 'az': 'b'})
ok: [localhost] => (item={'subnet': '10.0.12.0/24', 'use': 'ec2', 'az': 'c'})
ok: [localhost] => (item={'subnet': '10.0.20.0/24', 'use': 'elb', 'az': 'a'})
```

```
ok: [localhost] => (item={u'subnet': u'10.0.21.0/24', u'use': u'elb', u'az': u'b'})
ok: [localhost] => (item={u'subnet': u'10.0.22.0/24', u'use': u'elb', u'az': u'c'})
ok: [localhost] => (item={u'subnet': u'10.0.30.0/24', u'use': u'rds', u'az': u'a'})
ok: [localhost] => (item={u'subnet': u'10.0.31.0/24', u'use': u'rds', u'az': u'b'})
ok: [localhost] => (item={u'subnet': u'10.0.40.0/24', u'use': u'efs', u'az': u'b'})
ok: [localhost] => (item={u'subnet': u'10.0.41.0/24', u'use': u'efs', u'az': u'c'})

TASK [roles/subnets : gather information about the ec2 subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the elb subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the rds subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the efs subnets]
*****
ok: [localhost]

TASK [roles/subnets : register just the IDs for each of the subnets]
*****
ok: [localhost]

TASK [roles/gateway : ensure that there is an internet gateway]
*****
ok: [localhost]

TASK [roles/gateway : check that we can route through internet gateway]
*****
ok: [localhost]

TASK [roles/securitygroups : provision elb security group]
*****
ok: [localhost]

TASK [roles/securitygroups : find out your current public IP address using
https://ipify.org/] **
ok: [localhost]

TASK [roles/securitygroups : set your public ip as a fact]
*****
ok: [localhost]

TASK [roles/securitygroups : provision ec2 security group]
*****
ok: [localhost]

TASK [roles/securitygroups : provision rds security group]
*****
ok: [localhost]

TASK [roles/securitygroups : provision efs security group]
*****
ok: [localhost]

TASK [roles/elb : provision the target group]
*****
changed: [localhost]

TASK [roles/elb : provision an application elastic load balancer]
```

```
*****
changed: [localhost]

TASK [roles/elb : print the information on the load balancer we have registered]
*****
ok: [localhost] => {
  "msg": {
    "access_logs_s3_bucket": "",
    "access_logs_s3_enabled": "false",
    "access_logs_s3_prefix": "",
    "attempts": 1,
    "availability_zones": [
      {
        "subnet_id": "subnet-51f15137",
        "zone_name": "eu-west-1a"
      },
      {
        "subnet_id": "subnet-64eb083e",
        "zone_name": "eu-west-1c"
      },
      {
        "subnet_id": "subnet-6744f22f",
        "zone_name": "eu-west-1b"
      }
    ],
    "canonical_hosted_zone_id": "Z32012XQLNTSW2",
    "changed": true,
    "created_time": "2018-04-22T16:12:31.780000+00:00",
    "deletion_protection_enabled": "false",
    "dns_name": "my-vpc-elb-374523105.eu-west-1.elb.amazonaws.com",
    "failed": false,
    "idle_timeout_timeout_seconds": "60",
    "ip_address_type": "ipv4",
    "listeners": [
      {
        "default_actions": [
          {
            "target_group_arn": "arn:aws:elasticloadbalancing:eu-west-1:687011238589:targetgroup/my-vpc-target-group/d5bab5efb2d314a8",
            "type": "forward"
          }
        ],
        "listener_arn": "arn:aws:elasticloadbalancing:eu-west-1:687011238589:listener/app/my-vpc-elb/98dd881c7a931ab3/3f4be2b480657bf9",
        "load_balancer_arn": "arn:aws:elasticloadbalancing:eu-west-1:687011238589:loadbalancer/app/my-vpc-elb/98dd881c7a931ab3",
        "port": 80,
        "protocol": "HTTP",
        "rules": [
          {
            "actions": [
              {
                "target_group_arn": "arn:aws:elasticloadbalancing:eu-west-1:687011238589:targetgroup/my-vpc-target-group/d5bab5efb2d314a8",
                "type": "forward"
              }
            ],
            "conditions": [],
            "is_default": true,
            "priority": "default",
            "rule_arn": "arn:aws:elasticloadbalancing:eu-west-1:687011238589:listener-rule/app/my-vpc-elb/98dd881c7a931ab3/3f4be2b480657bf9/c70feab5b31460c2"
          }
        ]
      }
    ]
  }
}
```

```

}
],
"load_balancer_arn": "arn:aws:elasticloadbalancing:eu-west-
1:687011238589:loadbalancer/app/my-vpc-elb/98dd881c7a931ab3",
"load_balancer_name": "my-vpc-elb",
"routing_http2_enabled": "true",
"scheme": "internet-facing",
"security_groups": [
"sg-97778eea"
],
"state": {
"code": "provisioning"
},
"tags": {},
"type": "application",
"vpc_id": "vpc-ccef75aa"
}
}

```

PLAY RECAP

```
*****
localhost : ok=19 changed=2 unreachable=0 failed=0
```

You should now be able to see the ELB in the EC2 part of the AWS console:

Name	DNS name	State	VPC ID	Availability Zones	Type
my-vpc-elb	my-vpc-elb-374523105.eu-west-1.elb.amazonaws.com (A Record)	active	vpc-ccef75aa	eu-west-1a, eu-west-1c...	application



While VPC's do not incur any cost, ELBs do; please ensure that you remove any unused resources as soon as you have completed your test.

That concludes this chapter on the VPC playbook; we will be using elements of this in the next chapter, where we will be looking at deploying our WordPress installation into AWS using a VPC as the foundation of our installation.

Summary

In this chapter, we have taken our next step in using Ansible to launch resources in a public cloud. We have laid the groundwork for automating quite a complex environment by creating a VPC, setting up the subnets we need for our application, provisioning an internet gateway, and setting our instances to route their outgoing traffic through it.

We have also configured four security groups, three of which contained dynamic content, to secure our services before finally provisioning an ELB into our VPC.

In the next chapter, we will build on the foundations we have laid in this chapter and launch a more complex set of services.

Questions

1. What are the two environment variables used by the AWS modules to read your access ID and secret?
2. True or false: Every time you run the playbook, you will get a new VPC.
3. State and explain why we are not bothering to register the results of creating the subnets.
4. What is the difference between using `cidr_ip` and `group_id` when defining a rule in a security group?
5. True or false: The order in which the security groups are added when using rules which have `group_id` defined doesn't matter.
6. Create a second VPC alongside the existing VPC, give it a different name, and also have it use 10.1.0.0/24.

Further reading

You can find more details on the AWS technologies we have used in this chapter at the following links:

- **AWS:** <https://aws.amazon.com/>
- **AWS Management Console:** <https://aws.amazon.com/console/>
- **AWS IAM:** <https://aws.amazon.com/iam/>
- **Amazon VPC:** <https://aws.amazon.com/vpc/>
- **ELB:** <https://aws.amazon.com/elasticloadbalancing/>

Highly Available Cloud Deployments

Continuing with our AWS deployment, we will start to deploy services into the network we created in the previous chapter, and by the end of the chapter, we will be left with a highly available WordPress installation, which we will test by removing the instances while sending traffic to the site.

Building on top of the roles we created in the previous chapter, we will be doing the following:

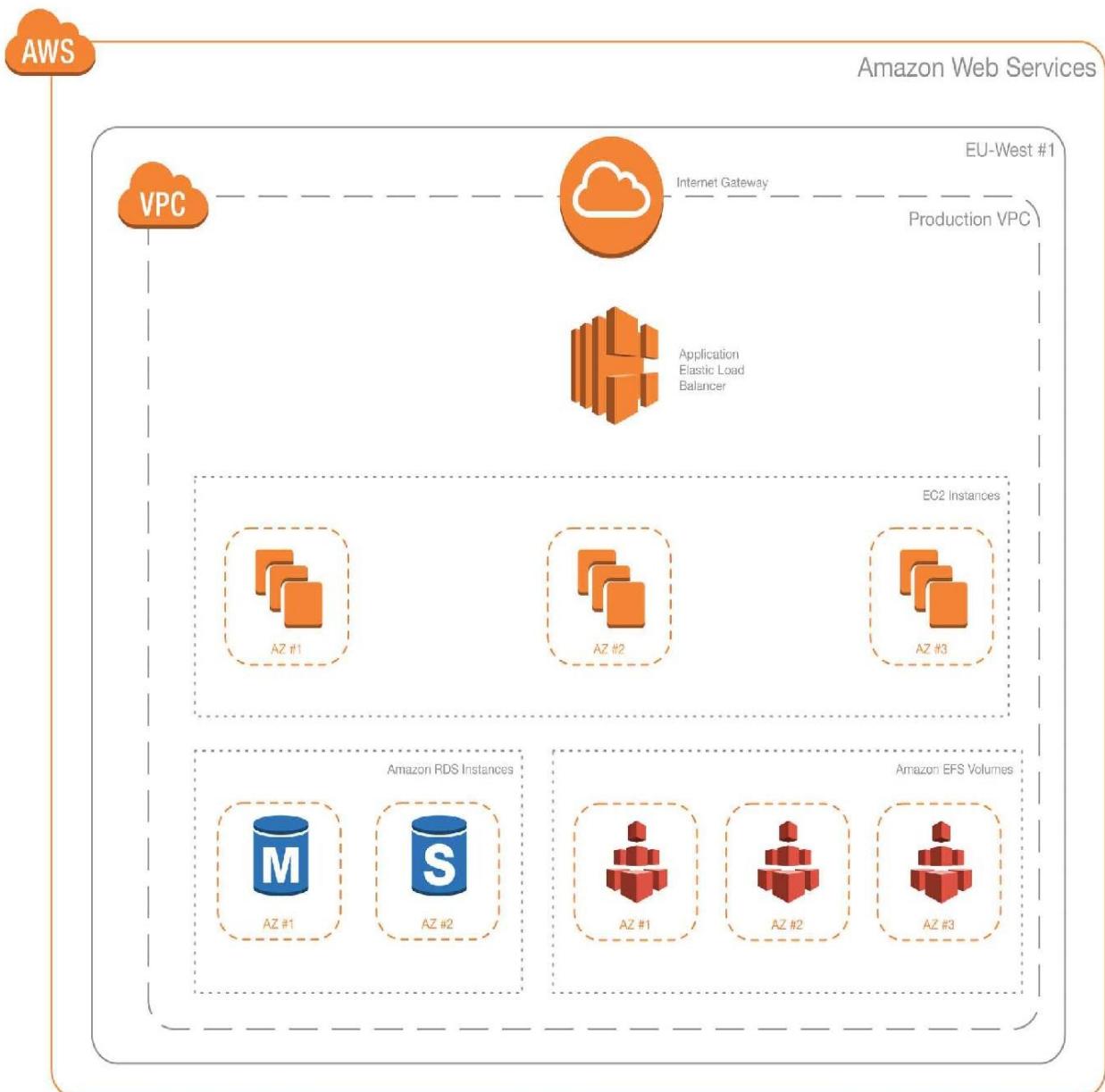
- Launching and configuring Amazon RDS (database)
- Launching and configuring Amazon EFS (shared storage)
- Launching and creating an **Amazon Machine Image (AMI)** (deploying the WordPress code)
- Launching and configuring a launch configuration and autoscaling group (high availability)

Technical requirements

As in the previous chapter, we are going to be using AWS; you will need the access key and secret key we created in the previous chapter to launch the resources needed for our highly available WordPress installation. Please note that we will be launching resources that incur charges. Again, you can find the complete playbook in the `Chapter10` folder of the accompanying GitHub repository at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter10/aws-wordpress>.

Planning the deployment

Before we dive into the playbooks, we should get an idea of what it is we are trying to achieve. As already mentioned, we are going to be building on our AWS VPC role by adding instances and storage. Our updated diagram looks as follows:



In the diagram, we have the following:

- 3 x EC2 instances (t2.micro), one in each availability zone
- 2 x RDS instances (t2.micro), in a master/standby multi-AZ configuration
- 5 GB of EFS storage across three availability zones

Before we talk about the deployment itself, based on the diagram and specifications here, how much is this deployment going to cost us to run?

Costing the deployment

The cost of running this deployment in the EU-West-1 region is as follows:

Instance type	# Instances	Total cost per hour	Total cost per day	Total cost per month
EC2 instances (t2.micro)	3	\$0.038	\$0.091	\$27.22
RDS instance (t2.micro) —Master and Standby	2	\$0.036	\$0.086	\$25.92
Application Load Balancer	1	\$0.033	\$0.80	\$23.90
5 GB EFS	1	\$0.002	£0.06	\$1.65
Total:		\$0.109	\$2.62	\$78.69

There will be a few other small costs, such as bandwidth and the cost of storing the AMI that contains our software stack. We could look at reducing these costs significantly by removing some of the redundancy, by disabling the multi-AZ RDS instance and also reducing the number of EC2 instances down to just one; however, this starts to introduce single points of failure into our deployment, which we do not want to do.

WordPress considerations and high availability

So far, we have been launching WordPress on a single server, which is fine, but as we are trying to remove as many of the single points of failure within our deployment as possible, this means that we have to put a little thought into how we initially configure and launch our deployment.

First of all, let's discuss the order that we will need to launch our deployment in. The basic order we will need to launch the elements in is as follows:

- **VPC, subnets, internet gateway, routing and security groups:** These are all needed to launch our deployment
- **The Application Elastic Load Balancer:** We will be using the public hostname of the Elastic Load Balancer for our installation, so this needs to be launched before we start our installation
- **The RDS database instance:** It is important that our database instance is available before we launch our installation as we need to create the WordPress database and bootstrap the installation
- **The EFS storage:** We need some storage to share between the EC2 instances we will be launching next

So far, so good; however, this is where we have to start taking WordPress into account.

As some of you may know from experience, the current version of WordPress is not really designed to be spread across multiple servers. There are plenty of hacks and workarounds we can apply to make WordPress play nicely in this sort of deployment; however, this chapter is not about the finer points of deploying WordPress. Instead, it is about how you can use Ansible to deploy a multi-tiered web application.

Because of this, we will be going for the most basic of the multi-instance WordPress options by deploying our code and content on the EFS volume. This means that all we have to do is install our LEMP stack. It should be noted that

this option is not the most performant, but it will serve our needs.

Now back to the list of tasks. When it comes to launching our instances, we need to do the following:

- Launch a temporary EC2 instance running CentOS 7 so that we can reuse parts of existing playbooks
- Update the operating system and install the software stack, supporting tools, and configuration needed for us to install and run our WordPress installation
- Mount the EFS volume and set the correct permissions, and configure it to mount on boot
- Attach the temporary instance to our load balancer, and install and configure WordPress
- Create an AMI from our temporary instance
- Create a launch configuration that uses the AMI we just created
- Create an autoscaling group and attach the launch configuration; it should also register our WordPress instances with the Elastic Load Balancer

During the initial playbook execution, there will be a short period of downtime as we create the AMI; further playbook runs should repeat the process with the existing instances up and running, and then, once the AMI is built, it should be deployed alongside the current instances, which will then be terminated once the new instances are registered with the Elastic Load Balancer and receiving traffic. This will allow us to update our operating system packages and configuration without any downtime—this will also simulate us deploying AMIs that have our code base baked in; more on that later in the chapter.

Now we have an idea of what we are trying to achieve, let's make a start on our playbook.

The playbook

The playbook is going to be split up into several sections. Before we make a start on the first one, let's create the folder structure. As per previous chapters, we simply need to run the following commands:

```
$ mkdir aws-wordpress aws-wordpress/group_vars aws-wordpress/roles  
$ touch aws-wordpress/production aws-wordpress/site.yml aws-  
wordpress/group_vars/common.yml
```

Now that we have our basic structure in place, we can make a start on creating the roles, starting with the network.

Amazon VPC

All the work for creating the underlying network was completed in the previous chapter, meaning that we simply need to copy the `elb`, `gateway`, `securitygroups`, `subnets`, and `vpc` folders from your previous playbook across to your current `roles` folder.

Once copied, update the `site.yml` file so it reads:

```
- name: Create and configure an Amazon VPC
  hosts: localhost
  connection: local
  gather_facts: True

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/vpc
    - roles/subnets
    - roles/gateway
    - roles/securitygroups
    - roles/elb
```

Also, add the following to the `group_vars/common.yml` file:

```
---
# the common variables

environment_name: "wordpress"
ec2_region: "eu-west-1"
```

Finally, we need to update the subnets that are being created; to do this, update the `the_subnets` variable in `roles/subnets/defaults/main.yml` to read:

```
the_subnets:
  - { use: 'ec2', az: 'a', subnet: '10.0.10.0/24' }
  - { use: 'ec2', az: 'b', subnet: '10.0.11.0/24' }
  - { use: 'ec2', az: 'c', subnet: '10.0.12.0/24' }
  - { use: 'elb', az: 'a', subnet: '10.0.20.0/24' }
  - { use: 'elb', az: 'b', subnet: '10.0.21.0/24' }
  - { use: 'elb', az: 'c', subnet: '10.0.22.0/24' }
  - { use: 'rds', az: 'a', subnet: '10.0.30.0/24' }
  - { use: 'rds', az: 'b', subnet: '10.0.31.0/24' }
  - { use: 'efs', az: 'a', subnet: '10.0.40.0/24' }
  - { use: 'efs', az: 'b', subnet: '10.0.41.0/24' }
  - { use: 'efs', az: 'c', subnet: '10.0.42.0/24' }
```

As you can see, we are adding an additional subnet for our EFS volume, making

it available in all three availability zones. More on why later. However, it does demonstrate the flexibility of our playbook, when all we have to do is add an additional line to our variables to create the additional subnet.

That completes the first part of the playbook; we can now move on to some new territory and launch our Amazon RDS instance.

Amazon RDS

Let's start by creating the file structure for the role by running:

```
| $ ansible-galaxy init roles/rds
```

Now that's in place, let's discuss what we need to do to launch the RDS instance. To start with, we need to define some default values; add the following to the `roles/rds/defaults/main.yml` file:

```
rds:  
  db_username: "{{ environment_name }}"  
  db_password: "{{ lookup('password', 'group_vars/rds_passwordfile  
  chars=ascii_letters,digits length=30') }}"  
  db_name: "{{ environment_name }}"  
  app_instance_type: "db.t2.micro"  
  engine: "mariadb"  
  hdd_size: "5"  
  no_of_backups: "7"  
  multi_az: "yes"  
  wait: "yes"  
  wait_time: "1200"
```

Some of the variables are self-explanatory, such as `db_username`, `db_password`, and `db_name`, although, as you can see, we are doing something interesting with the content of `db_password`. Rather than hardcoding a password, we are using a lookup plugin; these allow Ansible to read external data, for example, the contents of a file, Redis, MongoDB, or various APIs.

In our case, we are using the Ansible password lookup plugin to populate a file on our Ansible controller with a randomly generated password; this file is left alone on subsequent lookups, meaning that the password can be reused. Ansible will be generating a password that contains letters and numbers and is 30 characters long, and it is placing it in a file at `group_vars/rds_passwordfile`. This file is then added to the `.gitignore` file, so we don't end up shipping our passwords to version control.

Other things to note are that we are launching a db.t2.micro (`app_instance_type`) MariaDB (`engine`) instance, with 5 GB (`hdd_size`) of storage in a multi-AZ configuration (`multi_az`). We will be keeping 7 days of backups (`no_of_backups`), and when the instance first launches, we will wait (`wait`) for 20 minutes (`wait_time`) for

the instance to become available before moving on to the next part of the playbook.

There is one thing we need to do before we launch our RDS instance, and that is to create an RDS subnet group; this is how we associate our RDS instance with the subnets we created when launched the VPC. In `roles/rds/tasks/main.yml`, enter the following:

```
- name: create RDS subnet group
  rds_subnet_group:
    region: "{{ ec2_region }}"
    state: present
    name: "{{ environment_name }}_rds_group"
    description: "RDS Group for {{ environment_name }}"
    subnets: "{{ subnet_rds_ids }}"
```

This task uses the list of two subnets we registered in the `subnets` role to, in our case, create a group called `wordpress_rds_group`. When it comes to associating the subnet group with our RDS instance, we will be using its name rather than its unique ID, so there is no need for us to register the output of the task for later use. The next, and final, task in the role launches the RDS instance. Enter the following `rds_subnet_group` task:

```
- name: launch the rds instance
  rds:
    region: "{{ ec2_region }}"
    command: "create"
    instance_name: "{{ environment_name }}-rds"
    db_engine: "{{ rds.engine }}"
    size: "{{ rds.hdd_size }}"
    backup_retention: "{{ rds.no_of_backups }}"
    instance_type: "{{ rds.app_instance_type }}"
    multi_zone: "{{ rds.multi_az }}"
    subnet: "{{ environment_name }}_rds_group"
    vpc_security_groups: "{{ sg_rds.group_id }}"
    username: "{{ rds.db_username }}"
    password: "{{ rds.db_password }}"
    db_name: "{{ rds.db_name }}"
    wait: "{{ rds.wait }}"
    wait_timeout: "{{ rds.wait_time }}"
    tags:
      Name: "{{ environment_name }}-rds"
      Environment: "{{ environment_name }}"
```

Apart from the `command` option, everything else is populated using a variable—this means that if there is any part of the instance we want to change when reusing the role, we can simply override the default variables by copying them to our `group_vars/common.yml` file. There are several options for the `command` option you can choose when interacting with the RDS module, which are:

- `create`: This creates an RDS instance. If one already exists, the module will gather facts on it
- `replicate`: This creates a read-only replica of the RDS instance you pass to it
- `delete`: This deletes the RDS instance; you have the option to take a snapshot before the instance is deleted
- `facts`: Gathers information on the RDS instance
- `modify`: If you have changed any part of your configuration, then this will update your instance, either immediately or during the next scheduled maintenance window
- `promote`: This will promote one of your read-replicas to be the new master
- `snapshot`: This creates a manual snapshot of your RDS instance
- `reboot`: This reboots the named RDS instance
- `restore`: This creates a new RDS instance from a named snapshot

There are a few niggles with the current RDS module you might want to take into account. The biggest of which is that it currently only allows you to launch RDS instances backed with magnetic storage. It is possible to add a task that uses the AWS command-line tools to migrate the storage to general purpose SSD once the instance has launched; however, we will not be covering that here.

Also, Ansible does not yet support Amazon Aurora, even though it is listed as an option. Again, it is possible to create tasks that use the AWS command-line tools to create and configure an Aurora cluster, but if you want native Ansible support, you are currently out of luck.



Amazon Aurora is Amazon's own database engine, which allows you to run either your MySQL or PostgreSQL databases on top of Amazon's custom-built, SSD-based, fault-tolerant, and self-healing database storage clusters. This custom storage architecture allows you to scale your database to over 60 TB without disruption or the need to reorganize your datasets.

Work within the Ansible community is ongoing to refactor the RDS module to support custom storage options and also introduce native support for Aurora. However, this is very much a work in progress, which has not made its way into the current Ansible release (2.5 at the time of writing).

That is all we need for our RDS instance; we can move on to the next role.

Amazon EFS

There are only three tasks needed to create the EFS volumes; as with previous roles, we can use the `ansible-galaxy` command to create the folder and file structure:

```
| $ ansible-galaxy init roles/efs
```

Before we add the tasks, we need to add some default variables and a template, so add the following to `roles/efs/default/main.yml`:

```
efs:
  wait: "yes"
  wait_time: "1200"
```

Now, create a file in `roles/efs/templates` called `targets.j2`, which should contain:

```
---
efs_targets:
{% for item in subnet_efs_ids %}
  - subnet_id: "{{ item }}"
    security_groups: [ "{{ sg_efs.group_id }}" ]
{% endfor %}
```

As you can see, this template is looping over the `subnet_efs_ids` variable to create a list of subnet IDs and security groups under the variable name `efs_targets`; we will find out why this is needed shortly.

The first task in `roles/efs/tasks/main.yml` uses the `template` module to read the previous file to create a file and store it in the `group_vars` folder, and the second task loads the contents of the file using the `include_vars` module:

```
- name: generate the efs targets file
  template:
    src: "targets.j2"

  dest: "group_vars/generated_efs_targets.yml"
- name: load the efs targets
  include_vars: "group_vars/generated_efs_targets.yml"
```

Now that we have the `efs_targets` variable populated and loaded, we can add the third and final task; this task uses the `efs` module to create the volume:

```

- name: create the efs volume
  efs:
    region: "{{ ec2_region }}"
    state: present
    name: "{{ environment_name }}-efs"
    tags:
      Name: "{{ environment_name }}-efs"
      Environment: "{{ environment_name }}"
    targets: "{{ efs_targets }}"
    wait: "{{ efs.wait }}"
    wait_timeout: "{{ efs.wait_time }}"

```

"So, why go to the effort of creating a template, generating a file, and then loading the contents in when you could use `with_items`?" you may be asking yourself.

If we were to use `with_items`, then our task would look like the following:

```

- name: create the efs volume
  efs:
    region: "{{ ec2_region }}"
    state: present
    name: "{{ environment_name }}-efs"
    tags:
      Name: "{{ environment_name }}-efs"
      Environment: "{{ environment_name }}"
    targets:
      - subnet_id: "{{ item }}"
        security_groups: [ "{{ sg_efs.group_id }}" ]
    wait: "{{ efs.wait }}"
    wait_timeout: "{{ efs.wait_time }}"
  with_items: "{{ subnet_efs_ids }}"

```

This, at first glance, looks like it should work; however, if we take a look at an example of what `group_vars/generated_efs_targets.yml` looks like once it is has been generated, you may notice one important difference:

```

  efs_targets:
    - subnet_id: "subnet-0ce64b6a"
      security_groups: [ "sg-695f8b14" ]
    - subnet_id: "subnet-2598747f"
      security_groups: [ "sg-695f8b14" ]
    - subnet_id: "subnet-ee3487a6"
      security_groups: [ "sg-695f8b14" ]

```

As you can see from the example, we have three sections, each with the `subnet_id` unique to an availability zone. If we were to use `with_items`, we would only have one section and the task would be executed three times, each time overwriting the previous targets. Sure, we could have hardcoded three targets, but then what if we decided to reuse the role in a region that only has two availability zones, or one that has four? Hardcoding would mean we would lose the flexibility to have

Ansible dynamically adapt to situations where there is a range of dynamic results depending on what is being targeted.

Now we have our EFS role complete and the basics finished. Before we start to launch EC2 instances, we can look at testing our playbook.

Testing the playbook

As mentioned, now would be a good time to test the roles we have completed to make sure they are working as expected. To do this, open the `site.yml` file and add the following content:

```
---
- name: Create, launch and configure our basic AWS environment
  hosts: localhost
  connection: local
  gather_facts: True

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/vpc
    - roles/subnets
    - roles/gateway
    - roles/securitygroups
    - roles/elb
    - roles/rds
    - roles/efs
```

Before we run our playbook, we will need to set the `AWS_ACCESS_KEY` and `AWS_SECRET_KEY` environment variables; to do this, run the following, replacing the value of each variable with the details that we generated in the previous chapter:

```
| $ export AWS_ACCESS_KEY=AKIAI5KECPOTNTTVM3EDA
| $ export AWS_SECRET_KEY=Y4B7FFiSWl0Am3VIFc07lgnc/TAtK5+RpxzIGTr
```

We will want to time our playbook run. To do this, we can prefix our `ansible-playbook` command with `time`, which means the command we need to run looks like:

```
| $ time ansible-playbook -i production site.yml
```

Don't forget that we have told Ansible to wait for a maximum of 20 minutes before launching the RDS instance and creating the EFS volume, so the initial playbook run may take a little time.

The reason for this is that when the RDS instance is launched, it is first created, then cloned to a standby server, and then, finally, an initial backup is made. Only once these steps have been completed is the RDS instance marked as ready and

our playbook run progresses. Also, for the EFS volumes, we are creating a cluster of three volumes across three availability zones, so it takes a little while to configure them:

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the
implicit
localhost does not match 'all'

PLAY [Create, launch and configure our basic AWS environment]
*****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [roles/vpc : ensure that the VPC is present]
*****
changed: [localhost]

TASK [roles/subnets : ensure that the subnets are present]
*****
changed: [localhost] => (item={u'subnet': u'10.0.10.0/24', u'use': u'ec2', u'az': u'a'})
changed: [localhost] => (item={u'subnet': u'10.0.11.0/24', u'use': u'ec2', u'az': u'b'})
changed: [localhost] => (item={u'subnet': u'10.0.12.0/24', u'use': u'ec2', u'az': u'c'})
changed: [localhost] => (item={u'subnet': u'10.0.20.0/24', u'use': u'elb', u'az': u'a'})
changed: [localhost] => (item={u'subnet': u'10.0.21.0/24', u'use': u'elb', u'az': u'b'})
changed: [localhost] => (item={u'subnet': u'10.0.22.0/24', u'use': u'elb', u'az': u'c'})
changed: [localhost] => (item={u'subnet': u'10.0.30.0/24', u'use': u'rds', u'az': u'a'})
changed: [localhost] => (item={u'subnet': u'10.0.31.0/24', u'use': u'rds', u'az': u'b'})
changed: [localhost] => (item={u'subnet': u'10.0.40.0/24', u'use': u'efs', u'az': u'a'})
changed: [localhost] => (item={u'subnet': u'10.0.41.0/24', u'use': u'efs', u'az': u'b'})
changed: [localhost] => (item={u'subnet': u'10.0.42.0/24', u'use': u'efs', u'az': u'c'})

TASK [roles/subnets : gather information about the ec2 subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the elb subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the rds subnets]
*****
ok: [localhost]

TASK [roles/subnets : gather information about the efs subnets]
*****
ok: [localhost]

TASK [roles/subnets : register just the IDs for each of the subnets]
```

```
*****
ok: [localhost]

TASK [roles/gateway : ensure that there is an internet gateway]
*****
changed: [localhost]

TASK [roles/gateway : check that we can route through internet gateway]
*****
changed: [localhost]

TASK [roles/securitygroups : provision elb security group]
*****
changed: [localhost]

TASK [roles/securitygroups : find out your current public IP address using
https://ipify.org/] *****
ok: [localhost]

TASK [roles/securitygroups : set your public ip as a fact]
*****
ok: [localhost]

TASK [roles/securitygroups : provision ec2 security group]
*****
changed: [localhost]

TASK [roles/securitygroups : provision rds security group]
*****
changed: [localhost]

TASK [roles/securitygroups : provision efs security group]
*****
changed: [localhost]

TASK [roles/elb : provision the target group]
*****
changed: [localhost]

TASK [roles/elb : provision an application elastic load balancer]
*****
changed: [localhost]

TASK [roles/rds : create RDS subnet group]
*****
changed: [localhost]

TASK [roles/rds : launch the rds instance]
*****
changed: [localhost]

TASK [roles/efs : generate the efs targets file]
*****
changed: [localhost]

TASK [roles/efs : load the efs targets]
*****
ok: [localhost]

TASK [roles/efs : create the efs volume]
*****
changed: [localhost]

PLAY RECAP
```

```
*****
| localhost : ok=23 changed=14 unreachable=0 failed=0
```

As you can see from the output, the playbook run executed as expected. We can check the AWS console to make sure everything has been created, starting with the VPC:

The screenshot shows the AWS VPC console interface. At the top, there's a navigation bar with the AWS logo, 'Services', 'Resource Groups', and user information ('Russell McKendrick', 'Ireland', 'Support'). Below the navigation bar is a search bar with the placeholder 'Search VPCs and their properties'. A table lists two VPCs:

Name	VPC ID	State	IPv4 CIDR	IPv6 CIDR	DHCP options set	Route table	Network ACL	Tenancy
vpc-205dff47	available	172.16.0.0/16			dopt-44851321	rtb-a3a19bc4	acl-b999eade	Default
wordpress	available	10.0.0.0/16			dopt-44851321	rtb-0e289377	acl-1f859779	Default

Below the table, the 'wordpress' VPC is selected, and its details are shown in a summary card:

VPC ID: vpc-7596f013 wordpress	Network ACL: acl-1f859779
State: available	Tenancy: Default
IPv4 CIDR: 10.0.0.0/16	DNS resolution: yes
IPv6 CIDR:	DNS hostnames: yes
DHCP options set: dopt-44851321	ClassicLink DNS Support: no
Route table: rtb-0e289377	
ClassicLink: Disabled	

At the bottom of the screen, there are links for 'Feedback', 'English (US)', and copyright information: '© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use'.

Then, check the Elastic Load Balancer, which can be found in the EC2 section:

The screenshot shows the AWS Elastic Load Balancing (ELB) console. At the top, there's a navigation bar with the AWS logo, 'Services' dropdown, 'Resource Groups' dropdown, and user information ('Russell Mckendrick', 'Ireland', 'Support'). Below the navigation is a toolbar with 'Create Load Balancer' (blue button), 'Actions' dropdown, and three icons (refresh, gear, question mark). A search bar with placeholder text 'Filter by tags and attributes or search by keyword' is followed by a pagination indicator '1 to 1 of 1'. A table lists one load balancer entry:

Name	DNS name	State	VPC ID	Availability Zones	Type
wordpress-elb	wordpress-elb-441525357.e...	active	vpc-7596f013	eu-west-1c, eu-west-1a...	application

Below the table, a section titled 'Load balancer: wordpress-elb' contains tabs for 'Description' (selected), 'Listeners', 'Monitoring', and 'Tags'. The 'Basic Configuration' section displays various details about the load balancer:

Name:	wordpress-elb	Creation time:	April 29, 2018 at 3:07:44 PM UTC+1
ARN:	arn:aws:elasticloadbalancing:eu-west-1:687011238589:loadbalancer/app/wordpress-elb/08d4de808d7c35b9	Hosted zone:	Z32O12XQLNTSW2
		State:	active
DNS name:	wordpress-elb-441525357.eu-west-1.elb.amazonaws.com	VPC:	vpc-7596f013
name:	(A Record)	IP address type:	ipv4
Scheme:	internet-facing	AWS WAF Web ACL:	
Type:	application		
Availability Zones:	subnet-029b7758 - eu-west-1c, subnet-80f954e6 - eu-west-1a, subnet-c227948a - eu-west-1b		

[Edit availability zones](#)

At the bottom, there are links for 'Feedback', 'English (US)', and legal notices: '© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.' and 'Privacy Policy Terms of Use'.

We can also check that our RDS instance is up and running:

The screenshot shows the AWS RDS console for the 'wordpress-rds' instance. The top navigation bar includes the AWS logo, Services, Resource Groups, and user information (Russell Mckendrick, Ireland, Support). The left sidebar shows the hierarchy: RDS > Instances > wordpress-rds. The main content area displays the instance details and monitoring metrics.

Summary

Engine	DB instance class Info	DB instance status	Pending maintenance
MariaDB 10.1.31	db.t2.micro	available	none

CloudWatch (54)

Legend: [wordpress-rds](#)

Monitoring ▾ Last Hour ▾

CPU Utilization (Percent)

04/29 14:30 04/29 15:00

DB Connections (Count)

04/29 14:30 04/29 15:00

Free Storage Space (MB)

04/29 14:30 04/29 15:00

[Feedback](#) [English \(US\)](#)

© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved. [Privacy Policy](#) [Terms of Use](#)

Then, the final part of our playbook is the EFS volume:

Screenshot of the AWS CloudFormation console showing the creation of a new AWS Lambda function named "HelloWorld".

The Lambda function has the following configuration:

- Function name:** HelloWorld
- Description:** A simple Lambda function that prints "Hello World" to the CloudWatch logs.
- Runtime:** Node.js 6.10
- Memory size:** 128 MB
- Timeout:** 3 seconds
- Role:** arn:aws:lambda:eu-west-1:687011238589:role/HelloWorldExecutionRole
- Code:** zip file uploaded from local machine.
- Environment variables:** None
- Triggers:** None

The Lambda function is currently in the **Creating** state.

When I ran the playbook, it took just over 18 minutes, as you can see from the following output:

```
1. aws-wordpress (bash)
changed: [localhost]

PLAY RECAP ****
localhost : ok=23    changed=14    unreachable=0    failed=0

real    18m20.106s
user    1m8.357s
sys     0m20.363s
russ in ~/aws-wordpress
⚡
```

As expected, the majority of that time was Ansible waiting for the RDS instance and the EFS volume to be ready.

Now that we know that the playbook can launch our basic infrastructure without error, we can proceed with the rest of playbook. Or can we?

Terminating resources

As already mentioned at the start of this chapter, we are launching resources that are going to incur costs when they are up and running. As we are still writing our playbook, we don't want the resource to sit idle and rack up costs while we work, so let's create a supporting playbook that undoes everything we have just ran.

To do this, let's create a single role called `remove`:

```
| $ ansible-galaxy init roles/remove
```

This role will use Ansible to, well, remove everything we have just launched, thus keeping costs down while we are developing our playbook. First of all, we need to add some default variables to `roles/remove/defaults/main.yml`; these are:

```
wait:
  wait: "yes"
  wait_time: "1200"
vpc_cidr_block: "10.0.0.0/16"
```

The `vpc_cidr_block` variable should match your VPC CIDR. Now, we can make a start on adding the tasks to `roles/remove/tasks/main.yml`, which removes everything we have launched. We will be working our way backwards as each of the resources were launched in a certain order, meaning that we need to remove them in reverse order. So let's start with the EFS volume:

```
- name: remove the efs shares
  efs:
    region: "{{ ec2_region }}"
    state: absent
    name: "{{ environment_name }}-efs"
    wait: "{{ wait.wait }}"
    wait_timeout: "{{ wait.wait_time }}"
```

We only have to provide a few details this time as the volume is already present; we need to give it the name of the volume and also a `state` of `absent`. You will notice that we wait for the volume to be removed completely before continuing. We are going to have quite a few pauses in this playbook to allow for the resources to be fully unregistered with the AWS API before we move on to the next task.

The next few tasks deal with the removal of the RDS instance and the RDS subnet group:

```
- name: terminate the rds instance
  rds:
    region: "{{ ec2_region }}"
    command: "delete"
    instance_name: "{{ environment_name }}-rds"
    wait: "{{ wait.wait }}"
    wait_timeout: "{{ wait.wait_time }}"

- name: wait for 2 minutes before continuing
  pause:
    minutes: 2

- name: remove RDS subnet group
  rds_subnet_group:
    region: "{{ ec2_region }}"
    state: absent
    name: "{{ environment_name }}_rds_group"
```

As you can see, we have a pause, using the `pause` module, of 2 minutes between the RDS instance being terminated and the removal of the RDS subnet group. If we remove this pause, then we run the risk of the RDS instance not being fully unregistered, meaning that we would not be able to remove the subnet group, which would result in an error in the playbook.

If, at any stage, the playbook throws an error, we should be able to run it a second time and it should pick up where it left off. Although, there is a point of no return when we will not be able to run the playbook at all; I will let you know when this is.

Now that the RDS instance and subnet group have been removed, we can remove the Elastic Load Balancer:

```
- name: terminate the application elastic load balancer
  elb_application_lb:
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}-elb"
    state: "absent"

- name: prompt
  pause:
    prompt: "Make sure the elastic load balancer has been terminated before proceeding"
```

You will notice that, this time, although the `pause` module is being used again, we are not providing a period of time. Instead, we are instructing the user to check the AWS console, then to press a key once the Elastic Load Balancer has been removed. This is because the `elb_application_lb` module doesn't support waiting

around for the resource to be removed.

The next task will immediately fail if you just hit *Enter* when the resource is in the process of being removed, hence the need for the manual check. The task removes the ELB target group:

```
- name: remove the target group
  elb_target_group:
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}-target-group"
    state: "absent"
```

The tasks that follow remove the security groups; as we have groups that reference other groups, there is a 30-second `pause` before we remove the next group in line:

```
- name: remove the efs security group
  ec2_group:
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}-efs"
    state: "absent"

- name: wait for 30 seconds before continuing
  pause:
    seconds: 30

- name: remove the rds security group
  ec2_group:
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}-rds"
    state: "absent"

- name: wait for 30 seconds before continuing
  pause:
    seconds: 30

- name: remove the ec2 security group
  ec2_group:
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}-ec2"
    state: "absent"

- name: wait for 30 seconds before continuing
  pause:
    seconds: 30

- name: remove the elb security group
  ec2_group:
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}-elb"
    state: "absent"

- name: wait for 30 seconds before continuing
  pause:
    seconds: 30
```

Again, as you can see, we only have to provide the group name and the `state` of `absent`. The next task, the removal of the route table, requires a little more than just the name:

```
- name: get some facts on the route table
  ec2_vpc_route_table_facts:
    region: "{{ ec2_region }}"
    filters:
      "tag:Name": "{{ environment_name }}_outbound"
      "tag:Environment": "{{ environment_name }}"
  register: route_table_facts

- name: remove the route table
  ec2_vpc_route_table:
    region: "{{ ec2_region }}"
    vpc_id: "{{ route_table_facts.route_tables[0].vpc_id }}"
    route_table_id: "{{ route_table_facts.route_tables[0].id }}"
    lookup: "id"
    state: "absent"
    ignore_errors: yes
```

To remove the route table, we need to know the VPC ID and also the route table's ID. To find out this information, we are using the `ec2_vpc_route_table_facts` module to gather the data based on the `Name` and `Environment` tags, so we only remove what we intend to. This is information that is then passed to the `ec2_vpc_route_table` module, which we are instructing to use the ID of the route table to do the `lookup`.

We are also telling Ansible to ignore any errors generated here. The reason being that if a subsequent task throws an error and we need to rerun the playbook, we will need it to progress past this point in the playbook run, and if this task has successfully run, it won't be able to as there will be nothing to remove, which itself will generate an error.

The next two tasks gather information on the VPC and remove the internet gateway:

```
- name: get some facts on the vpc
  ec2_vpc_net_facts:
    region: "{{ ec2_region }}"
    filters:
      "tag:Name": "{{ environment_name }}"
      "tag:Environment": "{{ environment_name }}"
  register: vpc_facts

- name: ensure that there isn't an internet gateway
  ec2_vpc_igw:
    region: "{{ ec2_region }}"
    state: "absent"
```

```

    vpc_id: "{{ vpc_facts.vpcs[0].vpc_id }}"
    tags:
      "Name": "{{ environment_name }}_internet_gateway"
      "Environment": "{{ environment_name }}"
  ignore_errors: yes

```

Again, we are ignoring any errors generated so that we can progress with the playbook run should it need to be executed more than once. The task gathers information on the subnets that are active in the environment using the `ec2_vpc_subnet_facts` module; we then register this information as `the_subnets`:

```

- name: gather information about the subnets
  ec2_vpc_subnet_facts:
    region: "{{ ec2_region }}"
  filters:
    "tag:Environment": "{{ environment_name }}"
  register: the_subnets

```

Once we have information on the subnets, we can remove them using their CIDR block and by setting the `state` to `absent`:

```

- name: ensure that the subnets are absent
  ec2_vpc_subnet:
    region: "{{ ec2_region }}"
    state: "absent"
    vpc_id: "{{ vpc_facts.vpcs[0].vpc_id }}"
    cidr: "{{ item.cidr_block }}"
  with_items: "{{ the_subnets.subnets }}"

```

 It is at this point that the playbook would generate an error if you were to run it more than once and make it this far. If it does, you can remove the VPC manually.

Finally, now that we have removed all of the contents from our VPC and it is empty which means we can remove the VPC itself without error:

```

- name: ensure that the VPC is absent
  ec2_vpc_net:
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}"
    state: "absent"
    cidr_block: "{{ vpc_cidr_block }}"

```

Now that we have our role completed, we can create a playbook called `remove.yml`, which contains the following:

```

---
- name: Terminate everything in our basic AWS environment
  hosts: localhost
  connection: local
  gather_facts: True
  vars_files:

```

```
- group_vars/common.yml  
roles:  
  - roles/remove
```

We now have all the pieces in place to remove our AWS environment; to do this, run the following command:

| \$ time ansible-playbook -i production remove.yml



Don't forget to check that the Elastic Load Balancer has been removed and press any key to continue during the playbook run. Otherwise, you will be waiting around for a while.

When I ran the playbook, it took just under 12 minutes:

```
1. aws-wordpress (bash)  
TASK [roles/remove : ensure that the VPC is absent] ****  
changed: [localhost]  
  
PLAY RECAP ****  
localhost : ok=23    changed=13    unreachable=0    failed=0  
  
real    11m32.724s  
user    0m47.679s  
sys     0m15.653s  
ruess in ~/aws-wordpress  
⚡
```

If you are not following along with the output of the playbook, you can see all of the pauses and the information on the subnets collected by the `ec2_vpc_subnet_facts` module here:

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the  
implicit  
localhost does not match 'all'  
  
PLAY [Terminate everything in our basic AWS environment]  
*****  
  
TASK [Gathering Facts]  
*****  
ok: [localhost]  
  
TASK [roles/remove : remove the efs shares]  
*****  
changed: [localhost]  
  
TASK [roles/remove : terminate the rds instance]  
*****  
changed: [localhost]  
  
TASK [roles/remove : wait for 2 minutes before continuing]  
*****  
Pausing for 120 seconds  
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
```

```
ok: [localhost]

TASK [roles/remove : remove RDS subnet group]
*****
changed: [localhost]

TASK [roles/remove : terminate the application elastic load balancer]
*****
changed: [localhost]

TASK [roles/remove : prompt]
*****
[roles/remove : prompt]
Make sure the elastic load balancer has been terminated before proceeding:

ok: [localhost]

TASK [roles/remove : remove the target group]
*****
changed: [localhost]

TASK [roles/remove : remove the efs security group]
*****
changed: [localhost]

TASK [roles/remove : wait for 30 seconds before continuing]
*****
Pausing for 30 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [localhost]

TASK [roles/remove : remove the rds security group]
*****
changed: [localhost]

TASK [roles/remove : wait for 30 seconds before continuing]
*****
Pausing for 30 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [localhost]

TASK [roles/remove : remove the ec2 security group]
*****
changed: [localhost]

TASK [roles/remove : wait for 30 seconds before continuing]
*****
Pausing for 30 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [localhost]

TASK [roles/remove : remove the elb security group]
*****
changed: [localhost]

TASK [roles/remove : wait for 30 seconds before continuing]
*****
Pausing for 30 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [localhost]

TASK [roles/remove : get some facts on the route table]
*****
ok: [localhost]
```

```

TASK [roles/remove : remove the route table]
*****
changed: [localhost]

TASK [roles/remove : get some facts on the vpc]
*****
ok: [localhost]

TASK [roles/remove : ensure that there isn't an internet gateway]
*****
changed: [localhost]

TASK [roles/remove : gather information about the subnets]
*****
ok: [localhost]

TASK [roles/remove : ensure that the subnets are absent]
*****
changed: [localhost] => (item={'availability_zone': 'eu-west-1b', 'subnet_id': 'subnet-50259618', 'assign_ipv6_address_on_creation': False, 'tags': {'Environment': 'wordpress', 'Use': 'rds', 'Name': 'wordpress_rds_eu-west-1b'}, 'default_for_az': False, 'state': 'available', 'ipv6_cidr_block_association_set': [], 'vpc_id': 'vpc-7596f013', 'cidr_block': '10.0.31.0/24', 'available_ip_address_count': 251, 'id': 'subnet-50259618', 'map_public_ip_on_launch': False})
changed: [localhost] => (item={'availability_zone': 'eu-west-1a', 'subnet_id': 'subnet-80f954e6', 'assign_ipv6_address_on_creation': False, 'tags': {'Environment': 'wordpress', 'Use': 'elb', 'Name': 'wordpress_elb_eu-west-1a'}, 'default_for_az': False, 'state': 'available', 'ipv6_cidr_block_association_set': [], 'vpc_id': 'vpc-7596f013', 'cidr_block': '10.0.20.0/24', 'available_ip_address_count': 251, 'id': 'subnet-80f954e6', 'map_public_ip_on_launch': False})
changed: [localhost] => (item={'availability_zone': 'eu-west-1c', 'subnet_id': 'subnet-499f7313', 'assign_ipv6_address_on_creation': False, 'tags': {'Environment': 'wordpress', 'Use': 'ec2', 'Name': 'wordpress_ec2_eu-west-1c'}, 'default_for_az': False, 'state': 'available', 'ipv6_cidr_block_association_set': [], 'vpc_id': 'vpc-7596f013', 'cidr_block': '10.0.12.0/24', 'available_ip_address_count': 251, 'id': 'subnet-499f7313', 'map_public_ip_on_launch': False})
changed: [localhost] => (item={'availability_zone': 'eu-west-1a', 'subnet_id': 'subnet-74fc5112', 'assign_ipv6_address_on_creation': False, 'tags': {'Environment': 'wordpress', 'Use': 'ec2', 'Name': 'wordpress_ec2_eu-west-1a'}, 'default_for_az': False, 'state': 'available', 'ipv6_cidr_block_association_set': [], 'vpc_id': 'vpc-7596f013', 'cidr_block': '10.0.10.0/24', 'available_ip_address_count': 251, 'id': 'subnet-74fc5112', 'map_public_ip_on_launch': False})
changed: [localhost] => (item={'availability_zone': 'eu-west-1b', 'subnet_id': 'subnet-9f3a89d7', 'assign_ipv6_address_on_creation': False, 'tags': {'Environment': 'wordpress', 'Use': 'ec2', 'Name': 'wordpress_ec2_eu-west-1b'}, 'default_for_az': False, 'state': 'available', 'ipv6_cidr_block_association_set': [], 'vpc_id': 'vpc-7596f013', 'cidr_block': '10.0.11.0/24', 'available_ip_address_count': 251, 'id': 'subnet-9f3a89d7', 'map_public_ip_on_launch': False})
changed: [localhost] => (item={'availability_zone': 'eu-west-1c', 'subnet_id': 'subnet-8e967ad4', 'assign_ipv6_address_on_creation': False, 'tags': {'Environment': 'wordpress', 'Use': 'efs', 'Name': 'wordpress_efs_eu-west-1c'}, 'default_for_az': False, 'state': 'available', 'ipv6_cidr_block_association_set': [], 'vpc_id': 'vpc-7596f013', 'cidr_block': '10.0.42.0/24', 'available_ip_address_count': 251, 'id': 'subnet-8e967ad4', 'map_public_ip_on_launch': False})
changed: [localhost] => (item={'availability_zone': 'eu-west-1a', 'subnet_id': 'subnet-d7fe53b1', 'assign_ipv6_address_on_creation': False, 'tags': {'Environment': 'wordpress', 'Use': 'efs', 'Name': 'wordpress_efs_eu-west-1a'}})

```

```

u'default_for_az': False, u'state': u'available', u'ipv6_cidr_block_association_set':
[], u'vpc_id': u'vpc-7596f013', u'cidr_block': u'10.0.40.0/24',
u'available_ip_address_count': 251, u'id': u'subnet-d7fe53b1',
u'map_public_ip_on_launch': False})
changed: [localhost] => (item={u'availability_zone': u'eu-west-1c', u'subnet_id':
u'subnet-029b7758', u'assign_ipv6_address_on_creation': False, u'tags':
{u'Environment': u'wordpress', u'Use': u'elb', u'Name': u'wordpress_elb_eu-west-1c'},
u'default_for_az': False, u'state': u'available', u'ipv6_cidr_block_association_set':
[], u'vpc_id': u'vpc-7596f013', u'cidr_block': u'10.0.22.0/24',
u'available_ip_address_count': 251, u'id': u'subnet-029b7758',
u'map_public_ip_on_launch': False})
changed: [localhost] => (item={u'availability_zone': u'eu-west-1a', u'subnet_id':
u'subnet-ede5488b', u'assign_ipv6_address_on_creation': False, u'tags':
{u'Environment': u'wordpress', u'Use': u'rds', u'Name': u'wordpress_rds_eu-west-1a'},
u'default_for_az': False, u'state': u'available', u'ipv6_cidr_block_association_set':
[], u'vpc_id': u'vpc-7596f013', u'cidr_block': u'10.0.30.0/24',
u'available_ip_address_count': 251, u'id': u'subnet-ede5488b',
u'map_public_ip_on_launch': False})
changed: [localhost] => (item={u'availability_zone': u'eu-west-1b', u'subnet_id':
u'subnet-ec3e8da4', u'assign_ipv6_address_on_creation': False, u'tags':
{u'Environment': u'wordpress', u'Use': u'efs', u'Name': u'wordpress_efs_eu-west-1b'},
u'default_for_az': False, u'state': u'available', u'ipv6_cidr_block_association_set':
[], u'vpc_id': u'vpc-7596f013', u'cidr_block': u'10.0.41.0/24',
u'available_ip_address_count': 251, u'id': u'subnet-ec3e8da4',
u'map_public_ip_on_launch': False})
changed: [localhost] => (item={u'availability_zone': u'eu-west-1b', u'subnet_id':
u'subnet-c227948a', u'assign_ipv6_address_on_creation': False, u'tags':
{u'Environment': u'wordpress', u'Use': u'elb', u'Name': u'wordpress_elb_eu-west-1b'},
u'default_for_az': False, u'state': u'available', u'ipv6_cidr_block_association_set':
[], u'vpc_id': u'vpc-7596f013', u'cidr_block': u'10.0.21.0/24',
u'available_ip_address_count': 251, u'id': u'subnet-c227948a',
u'map_public_ip_on_launch': False})

TASK [roles/remove : ensure that the VPC is absent]
*****
changed: [localhost]

PLAY RECAP
*****
localhost : ok=23 changed=13 unreachable=0 failed=0

```

I would recommend double-checking that the resources have gone from your AWS console, as no one likes a surprise bill. Now that we have completed and executed our `remove` playbook, so that we don't incur any unnecessary costs, we can continue to build out our highly available WordPress installation.

EC2 instances

Now that we have all the basic services required for our WordPress installation to consume, we can make a start on deploying the compute resource to install WordPress on. This is where things get interesting, as we have to build logic into our playbook so that if our site is up and running, we can deploy updates to the operating system and roll out new images without any downtime.

But if it is a new deployment, we need to launch an instance, attach it to the Elastic Load Balancer, install the software stack, configure WordPress, and create an image we can then use in a launch configuration, which we will need to attach to an autoscaling group.

While this may seem complicated, building this logic into the playbook will make it a lot easier to maintain and hand over to someone else to manage/run, as they will not need to worry about the existing deployment, they will just need to run the playbook.

Instance discovery

We are going to simply call this role EC2, so we need to run the following command to create the role structure:

```
| $ ansible-galaxy init roles/ec2
```

The primary goal of this role is to ensure that by the end of its execution, we have an instance, either a new or an existing one, that we can then use in the forthcoming roles to base an AMI on.

The defaults in `roles/ec2/defaults/main.yml` define which image we want to use if our role discovers that this is a new deployment. For our installation, we are going to be using the AMI provided by CentOS in the AWS Marketplace; this means we can reuse large chunks of our WordPress installation playbook:

```
image:
  base: "CentOS Linux 7 x86_64"
  owner: "679593333241"
  root_device: "ebs"
  architecture: "x86_64"
  wait_port: "22"
  ec2_instance_type: "t2.micro"
```

We will go into a little more detail about why we need this information when we come to use images. Now we have the defaults in place, we can move on to the tasks in `roles/ec2/tasks/main.yml`.

When we launch our instances using the autoscaling group, they will all be named `wordpress_ec2`, so the first thing our EC2 role has to do is figure out whether we have any running instances. To do this, we will use the `ec2_instance_facts` module to gather information on any instances that are running and are tagged with the name `wordpress_ec2`:

```
- name: gather facts on any already running instances
  ec2_instance_facts:
    region: "{{ ec2_region }}"
    filters:
      instance-state-name: "running"
      "tag:environment": "{{ environment_name }}"
      "tag:Name": "{{ environment_name }}-ec2"
  register: running_instances
```

Although we now have information on any instances that are already running, it is not really in a format we can use, so let's add the results to a host group called `already_running`:

```
- name: add any already running instances to a group
  add_host:
    name: "{{ item.public_dns_name }}"
    ansible_ssh_host: "{{ item.public_dns_name }}"
    groups: "already_running"
  with_items: "{{ running_instances.instances }}"
```

Now, we are left with a host group called `already_running`, which may contain from zero to three hosts; we now need to count the number of hosts in the group and set a fact that contains the number of hosts:

```
- name: set the number of already running instances as a fact
  set_fact:
    number_of_running_hosts: "{{ groups['already_running'] | length | default(0) }}"
```

Here, we are using the inbuilt Ansible variable `groups` along with our group name; now we have a list of hosts, we can count the number of items in the list by using the `length` filter. Finally, we are saying that if the list is empty, then the default value should be `0`.

Now we have a variable that contains `number_of_running_hosts`, we can now make some decisions on what we need to do next.

First, if `number_of_running_hosts` is `0`, then we are working on a new deployment, and we should run the tasks that launch a fresh EC2 instance:

```
- name: run the tasks for a new deployment
  include_tasks: "new_deployment.yml"
  when: number_of_running_hosts|int == 0
```

Or, if `number_of_running_hosts` is more than `1`, then we need to choose an already running instance to work with:

```
- name: run the tasks for an existing deployment
  include_tasks: "existing_deployment.yml"
  when: number_of_running_hosts|int >= 1
```

Let's take a look at these tasks, starting with what happens during a new deployment.

New deployment

If we are working on a new deployment, then we need to perform the following tasks:

1. Find the latest CentOS 7 AMI in the region we are using
2. Upload a copy of our public key so that we can use it to SSH into the instance
3. Launch an instance using the previous information
4. Add the new instance to a host group
5. Wait until SSH is available
6. Add our instance to the Elastic Load Balancer

All of these tasks are defined in `roles/ec2/tasks/new_deployment.yml`, so let's start working on these tasks by looking at how we can find the correct AMI to use.

We can't simply supply an AMI ID here as each region has a different ID, and also each AMI is regularly updated to make sure it is patched. To get around this, we can run the following tasks:

```
- name: search for all of the AMIs in the defined region which match our selection
  ec2_ami_facts:
    region: "{{ ec2_region }}"
    owners: "{{ image.owner }}"
    filters:
      name: "{{ image.base }}"
      architecture: "{{ image.architecture }}"
      root-device-type: "{{ image.root_device }}"
  register: amiFind

- name: filter the list of AMIs to find the latest one with an EBS backed volume
  set_fact:
    amiSortFilter: "{{ amiFind.images | sort(attribute='creation_date') | last }}"

- name: finally grab AMI ID of the most recent result which matches our base image
  which is backed by an EBS volume
  set_fact:
    our_ami_id: "{{ amiSortFilter.image_id }}"
```

As you can see, we are first looking for all `x86_64` AMIs created by CentOS that have `centos Linux 7 x86_64*` in the name and also use **Elastic Block Store (EBS)**-backed storage. This will give us details about several AMIs, which we have registered as `amiFind`.

Next, we need to filter the list of AMIs down to just the latest one, so we set a fact called `amiSortFilter`. Here, it is taking the list of images, `amiFind`, and sorting them by the date they were created. We then take just the information for the last AMI in the list to register as `amiSortFilter`. Finally, we reduce the information down more by setting a fact called `our_ami_id`, which is the `image_id` in the `amiSortFilter` variable, leaving us with just the information we need.

Now we know the AMI ID, we need to ensure that there is an SSH key we can use so that we can access the instance when launched. First of all, let's check that your user on the Ansible controller has an SSH key; if we can't find one, then one will be created:

```
- name: check the user {{ ansible_user_id }} has a key, if not create one
  user:
    name: "{{ ansible_user_id }}"
    generate_ssh_key: yes
    ssh_key_file: "~/.ssh/id_rsa"
```

Now that we have confirmed the presence of a key, we need to upload the public portion to AWS:

```
- name: upload the users public key
  ec2_key:
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}-{{ ansible_user_id }}"
    key_material: "{{ item }}"
    with_file: "~/.ssh/id_rsa.pub"
```

We now have everything in place to launch an EC2 instance; to do this, we will be using the `ec2_instance` module, which was introduced in Ansible 2.5:

```
- name: launch an instance
  ec2_instance:
    region: "{{ ec2_region }}"
    state: "present"
    instance_type: "{{ ec2_instance_type }}"
    image_id: "{{ our_ami_id }}"
    wait: yes
    key_name: "{{ environment_name }}-{{ ansible_user_id }}"
    security_groups: [ "{{ sg_ec2.group_id }}" ]
    network:
      assign_public_ip: true
    filters:
      instance-state-name: "running"
      "tag:Name": "{{ environment_name }}-tmp"
      "tag:environment": "{{ environment_name }}"
    vpc_subnet_id: "{{ subnet_ec2_ids[0] }}"
    tags:
      Name: "{{ environment_name }}-tmp"
      environment: "{{ environment_name }}"
```

With this, we are launching our EC2 instance into one of the EC2 subnets, attaching a public IP address and also our EC2 security group. The instance will be a t2.micro CentOS 7 instance called `wordpress-tmp`. We are assigning tags to it, and we are also using filters so that if anything happens during the playbook run and we need to rerun it, it will use our instance that is already running rather than launching another.

Once the instance has launched, we need to find out its information and add it to a host group called `ec2_instance`:

```
- name: gather facts on the instance we just launched using the AWS API
  ec2_instance_facts:
    region: "{{ ec2_region }}"
    filters:
      instance-state-name: "running"
      "tag:Name": "{{ environment_name }}-tmp"
      "tag:environment": "{{ environment_name }}"
  register: singleinstance

- name: add our temporary instance to a host group for use in the next step
  add_host:

    name: "{{ item.public_dns_name }}"
    ansible_ssh_host: "{{ item.public_dns_name }}"
    groups: "ec2_instance"
    with_items: "{{ singleinstance.instances }}
```

We need to wait for SSH to be accessible before moving on; here, we will use the `wait_for` module:

```
- name: wait until SSH is available before moving onto the next step
  wait_for:
    host: "{{ item.public_dns_name }}"
    port: 22
    delay: 2
    timeout: 320
    state: "started"
  with_items: "{{ singleinstance.instances }}
```

Finally, once SSH is available, we need to register the instance with our Elastic Load Balancer target group:

```
- name: add the instance to the target group
  elb_target_group:
    name: "{{ environment_name }}-target-group"
    region: "{{ ec2_region }}"
    protocol: "http"
    port: "80"
    vpc_id: "{{ vpc_info.vpc.id }}"
    state: "present"
    targets:
      - Id: "{{ item.instance_id }}"
        Port: "80"
```

```
|   modify_targets: "true"
|   with_items: "{{ singleinstance.instances }}"
```

This will leave us with a single instance called `wordpress-tmp`, which is accessible over SSH and active behind our Elastic Load Balancer in a host group named `ec2_instance`.

Existing deployment

If we already have instances running, the previous tasks are skipped and the single task in `roles/ec2/existing_deployment.yml` is run. This task simply takes one of the running hosts and adds it to the host group named `ec2_instance`:

```
- name: add one of our running instances to a host group for use in the next step
  add_host:
    name: "{{ groups['already_running'][0] }}"
    ansible_ssh_host: "{{ groups['already_running'][0] }}"
    groups: "ec2_instance"
```

This leaves us in the same position as we were in at the end of the new deployment tasks, with a host called `ec2_instance` with a single instance that is accessible over SSH.

Stack

The next role that we are going to create is the one that is only executed on the host—the `ec2_instance` group called `stack`. As with the previous roles, we can run the following command from within our `aws-wordpress` folder to create the files needed: **\$ ansible-galaxy init roles/stack**

This role is three roles in one. As with the EC2 role, we are building in logic to execute tasks based on the state of the instance our playbook finds when it first connects. Let's look at the contents of `roles/stack/tasks/main.yml` first.

The first task in there is executed on both new and existing deployments; it runs a `yum update`:

```
- name: update all of the installed packages
  yum:
    name: "*"
    state: "latest"
    update_cache: "yes"
```

Next, we need to know whether WordPress is installed:

```
- name: are the wordpress files already there?
  stat:
    path: "{{ wordpress_system.home }}/index.php"
  register: wp_installed
```

The next two tasks include two additional roles; one installs and configures the software stack and the other performs the initial WordPress installation, but only if no existing installation is found:

```
- name: if no wordpress installed install and configure the software stack
  include_role:
    name: "stack"
    tasks_from: "deploy.yml"
  when: wp_installed.stat.exists == False

- name: if no wordpress installed, install it !!!
  include_role:
    name: "stack"
    tasks_from: "wordpress.yml"
  when: wp_installed.stat.exists == False
```

These two roles are condensed versions of the roles we created when we

installed WordPress locally.

Default variables

Before we look at the roles, let's take a look at the code of `roles/stack/default/main.yml` as there are a few differences:

```
wp_cli:
  download: "https://raw.githubusercontent.com/wp-cli/builds/gh-pages/phar/wp-cli.phar"
  path: "/usr/local/bin/wp"

wordpress:
  domain: "http://{{ elb_results.load_balancers[0].dns_name }}/"
  title: "WordPress installed by Ansible on AWS"
  username: "ansible"
  password: "password"
  email: "test@example.com"

efs_mount_dir: "/efs"

wordpress_system:
  user: "wordpress"
  group: "php-fpm"
  comment: "wordpress system user"
  home: "{{ efs_mount_dir }}/wordpress"
  state: "present"

php:
  ip: "127.0.0.1"
  port: "9000"
  upstream: "php"
  ini:
    - { regexp: '^date.timezone =', replace: 'date.timezone = Europe/London' }
    - { regexp: '^expose_php = On', replace: 'expose_php = Off' }
    - { regexp: '^upload_max_filesize = 2M', replace: 'upload_max_filesize = 20M' }

selinux:
  http_permissive: true

repo_packages:
  - "epel-release"
  - "https://centos7.iuscommunity.org/ius-release.rpm"

nginx_repo:
  name: "nginx"
  description: "The mainline NGINX repo"
  baseurl: "http://nginx.org/packages/mainline/centos/7/$basearch/"
  gpgcheck: "no"
  enabled: "yes"

system_packages:
  - "MySQL-python"
  - "policycoreutils-python"
  - "nfs-utils"

stack_packages:
```

```

- "nginx"
- "mariadb"
- "php72u"
- "php72u-bcmath"
- "php72u-cli"
- "php72u-common"
- "php72u-dba"
- "php72u-fpm"
- "php72u-fpm-nginx"
- "php72u-gd"
- "php72u-intl"
- "php72u-json"
- "php72u-mbstring"
- "php72u-mysqlnd"
- "php72u-process"
- "php72u-snmp"
- "php72u-soap"
- "php72u-xml"
- "php72u-xmlrpc"

extra_packages:
- "vim-enhanced"
- "git"
- "unzip"

```

The main differences are:

- The `wordpress.domain` URL: This time, rather than hardcoding the domain, we have the Elastic Load Balancer URL, which we get from using the `elb_application_lb_facts` module. More on that later.
- The `efs_mount_dir` variable: This is a new variable, which we will use to define where in the instance we want our EFS share mounted.
- The `wordpress_system.home` option: This now uses `efs_mount_dir` so our WordPress installation can be shared across all instances.
- Lack of a MariaDB server: You will notice that references to installing and configuring a MariaDB server have been removed; as we have an RDS instance, we no longer need these.

We are using the `include_role` module to execute the tasks as a role to ensure that the variables are loaded correctly.

Deploy

The first additional role, called `roles/stack/tasks/deploy.yml`, does as you would expect and deploys the software stack and configuration.

It starts by mounting the EFS share; first, we need to gather some information about the EFS share using the `efs_facts` module:

```
- name: find some information on the elastic load balancer
  local_action:
    module: efs_facts
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}-efs"
  become: no
```

You may have already noticed that we are calling the `efs_facts` module differently; we are actually using the `local_action` module, which runs the `efs_facts` module on our Ansible controller rather than the EC2 instance. This is because we are not actually giving our EC2 instance access to the API, as we are not installing Boto or passing our access key and secret access key as a variable.

Using the `local_action` module allows us to flip back to our Ansible controller to gather information on our EFS and then apply the results on our EC2 instance; we will be using this module again later in the installation.

We are using `become: no` as part of this task; otherwise, it will try to execute using `sudo`. This is because we are telling all tasks to use `become: yes` with `become_method: sudo` in this part of the `site.yml` file, which we will update later in this chapter.

The next task mounts the EFS share and also adds it to the `fstab` file, which means that it will automatically mount when the instance we will be launching from the AMI we are creating first boots:

```
- name: ensure EFS volume is mounted.
  mount:
    name: "{{ efs_mount_dir }}"
    src: "{{ efs[0].file_system_id }}.efs.{{ ec2_region }}.amazonaws.com:/"
    fstype: nfs4
    opts: nfsvers=4.1
    state: mounted
```

`efs_mount_dir` is automatically created so we don't need to worry about creating it

beforehand. The next part of the role installs and configures the stack:

```
- name: install the repo packages
  yum:
    name: "{{ item }}"
    state: "installed"
  with_items: "{{ repo_packages }}"

- name: add the NGINX mainline repo
  yum_repository:
    name: "{{ nginx_repo.name }}"
    description: "{{ nginx_repo.description }}"
    baseurl: "{{ nginx_repo.baseurl }}"
    gpgcheck: "{{ nginx_repo.gpgcheck }}"
    enabled: "{{ nginx_repo.enabled }}"

- name: install the stack packages
  yum:
    name: "{{ item }}"
    state: "installed"
  with_items: "{{ system_packages + stack_packages + extra_packages }}"

- name: add the wordpress user
  user:
    name: "{{ wordpress_system.user }}"
    group: "{{ wordpress_system.group }}"
    comment: "{{ wordpress_system.comment }}"
    home: "{{ wordpress_system.home }}"
    state: "{{ wordpress_system.state }}"

- name: copy the nginx.conf to /etc/nginx/
  template:
    src: "nginx-nginx.conf.j2"
    dest: "/etc/nginx/nginx.conf"
    notify: "restart nginx"

- name: create the global directory in /etc/nginx/
  file:
    dest: "/etc/nginx/global/"
    state: "directory"
    mode: "0644"

- name: copy the restrictions.conf to /etc/nginx/global/
  copy:
    src: "nginx-global-restrictions.conf"
    dest: "/etc/nginx/global/restrictions.conf"
    notify: "restart nginx"

- name: copy the wordpress_shared.conf to /etc/nginx/global/
  template:
    src: "nginx-global-wordpress_shared.conf.j2"
    dest: "/etc/nginx/global/wordpress_shared.conf"
    notify: "restart nginx"

- name: copy the default.conf to /etc/nginx/conf.d/
  template:
    src: "nginx-confd-default.conf.j2"
    dest: "/etc/nginx/conf.d/default.conf"
    notify: "restart nginx"

- name: copy the www.conf to /etc/php-fpm.d/
  template:
    src: "php-fpmd-www.conf.j2"
```

```

    dest: "/etc/php-fpm.d/www.conf"
    notify: "restart php-fpm"

- name: configure php.ini
  lineinfile:
    dest: "/etc/php.ini"
    regexp: "{{ item.regexp }}"
    line: "{{ item.replace }}"
    backup: "yes"
    backrefs: "yes"
  with_items: "{{ php.ini }}"
  notify: "restart php-fpm"

- name: start php-fpm
  service:
    name: "php-fpm"
    state: "started"

- name: start nginx
  service:
    name: "nginx"
    state: "started"

- name: set the selinux allowing httpd_t to be permissive is required
  selinux_permissive:
    name: httpd_t
    permissive: true
  when: selinux.http_permissive == true

```

For this to work, you will need to copy the files from `files`, `handlers`, and `templates` from the `stack-config` role of the LEMP playbook we created in [Chapter 5, Deploying WordPress](#).

WordPress

As you may have already guessed, this role, which can be found in the `roles/stack/tasks/wordpress.yml` file alongside `roles/stack/tasks/main.yml` and `roles/stack/tasks/deploy.yml`, installs and configures WordPress.

Before we progress with the tasks, we need to find out information about our RDS instance:

```
- name: find some information on the rds instance
  local_action:
    module: rds
    region: "{{ ec2_region }}"
    command: facts
    instance_name: "{{ environment_name }}-rds"
  become: no
  register: rds_results
```

This is so that we can use the tasks when defining the database connection; likewise, we also need to find out about the Elastic Load Balancer:

```
- name: find some information on the elastic load balancer
  local_action:
    module: elb_application_lb_facts
    region: "{{ ec2_region }}"
    names: "{{ environment_name }}-elb"
  become: no
  register: elb_results
```

The remaining tasks do the following:

1. Install WP-CLI.
2. Download WordPress.
3. Set the correct permissions on the WordPress folder.
4. Configure WordPress to connect to our RDS using the endpoint we found when gathering facts; we are reusing the password file we generated.
5. Install WordPress using the Elastic Load Balancer URL and details from the default variables:

```
- name: download wp-cli
  get_url:
    url: "{{ wp_cli.download }}"
    dest: "{{ wp_cli.path }}"
- name: update permissions of wp-cli to allow anyone to execute it
```

```

file:
  path: "{{ wp_cli.path }}"
  mode: "0755"

- name: are the wordpress files already there?
  stat:
    path: "{{ wordpress_system.home }}/index.php"
  register: wp_installed

- name: download wordpresss
  shell: "{{ wp_cli.path }} core download"
  args:
    chdir: "{{ wordpress_system.home }}"
  become_user: "{{ wordpress_system.user }}"
  become: true
  when: wp_installed.stat.exists == False

- name: set the correct permissions on the homedir
  file:
    path: "{{ wordpress_system.home }}"
    mode: "0775"
  when: wp_installed.stat.exists == False

- name: is wordpress already configured?
  stat:
    path: "{{ wordpress_system.home }}/wp-config.php"
  register: wp_configured

- name: configure wordpress
  shell: "{{ wp_cli.path }} core config --dbhost={{ rds_results.instance.endpoint }} --dbname={{ environment_name }} --dbuser={{ environment_name }} --dbpass={{ lookup('password', 'group_vars/rds_passwordfile chars=ascii_letters,digits length=30') }}"
  args:
    chdir: "{{ wordpress_system.home }}"
  become_user: "{{ wordpress_system.user }}"
  become: true
  when: wp_configured.stat.exists == False

- name: do we need to install wordpress?
  shell: "{{ wp_cli.path }} core is-installed"
  args:
    chdir: "{{ wordpress_system.home }}"
  become_user: "{{ wordpress_system.user }}"
  become: true
  ignore_errors: yes
  register: wp_installed

- name: install wordpress if needed
  shell: "{{ wp_cli.path }} core install --url='{{ wordpress.domain }}' --title='{{ wordpress.title }}' --admin_user={{ wordpress.username }} --admin_password={{ wordpress.password }} --admin_email={{ wordpress.email }}"
  args:
    chdir: "{{ wordpress_system.home }}"
  become_user: "{{ wordpress_system.user }}"
  become: true
  when: wp_installed.rc == 1

```

To keep things simple, we are not managing the theme or plugins using Ansible.

This is where we stop running tasks on the instance we discovered/launched in

the previous role; it is now time for us to switch back to our Ansible controller and make an AMI using our instance.

AMI

This role does not need to make any choices, it simply takes our host from the `ec2_instances` group and creates an image of it. To start, let's create the role:

```
| $ ansible-galaxy init roles/ami
```

The role is made up of three tasks, one of which is a pause. First of all, in `roles/ami/tasks/main.yml`, we need to find out some information about the instance. We are using the `ec2_instance_facts` module:

```
- name: find out some facts about the instance we have been using
  ec2_instance_facts:
    region: "{{ ec2_region }}"
    filters:
      dns-name: "{{ groups['ec2_instance'][0] }}"
  register: "our_instance"
```

Now we know about the instance, we can create the AMI:

```
- name: create the AMI
  ec2_ami:
    region: "{{ ec2_region }}"
    instance_id: "{{ our_instance.instances.0.instance_id }}"
    wait: "yes"
    name: "{{ environment_name }}-{{ ansible_date_time.date }}-{{ ansible_date_time.hour }}{{ ansible_date_time.minute }}"
    tags:
      Name: "{{ environment_name }}-{{ ansible_date_time.date }}-{{ ansible_date_time.hour }}{{ ansible_date_time.minute }}"
      Environment: "{{ environment_name }}"
      Date: "{{ ansible_date_time.date }} {{ ansible_date_time.time }}"
```

As you can see, we are using the `instance_id` we discovered when running the `ec2_instance_facts` module; we are also using the `ansible_date_time` variable, which was defined when the `gather_facts` module was called to give our AMI a unique name.

As already mentioned, the final task is a pause:

```
- name: wait for 2 minutes before continuing
  pause:
    minutes: 2
```

This is required as it can take a short while for our newly created AMI to fully

register and be available in the AWS API.

Autoscaling

The final role in our playbook creates a launch configuration and then creates/updates an autoscaling group to finally launch our instances. It then does a tiny bit of housekeeping. To create the role, run:

```
| $ ansible-galaxy init roles/autoscaling
```

First of all, there are a few default variables we need to set in `roles/autoscaling/default/main.yml`; these details show how many instances we want running at any one time, and also how many instances we replace at a time when doing a deployment of a new AMI:

```
min_size: 2
max_size: 9
desired_capacity: 3
replace_size: 2
health_check_type: ELB
assign_public_ip: yes
ec2_instance_type: "t2.micro"
```

What those variables are saying is that we want three instances running at all times, so if there are two, then launch more and never launch more than nine at any one time. When deploying a new image, replace instances two at a time.

We are also defining the health check, where, using the Elastic Load Balancer check, we are telling the instances to launch using a public IP address, meaning that we can access them over SSH, and finally, we are defining the type of instance to use.

The first task we need to define in `roles/autoscaling/tasks/main.yml` needs to find the right AMI to use:

```
- name: search through all of our AMIs
  ec2_ami_facts:
    region: "{{ ec2_region }}"
    filters:
      name: "{{ environment_name }}-*"
  register: amiFind
```

Again, we need to know the details of the last AMI we built:

```
| - name: find the last one we built
```

```
| set_fact:  
|   amiSortFilter: "{{ amiFind.images | sort(attribute='creation_date') | last }}"
```

Finally, we need to get the AMI ID and also the AMI name; we will be using this to name the launch configuration:

```
| - name: grab AMI ID and name of the most recent result  
|   set_fact:  
|     our_ami_id: "{{ amiSortFilter.image_id }}"  
|     our_ami_name: "{{ amiSortFilter.name }}"
```

Next up, we have the task, which uses the previous information to create the launch configuration:

```
| - name: create the launch configuration  
|   ec2_lc:  
|     region: "{{ ec2_region }}"  
|     name: "{{ our_ami_name }}"  
|     state: present  
|     image_id: "{{ our_ami_id }}"  
|     security_groups: [ "{{ sg_ec2.group_id }}" ]  
|     assign_public_ip: "{{ assign_public_ip }}"  
|     instance_type: "{{ ec2_instance_type }}"  
|     volumes:  
|       - device_name: /dev/xvda  
|         volume_size: 10  
|         volume_type: gp2  
|         delete_on_termination: true
```

Once we have the launch configuration created, we can create/update the autoscaling group to reference it. Before we do, we need to find out the **Amazon Resource Name (ARN)** of the target group:

```
| - name: find out the target group ARN  
|   elb_target_group_facts:  
|     region: "{{ ec2_region }}"  
|     names:  
|       - "{{ environment_name }}-target-group"  
|   register: elb_target_group
```

Once we have that information, we can move on to the next task:

```
| - name: create / update the auto-scaling group using the launch configuration we just  
| created  
|   ec2_asg:  
|     region: "{{ ec2_region }}"  
|     name: "{{ environment_name }}-asg"  
|     target_group_arns: [ "{{ elb_target_group.target_groups[0].target_group_arn }}" ]  
|     launch_config_name: "{{ our_ami_name }}"  
|     min_size: "{{ min_size }}"  
|     max_size: "{{ max_size }}"  
|     desired_capacity: "{{ desired_capacity }}"  
|     health_check_period: 300  
|     health_check_type: "{{ health_check_type }}"  
|     replace_all_instances: yes
```

```
replace_batch_size: "{{ replace_size }}"
vpc_zone_identifier: "{{ subnet_ec2_ids }}"
wait_for_instances: "yes"
wait_timeout: "900"
tags:
  - Name: "{{ environment_name }}-ec2"
  - environment: "{{ environment_name }}"
```

An autoscaling group ensures that we have our desired number of EC2 instances running at all times. If there are none running, it launches them and registers them with the target group for the Elastic Load Balancer.

If there are instances already running and we have updated the launch configuration, then it will do a rolling deploy of our new configuration, making sure that we never have any downtime as new instances are launched and registered before old ones are removed.

The final task removes any `tmp` instances we may have running:

```
- name: remove any tmp instances which are running
  ec2_instance:
    region: "{{ ec2_region }}"
    state: absent
  filters:
    instance-state-name: "running"
    "tag:environment": "{{ environment_name }}"
    "tag:Name": "{{ environment_name }}-tmp"
```

This should leave us with our desired state running and nothing more.

Running the playbook

The first thing we need to do is update our `production` inventory file; this should look like the following:

```
# Register all of the host groups we will be creating in the playbooks
[ec2_instance]
[already_running]

# Put all the groups into into a single group so we can easily apply one config to it
# for overriding things like the ssh user and key location
[aws:children]
ec2_instance
already_running

# Finally, configure some bits to allow us access to the instances before we deploy our
# credentials using Ansible
[aws:vars]
ansible_ssh_user=centos
ansible_ssh_private_key_file=~/ssh/id_rsa
host_key_checking=False
```

As you can see, we are defining the host groups and also configuring Ansible to use the `centos` user, which is the default for the original AMI we are using.

The `site.yml` file needs to be updated:

```
---
- name: Create, launch and configure our basic AWS environment
  hosts: localhost
  connection: local
  gather_facts: True

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/vpc
    - roles/subnets
    - roles/gateway
    - roles/securitygroups
    - roles/elb
    - roles/rds
    - roles/efs
    - roles/ec2

- name: Configure / update the EC2 instance
  hosts: ec2_instance
  become: yes
  become_method: sudo
  gather_facts: True
```

```

vars_files:
  - group_vars/common.yml

roles:
  - roles/stack

- name: Create, launch and configure our AMI
hosts: localhost
connection: local
gather_facts: True

vars_files:
  - group_vars/common.yml

roles:
  - roles/ami
  - roles/autoscaling

```

As you can see, we now have three sections; the first section prepares the environment, as we have already seen—there is also the addition of the `ec2` role. This section is all executed on the Ansible controller.

In the next section, we move over to running the roles against the host in the `ec2_instance` group; as already mentioned, we are using `become: yes` and `become_method: sudo` on this host because the user we are connecting with, `centos`, does not have the correct privileges we need to install our software stack. This is why we need to disable `become` when using the `local_action` module. The third section takes us back to our Ansible controller, where we use the AWS API to create our AMI and launch it.

Don't forget to set your access key and secret access key environment variables:

```

$ export AWS_ACCESS_KEY=AKIAI5KECPOTNTTVM3EDA
$ export AWS_SECRET_KEY=Y4B7FFiSw10Am3VIFc071gnc/TAtK5+RpxzIGTr

```

 Before we run the playbook you need to make sure that you are subscribed to the CentOS 7 Amazon Machine Image in the AWS Marketplace, to do this go to the following link and hit the subscribe button, if you are not subscribed to the AMI you will receive an error when you run the playbook instructing you that you do not have access to the image: <https://aws.amazon.com/marketplace/pp/B0007WM7QW>.

We are going to be timing our playbook to run again, so, to execute the playbook, use the following commands:

```
| $ time ansible-playbook -i production site.yml
```

As we have already seen the output of half of the playbook running, I am going to skip the output of the `vpc`, `subnets`, `gateway`, `securitygroups`, `elb`, `rds`, and `efs` roles, meaning that we will start with the `ec2` one:

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit localhost does not match 'all'

PLAY [Create, launch and configure our basic AWS environment]
*****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [roles/ec2 : gather facts on any already running instances]
*****
ok: [localhost]

TASK [roles/ec2 : add any already running instances to a group]
*****
TASK [roles/ec2 : set the number of already running instances as a fact]
*****
ok: [localhost]

TASK [roles/ec2 : run the tasks for a new deployment]
*****
included: /Users/russ/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter10/aws-wordpress/roles/ec2/tasks/new_deployment.yml for localhost

TASK [roles/ec2 : search for all of the AMIs in the defined region which match our selection] ***
ok: [localhost]

TASK [roles/ec2 : filter the list of AMIs to find the latest one with an EBS backed volume] ****
ok: [localhost]

TASK [roles/ec2 : finally grab AMI ID of the most recent result which matches our base image which is backed by an EBS volume]
*****
ok: [localhost]

TASK [roles/ec2 : check the user russ has a key, if not create one]
*****
ok: [localhost]

TASK [roles/ec2 : upload the users public key]
*****
ok: [localhost] => (item=ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQDmuoFR01i/Yf3HATl9c3sufJvghTFgYzK/Zt29JiTqWlSQhmXhNNTh6iIE
russ@mckendrick.io)

TASK [roles/ec2 : launch an instance]
*****
changed: [localhost]

TASK [roles/ec2 : gather facts on the instance we just launched using the AWS API]
*****
ok: [localhost]

TASK [roles/ec2 : add our temporary instance to a host group for use in the next step]
*****
changed: [localhost] =>

TASK [roles/ec2 : wait until SSH is available before moving onto the next step]
*****
```

```
ok: [localhost] =>

TASK [roles/ec2 : add the instance to the target group]
*****
changed: [localhost] =>

TASK [roles/ec2 : run the tasks for an existing deployment]
*****
skipping: [localhost]

PLAY [Configure / update the EC2 instance]
*****

TASK [Gathering Facts]
*****
ok: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [roles/stack : update all of the installed packages]
*****
ok: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [roles/stack : are the wordpress files already there?]
*****
ok: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [roles/stack : if no wordpress installed install and configure the software stack]
*****
ok: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : find some information on the elastic load balancer]
*****
ok: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com -> localhost]

TASK [stack : ensure EFS volume is mounted.]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : install the repo packages]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com] => (item=[u'epel-release', u'https://centos7.iuscommunity.org/ius-release.rpm'])

TASK [stack : add the NGINX mainline repo]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : install the stack packages]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com] => (item=[u'MySQL-python', u'policycoreutils-python', u'nfs-utils', u'nginx', u'mariadb', u'php72u', u'php72u-bcmath', u'php72u-cli', u'php72u-common', u'php72u-dba', u'php72u-fpm', u'php72u-fpm-nginx', u'php72u-gd', u'php72u-intl', u'php72u-json', u'php72u-mbstring', u'php72u-mysqlnd', u'php72u-process', u'php72u-snmp', u'php72u-soap', u'php72u-xml', u'php72u-xmlrpc', u'vim-enhanced', u'git', u'unzip'])

TASK [stack : add the wordpress user]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : copy the nginx.conf to /etc/nginx/]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : create the global directory in /etc/nginx/]
*****
```

```
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : copy the restrictions.conf to /etc/nginx/global/]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : copy the wordpress_shared.conf to /etc/nginx/global/]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : copy the default.conf to /etc/nginx/conf.d/]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : copy the www.conf to /etc/php-fpm.d/]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : configure php.ini]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com] => (item={u'regexp': u'^date.timezone =', u'replace': u'date.timezone = Europe/London'})
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com] => (item={u'regexp': u'^expose_php = On', u'replace': u'expose_php = Off'})
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com] => (item={u'regexp': u'^upload_max_filesize = 2M', u'replace': u'upload_max_filesize = 20M'})

TASK [stack : start php-fpm]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : start nginx]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : set the selinux allowing httpd_t to be permissive is required]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [roles/stack : if no wordpress installed, install it !!!]
*****

TASK [stack : download wp-cli]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : update permissions of wp-cli to allow anyone to execute it]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : find some information on the rds instance]
*****
ok: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com -> localhost]

TASK [stack : find some information on the elastic load balancer]
*****
ok: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com -> localhost]

TASK [stack : are the wordpress files already there?]
*****
ok: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : download wordpresss]
*****
```

```
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : set the correct permissions on the homedir]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : is wordpress already configured?]
*****
ok: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : configure wordpress]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

TASK [stack : do we need to install wordpress?]
*****
fatal: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]: FAILED! => {"changed": true, "cmd": "/usr/local/bin/wp core is-installed", "delta": "0:00:01.547784", "end": "2018-05-06 14:19:01.301168", "msg": "non-zero return code", "rc": 1, "start": "2018-05-06 14:18:59.753384", "stderr": "", "stderr_lines": [], "stdout": "", "stdout_lines": []} ...ignoring

TASK [stack : install wordpress if needed]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

RUNNING HANDLER [roles/stack : restart nginx]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

RUNNING HANDLER [roles/stack : restart php-fpm]
*****
changed: [ec2-34-244-58-38.eu-west-1.compute.amazonaws.com]

PLAY [Create, launch and configure our AMI]
*****

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

TASK [roles/ami : find out some facts about the instance we have been using]
*****
ok: [localhost]

TASK [roles/ami : create the AMI]
*****
changed: [localhost]

TASK [roles/ami : wait for 2 minutes before continuing]
*****
Pausing for 120 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [localhost]

TASK [roles/autoscaling : search through all of our AMIs]
*****
ok: [localhost]

TASK [roles/autoscaling : find the last one we built]
*****
ok: [localhost]

TASK [roles/autoscaling : grab AMI ID and name of the most recent result]
```

```
*****
ok: [localhost]

TASK [roles/autoscaling : create the launch configuration]
*****
changed: [localhost]

TASK [roles/autoscaling : find out the target group ARN]
*****
ok: [localhost]

TASK [roles/autoscaling : create / update the auto-scaling group using the launch
configuration we just created]
*****
changed: [localhost]

TASK [roles/autoscaling : remove any tmp instances]
*****
changed: [localhost]

PLAY RECAP
*****
ec2-34-244-58-38.eu-west-1.compute.amazonaws.com : ok=32 changed=24 unreachable=0
failed=0
localhost : ok=47 changed=21 unreachable=0 failed=0
```

The playbook ran for me in the following time:

```
| real 31m34.752s
| user 2m4.008s
| sys 0m39.274s
```

So, from a single command and in 32 minutes, we have a highly available vanilla WordPress installation. If you find out the public URL of your Elastic Load Balancer from the AWS console, you should be able to see your site:



Checking the EC2 instances in the AWS console, we can see that there are three instances, all called `wordpress-ec2`, running and the `wordpress-tmp` instance has been terminated:

The screenshot shows the AWS CloudWatch Metrics interface. A metric named "AWS Lambda" is selected. The graph displays the metric value over time, with data points for each hour. The Y-axis is labeled "Value" and ranges from 0 to 100. The X-axis shows dates from June 2018 to July 2018. The metric value fluctuates between 0 and 100, with a notable peak around July 1st.

Now, let's see what happens when we run the playbook again. We should not only see it execute more quickly, but it should skip a few roles:

```
| $ time ansible-playbook -i production site.yml
```

Again, I have truncated the output:

```
WARNING]: provided hosts list is empty, only localhost is available. Note that the
implicit localhost does not match 'all'

PLAY [Create, launch and configure our basic AWS environment]
*****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [roles/ec2 : gather facts on any already running instances]
*****
ok: [localhost]

TASK [roles/ec2 : add any already running instances to a group]
*****
changed: [localhost] =>

TASK [roles/ec2 : set the number of already running instances as a fact]
*****
ok: [localhost]

TASK [roles/ec2 : run the tasks for a new deployment]
*****
skipping: [localhost]

TASK [roles/ec2 : run the tasks for an existing deployment]
*****
included: /Users/russ/Documents/Code/learn-ansible-fundamentals-of-ansible-
2x/chapter10/aws-wordpress/roles/ec2/tasks/existing_deployment.yml for localhost

TASK [roles/ec2 : add one of our running instances to a host group for use in the next
step] ****
changed: [localhost]

PLAY [Configure / update the EC2 instance]
*****
TASK [Gathering Facts]
*****
ok: [ec2-52-211-180-156.eu-west-1.compute.amazonaws.com]

TASK [roles/stack : update all of the installed packages]
*****
changed: [ec2-52-211-180-156.eu-west-1.compute.amazonaws.com]

TASK [roles/stack : are the wordpress files already there?]
*****
ok: [ec2-52-211-180-156.eu-west-1.compute.amazonaws.com]

TASK [roles/stack : if no wordpress installed install and configure the software stack]
*****
skipping: [ec2-52-211-180-156.eu-west-1.compute.amazonaws.com]

TASK [roles/stack : if no wordpress installed, install it !!!]
*****
skipping: [ec2-52-211-180-156.eu-west-1.compute.amazonaws.com]

PLAY [Create, launch and configure our AMI]
*****
```

```

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

TASK [roles/ami : find out some facts about the instance we have been using]
*****
ok: [localhost]

TASK [roles/ami : create the AMI]
*****
changed: [localhost]

TASK [roles/ami : wait for 2 minutes before continuing]
*****
Pausing for 120 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [localhost]

TASK [roles/autoscaling : search through all of our AMIs]
*****
ok: [localhost]

TASK [roles/autoscaling : find the last one we built]
*****
ok: [localhost]

TASK [roles/autoscaling : grab AMI ID and name of the most recent result]
*****
ok: [localhost]

TASK [roles/autoscaling : create the launch configuration]
*****
changed: [localhost]

TASK [roles/autoscaling : find out the target group ARN]
*****
ok: [localhost]

TASK [roles/autoscaling : create / update the auto-scaling group using the launch
configuration we just created]
*****
changed: [localhost]

TASK [roles/autoscaling : remove any tmp instances]
*****
ok: [localhost]

PLAY RECAP
*****
ec2-52-211-180-156.eu-west-1.compute.amazonaws.com : ok=3 changed=1 unreachable=0
failed=0
localhost : ok=39 changed=5 unreachable=0 failed=0

```

This time, I got the following timings returned:

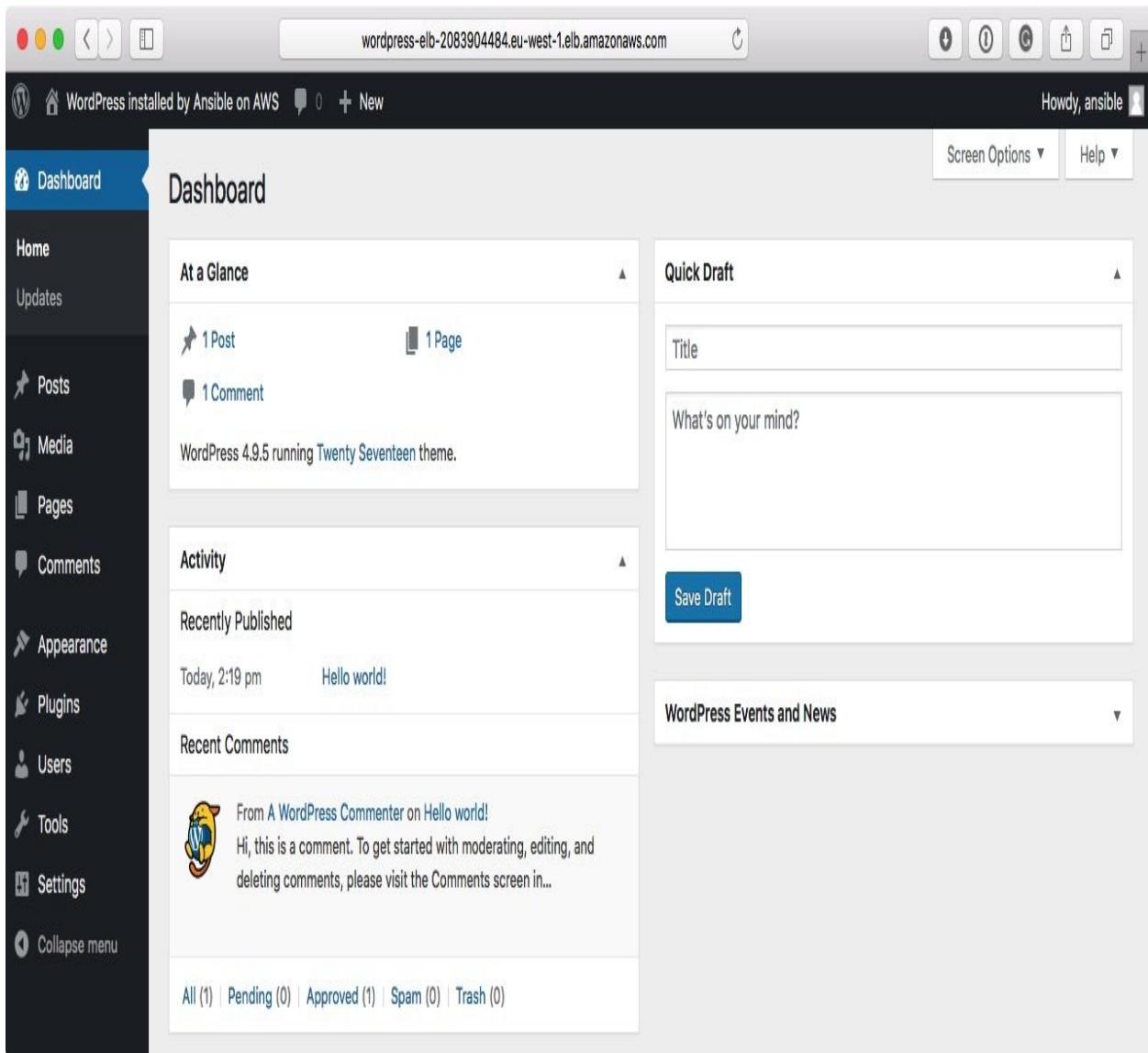
```

real 9m18.502s
user 0m48.718s
sys 0m14.115s

```

Once complete, I checked that I could still log in to WordPress using the

username (`ansible`) and password (`password`) we set in the playbook by going to my Elastic Load Balancer URL and adding `/wp-admin` to the end:



You can see what happened in the autoscaling activity logs in the AWS console:

The screenshot shows the AWS Auto Scaling Groups page. At the top, there's a navigation bar with the AWS logo, 'Services' dropdown, 'Resource Groups' dropdown, and user information ('Russell Mckendrick', 'Ireland', 'Support'). Below the navigation is a toolbar with 'Create Auto Scaling group' (highlighted in blue), 'Actions' dropdown, and three icons. The main area has a 'Filter' input field and a breadcrumb path '1 to 1 of 1 Auto Scaling Groups'. A table lists one Auto Scaling Group: 'wordpress-asg' with details: Launch Configuration: 'wordpress-2018-05-06...', Instances: 3, Desired: 3, Min: 2, Max: 9, Availability Zones: 'eu-west-1b, eu-west-1c, eu...', Default Cooldown: 300, and Health Check Grace: 300. Below this, under 'Auto Scaling Group: wordpress-asg', is a tab navigation bar with 'Details' (disabled), 'Activity History' (selected), 'Scaling Policies', 'Instances', 'Monitoring', 'Notifications', 'Tags', 'Scheduled Actions', and 'Lifecycle Hooks'. The 'Activity History' section shows a table of scaling events from May 6, 2018, with 9 items. The columns are 'Status' (Successful), 'Description', 'Start Time', and 'End Time'. The events listed are:

Status	Description	Start Time	End Time
Successful	Terminating EC2 instance: i-0fd311f61d1d789e	2018 May 6 15:55:54 UTC+1	2018 May 6 15:57:15 UTC+1
Successful	Terminating EC2 instance: i-053e302b852908e5d	2018 May 6 15:55:54 UTC+1	2018 May 6 15:57:18 UTC+1
Successful	Launching a new EC2 instance: i-0a44454ceab42f6f1	2018 May 6 15:54:43 UTC+1	2018 May 6 15:55:16 UTC+1
Successful	Terminating EC2 instance: i-0103e58a7ca44e0f1	2018 May 6 15:54:19 UTC+1	2018 May 6 15:55:37 UTC+1
Successful	Launching a new EC2 instance: i-062139a24092fbea4	2018 May 6 15:53:14 UTC+1	2018 May 6 15:53:47 UTC+1
Successful	Launching a new EC2 instance: i-070de215d571a889a	2018 May 6 15:53:14 UTC+1	2018 May 6 15:53:47 UTC+1

At the bottom, there are links for 'Feedback', 'English (US)', and legal notices: '© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.' and 'Privacy Policy Terms of Use'.

As you can see, three new instances were launched and three terminated.

Terminating all the resources

Before we complete this chapter, we need to look at terminating the resources; the only additions we need to make are to remove the autoscaling group and AMIs. To do this, we are going to add four tasks to `roles/remove/tasks/main.yml`; starting at the top of the file, add the following two tasks:

```
- name: remove the auto-scaling group
  ec2_asg:
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}-asg"
    state: absent
    wait_for_instances: "yes"
    wait_timeout: "900"

- name: wait for 2 minutes before continuing
  pause:
    minutes: 2
```

The first task removes the autoscaling group. This, in turn, will terminate any instances that have been launched by it. We have also built in a pause to ensure that everything has been removed properly from the AWS API.

At the end of the role, add the following two tasks to remove all of the AMIs:

```
- name: search through all of our AMIs
  ec2_ami_facts:
    region: "{{ ec2_region }}"
    filters:
      name: "{{ environment_name }}-*"
    register: amiFind

- name: unregister all of our AMIs
  ec2_ami:
    image_id: "{{ item.image_id }}"
    delete_snapshot: True
    state: absent
    with_items: "{{ amiFind.images }}"
```

You can then run the playbook with the following command:

```
| $ ansible-playbook -i production remove.yml
```

As before, don't forget to check that the Elastic Load Balancer has been removed before progressing. Once the playbook has run, I would recommend you log in to the AWS console and double-check that everything has been correctly

removed. The playbook does not remove the launch configurations, which should not be a problem as there are no costs associated with them. However, I would recommend checking on unattached EBS volumes and snapshots as these will incur costs.

Summary

In this chapter, we have taken our AWS to the next level by creating and launching a highly available WordPress installation. By leveraging the various services offered by AWS, we engineered out any single points of failure with regards to the availability of instances and availability zones.

We also built logic into our playbook so that we can use the same command to launch a new deployment or update the operating system on an existing one with a rolling deploy of new instance AMIs that contain our updated packages—allowing for zero downtime during deployment.

While the WordPress deployment is probably as simple as we could make it, the process of deploying the production-ready images would remain similar when using a more complicated application.

In our next chapter, we are going to look at moving from public cloud to private cloud, and how Ansible interacts with VMware.

Questions

1. What is the name of the variable that is registered using the `gather_facts` option, which contains the date and time our playbook was executed?
2. True or false: Ansible automatically figures out which task it needs to execute, meaning we don't have to define any logic ourselves.
3. Explain why we have to use the `local_action` module.
4. Which command do we prepend to our `ansible-playbook` command to record how long our command took to execute?
5. True or false: When using autoscaling, you have to manually launch EC2 instances.
6. Update the playbook so that it gives you the public URL of the Elastic Load Balancer at the end of the playbook run.

Further reading

You can find more details about CentOS AMIs on the AWS Marketplace at <https://aws.amazon.com/mp/centos/>.

Building Out a VMware Deployment

Now that we know how to launch networking and services in AWS, we will now discuss deploying a similar setup in a VMware environment and also talk through the core VMware modules.

In this chapter, we will:

- Quickly introduce VMware
- Review the Ansible VMware modules
- Work through an example playbook that launches several virtual machines

Technical requirements

In this chapter, we are going to be discussing various components of the VMware family of products and how you can interact with them using Ansible. While there is an example playbook in this chapter, it may not be easily transferable to your installation. Because of this, it is not recommended that you use any of the examples in this chapter without first updating them.

An introduction to VMware

VMware has a nearly 20-year history, from a stealth start-up to being owned by Dell and acquired by EMC, with a revenue of \$7.92 billion. There are around 30 products currently available in the VMware product portfolio; the most commonly known ones are its hypervisors, of which there are two different types.

The first hypervisor, VMware ESXi, is a type 1 that runs directly on hardware using the instruction sets found in most modern 64-bit Intel and AMD CPUs. Its original type 2 hypervisor does not require virtualization instructions to be present within the CPU like they need to be in type 1. It was formerly known as GSX; this hypervisor pre-dates the type 1 hypervisor, meaning that it can support much older CPUs.

VMware is extremely commonplace in most enterprises; it allows administrators to quickly deploy virtual machines across numerous standard x86-based hardware configurations and types.

The VMWare modules

As already mentioned, there are around 30 products in the VMWare range; these cover everything from hypervisors to virtual switches, virtual storage, and several interfaces for interacting with your VMWare-based hosts and virtual machines. In this section, we will cover the core modules that ship with Ansible to manage all aspects of your VMWare estate.

I have tried to split them into logical groups and, for each of the groups, will give a brief explanation of the product the modules are targeting.

Requirements

The modules all have one thing in common: they all require a Python module called `PyVmomi` to be installed. To install this, run the following `pip` command:

```
$ sudo pip install PyVmomi
```

This module contains the VMware vSphere API Python bindings, and without it, the modules we are going to cover in this chapter will not be able to interact with your VMware installation.

While the modules in this chapter have been tested with vSphere 5.5 to 6.5, you may find that some of the older modules have some problems with newer versions of vSphere.

vCloud Air

vCloud Air was VMware's **Infrastructure as a Service (IaaS)** offering, I say *was* because the vCloud Air business unit and the team responsible for the service was acquired by French hosting and cloud company OVH from VMware in mid-2017. There are three Ansible modules which offer direct support for vCloud Air, as well as **VMware vCloud Hybrid Service (vCHS)** and **VMware vCloud Director (vCD)**.

The vca_fw module

This module enables you to add and remove firewall rules from a vCloud Air gateway. The following example shows you how to add a rule allowing SSH traffic:

```
- name: example fireware rule
```

```
vca_fw:  
  instance_id: "abcdef123456-1234-abcd-1234-abcdef123456"  
  vdc_name: "my_vcd"  
  service_type: "vca"  
  state: "present"  
  fw_rules:  
    - description: "Allow SSH"  
      source_ip: "10.20.30.40"  
      source_port: "Any"  
      dest_port: "22"  
      dest_ip: "192.0.10.20"  
      is_enable: "true"  
      enable_logging: "false"  
      protocol: "Tcp"  
      policy: "allow"
```

Notice how we are passing a `service_type`; this could be `vca`, `vcd`, OR `vchs`.

The vca_nat module

This module allows you to manage the **network address translation (NAT)** rules. In the following example, we are asking all traffic that hits port 2222 on the public IP address of 123.123.123.123 to be forwarded to port 22 on the virtual machine with an IP address of 192.0.10.20:

```
- name: example nat rule
vca_nat:
  instance_id: "abcdef123456-1234-abcd-1234-abcdef123456"
  vdc_name: "my_vcd"
  service_type: "vca"
  state: "present"
  nat_rules:
    - rule_type: "DNAT"
      original_ip: "123.123.123.123"
      original_port: "2222"
      translated_ip: "192.0.10.20"
      translated_port: "22"
```

This means that to access SSH on the virtual machine 192.0.10.20 from our external network, we would need to run something like the following command:

```
| $ ssh username@123.123.123.123 -p2222
```

Assuming we had the correct firewall rules in place, we should be routed through the 192.0.10.20 virtual machine.

The vca_vapp module

This module is used to both create and manage vApps. A vApp is one or more virtual machines which is combined to offer an application:

```
- name: example
```

```
vApp
```

```
vca_vapp:
```

```
vapp_name: "Example"
```

```
vdc_name: "my_vcd"
```

```
state: "present"
```

```
operation: "poweron"
```

```
template_name: "CentOS7 x86_64 1804"
```

The previous example is a really basic example of how you would use the `vca_vapp` module to ensure that there is a vApp called `example` present and powered on.

VMware vSphere

VMware vSphere is a suite of software from VMware which is made up of several VMware components. This is where VMware can get a little confusing as VMware vSphere is made up of VMware vCentre and VMware ESXi, each of which has their own Ansible modules as well, and on the face of it, they appear to accomplish similar tasks.

The vmware_cluster module

This module allows you to manage your VMware vSphere cluster. A VMware vSphere cluster is a collection of hosts, which when clustered together share resources, allowing you to add **high availability (HA)** and also launch a **Distributed Resource Scheduler (DRS)** which manages the placement of workloads within your cluster:

```
- name: Create a cluster
  vmware_cluster:
    hostname: "{{ item.ip }}"
    datacenter_name: "my_datacenter"
    cluster_name: "cluster"
    enable_ha: "yes"
    enable_drs: "yes"
    enable_vsan: "yes"
    username: "{{ item.username }}"
    password: "{{ item.password }}"
    with_items: "{{ vsphere_hosts }}"
```

The preceding code would loop through a list of hosts, usernames, and passwords to create a cluster.

The vmware_datacenter module

A VMware vSphere data center is the name given to the collection of physical resources, hosts, storage, and networking which power your cluster:

- name:
Create a datacenter

vmware_datacenter:

hostname: "{{ item.ip }}"

username: "{{ item.username }}"

password: "{{ item.password }}"

datacenter_name: "my_datacenter"

state: present

with_items: "{{ vsphere_hosts }}"

The previous example adds the hosts listed in `vsphere_hosts` to the `my_datacenter` VMware vSphere data center.

The vmware_vm_facts module

This module can be used to gather facts on either virtual machines or templates which are running in your VMware vSphere cluster:

- name: Gather facts on all VMs in the cluster

```
vmware_vm_facts:  
hostname: "{{ vsphere_host }}"  
username: "{{ vsphere_username }}"  
password: "{{ vsphere_password }}"  
vm_type: "vm"  
delegate_to: "localhost"  
register: vm_facts
```

The previous example gathers information on just the virtual machines that have been created with our cluster and registers the results as the `vm_facts` variable. If we wanted to find information on just the templates, we could update `vm_type` to `template`, or we could list all virtual machines and templates by updating `vm_type` to `all`.

The vmware_vm_shell module

This module can be used to connect to a virtual machine using VMware and to run a shell command. At no point does Ansible need to connect to the virtual machine using a network-based service such as SSH, making it useful for configuring VMs before they are on the network:

```
- name: Shell example
  vmware_vm_shell:
```

```
  hostname: "{{ vsphere_host }}"
  username: "{{ vsphere_username }}"
  password: "{{ vsphere_password }}"
  datacenter: "my_datacenter"
  folder: "/my_vms"
  vm_id: "example_vm"
  vm_username: "root"
  vm_password: "supersecretpassword"
  vm_shell: "/bin/cat"
  vm_shell_args: " results_file "
  vm_shell_env:
    - "PATH=/bin"
    - "VAR=test"
  vm_shell_cwd: "/tmp"
  delegate_to: "localhost"
  register: shell_results
```

The previous example connects to a VM called `example_vm`, which is stored in the `my_vms` folder at the root of the `my_datacenter` data center. Once connected using the username and password we provide, it runs the following command: **\$ /bin/cat results_file**

In the `/tmp` folder on the VM, the output of running the command is then registered as `shell_results` so that we can use it later.

The vmware_vm_vm_drs_rule module

With this module, you can configure VMware DRS Affinity rules. These allow you to control the placement of virtual machines within your cluster:

- name: Create DRS Affinity Rule for VM-VM

```
vmware_vm_vm_drs_rule:
```

```
hostname: "{{ vsphere_host }}"
username: "{{ vsphere_username }}"
password: "{{ vsphere_password }}"
cluster_name: "cluster"
vms: "{{ item }}"
drs_rule_name: ""
enabled: "True"
mandatory: "True"
affinity_rule: "True"
with_items:
- "example_vm"
- "another_example_vm"
```

In the previous example, we are creating a rule that will result in the VMs `example_vm` and `another_example_vm` never running on the same physical host.

The vmware_vm_vss_dvs_migrate module

This module migrates the named virtual machine from standard vSwitches, which are tied to a single host, to a distributed vSwitch that's available across your cluster:

```
- name: migrate vm to dvs
  vmware_vm_vss_dvs_migrate"
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    vm_name: "example_vm"
    dvportgroup_name: "example_portgroup"
    delegate_to: localhost
```

As you can see, we are moving the `example_vm` from the standard vSwitch to the distributed vSwitch called `example_portgroup`.

The vsphere_copy module

This module has a single purpose—to copy a local file to a remote data store:

```
- name: copy file to datastore
vsphere_copy:
  hostname: "{{ vsphere_host }}"
  username: "{{ vsphere_username }}"
  password: "{{ vsphere_password }}"
  src: "/path/to/local/file"
  datacenter: "my_datacenter"
  datastore: "my_datastore"
  path: "path/to/remove/file"
  transport: local
```

As you can see, we are copying the file from `/path/to/local/file` to `path/to/remove/file` in the `my_datastore` data store which is hosted in the `my_datacenter` data center.

The vsphere_guest module

This module has been deprecated and will be removed in Ansible 2.9; it is recommended you use the `vmware_guest` module instead.

VMware vCentre

VMware vCentre is an important component of the VMware vSphere suite; it enables the clustering of features such as vMotion, the VMware Distributed Resource Scheduler, and VMware High Availability.

The vcenter_folder module

This module enables vCenter folder management. For example, the following sample creates a folder for your virtual machines:

```
- name: Create a vm folder
vcenter_folder:
  hostname: "{{ item.ip }}"
  username: "{{ item.username }}"
  password: "{{ item.password }}"
  datacenter_name: "my_datacenter"
  folder_name: "virtual_machines"
  folder_type: "vm"
  state: "present"
```

Here is an example of creating a folder for your hosts:

```
- name: Create a host folder
vcenter_folder:
  hostname: "{{ item.ip }}"
  username: "{{ item.username }}"
  password: "{{ item.password }}"
  datacenter_name: "my_datacenter"
  folder_name: "hosts"
  folder_type: "host"
  state: "present"
```

The vcenter_license module

This module lets you add and remove VMware vCenter licenses:

```
- name: Add a license
  vcenter_license:
    hostname: "{{ item.ip }}"
    username: "{{ item.username }}"
    password: "{{ item.password }}"
    license: "123abc-456def-abc456-def123"
    state: "present"
    delegate_to: localhost
```

The vmware_guest module

This module allows you to launch and manage virtual machines within your VMware cluster; the following example shows how you would launch a VM using a template:

```
- name: Create a VM from a template
  vmware_guest:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter: "my-datacenter"
    folder: "/vms"
    name: "yet_another_example_vm"
    state: "poweredon"
    template: "centos7-x86_64-1804"
    disk:
      - size_gb: "40"
        type: "thin"
        datastore: "my_datastore"
    hardware:
      memory_mb: "4048"
      num_cpus: "4"
      max_connections: "3"
      hotadd_cpu: "True"
      hotremove_cpu: "True"
      hotadd_memory: "True"
    networks:
      - name: "VM Network"
        ip: "192.168.1.100"
        netmask: "255.255.255.0"
        gateway: "192.168.1.254"
        dns_servers:
          - "192.168.1.1"
          - "192.168.1.2"
    wait_for_ip_address: "yes"
  delegate_to: "localhost"
  register: deploy
```

As you can see, we have quite a lot of control over the VM and how it is configured. There are separate sections for the hardware, network, and storage configuration; we will look at this module in a little more detail at the end of this chapter.

The vmware_guest_facts module

This module gathers information on already created VMs:

```
- name: Gather facts on the yet_another_example_vm VM
  vmware_guest_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter: "my-datacenter"
    folder: "/vms"
    name: "yet_another_example_vm"
    delegate_to: localhost
    register: facts
```

The previous example gathers lots of information on the machine we defined in the previous section and registers the information as a variable so that we can use it elsewhere in our playbook run.

The vmware_guest_file_operation module

This module was introduced in Ansible 2.5; it allows you to add and fetch files from a VM without the need for the VM to be on a network. It also allows you to create folders within the VM. The following example creates a directory within the VM:

```
- name: create a directory on a vm
  vmware_guest_file_operation:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter: "my-datacenter"
    vm_id: "yet_another_example_vm"
    vm_username: "root"
    vm_password: "supersecretpassword"
    directory:
      path: "/tmp/imported/files"
      operation: "create"
      recurse: "yes"
  delegate_to: localhost
```

The following example copies a file called `config.zip` from our Ansible host to the directory created previously:

```
- name: copy file to vm
  vmware_guest_file_operation:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter: "my-datacenter"
    vm_id: "yet_another_example_vm"
    vm_username: "root"
    vm_password: "supersecretpassword"
    copy:
      src: "files/config.zip"
      dest: "/tmp/imported/files/config.zip"
      overwrite: "False"
  delegate_to: localhost
```


The vmware_guest_find module

We know the name of the folder in which the VM has been running. If we didn't, or if it changed for any reason, we could use the `vmware_guest_find` module to dynamically discover the location:

```
- name: Find vm folder location
  vmware_guest_find:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    name: "yet_another_example_vm"
  register: vm_folder
```

The name of the folder will be registered as `vm_folder`.

The vmware_guest_powerstate module

This module is self-explanatory; it is used to manage the power stage of your VM. The following example power cycles a VM:

```
- name: Powercycle a vm
  vmware_guest_powerstate:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    folder: "/vms"
    name: "yet_another_example_vm"
    state: "reboot-guest"
  delegate_to: localhost
```

You can also schedule changes to the power state. The following example powers down the VM at 9 a.m. on the 1st April 2019:

```
- name: April fools
  vmware_guest_powerstate:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    folder: "/vms"
    name: "yet_another_example_vm"
    state: "powered-off"
    scheduled_at: "01/04/2019 09:00"
  delegate_to: localhost
```

Not that I would ever do something like that!

The vmware_guest_snapshot module

This module allows you to manage your VM snapshots; for example, the following creates a snapshot:

```
- name: Create a snapshot
  vmware_guest_snapshot:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter: "my-datacenter"
    folder: "/vms"
    name: "yet_another_example_vm"
    snapshot_name: "pre-patching"
    description: "snapshot made before patching"
    state: "present"
  delegate_to: localhost
```

As you can see from the previous example, this snapshot is being taken because we are about to patch the VM. If the patching goes as expected, then we can run the following task:

```
- name: Remove a snapshot
  vmware_guest_snapshot:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter: "my-datacenter"
    folder: "/vms"
    name: "yet_another_example_vm"
    snapshot_name: "pre-patching"
    state: "remove"
  delegate_to: localhost
```

If nothing goes as planned and the patching breaks our VM, then not to worry, we have a snapshot we can revert to:

```
- name: Revert to a snapshot
  vmware_guest_snapshot:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter: "my-datacenter"
    folder: "/vms"
    name: "yet_another_example_vm"
    snapshot_name: "pre-patching"
    state: "revert"
  delegate_to: localhost
```

Fingers crossed that you never have to revert to a snapshot (unless it is planned).

The vmware_guest_tools_wait module

The final module of this section is another self-explanatory one; it simply waits for VMware tools to become available and then gathers facts on the machine:

```
- name: Wait for VMware tools to become available by name
  vmware_guest_tools_wait:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    folder: "/vms"
    name: "yet_another_example_vm"
    delegate_to: localhost
    register: facts
```

VMware tools is an application that runs inside the VM. Once it starts, it allows VMware to interact with the VM, allowing modules such as `vmware_guest_file_operation` and `vmware_vm_shell` to function.

VMware ESXi

At the heart of most VMware installations is a number of VMware ESXi hosts. VMware ESXi is the type 1 hypervisor that enables VMs to run. Ansible provides several modules that allow you to configure and interact with your VMware ESXi hosts.

The vmware_dns_config module

This module lets you manage the DNS aspects of your ESXi host; it lets you set the hostname, domain, and the DNS resolvers:

```
- name: Configure the hostname and dns servers
  local_action
    module: vmware_dns_config:
      hostname: "{{ esxi_host }}"
      username: "{{ esxi_username }}"
      password: "{{ esxi_password }}"
      validate_certs: "no"
      change_hostname_to: "esxi-host-01"
      domainname: "my-domain.com"
      dns_servers:
        - "8.8.8.8"
        - "8.8.4.4"
```

In the previous example, we are setting the FQDN of the host to be `esxi-host-01.my-domain.com` and also configuring the host to use Google public DNS resolvers.

The vmware_host_dns_facts module

A simple module that gathers facts on the DNS configuration of your VMware ESXi hosts follows:

```
- name: gather facts on dns config
  vmware_host_dns_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    cluster_name: "my_cluster"
```

The vmware_host module

You can use this module to attach your ESXi host to a vCenter: - name: add an esxi host to vcenter

vmware_host:

```
hostname: "{{ vsphere_host }}"
username: "{{ vsphere_username }}"
password: "{{ vsphere_password }}"
datacenter_name: "my-datacenter"
cluster_name: "my_cluster"
esxi_hostname: "{{ exsi_host }}"
esxi_username: "{{ exsi_username }}"
esxi_password: "{{ exsi_password }}"
state: present
```

You can also use the module to reattach a host to your vCenter cluster: - name: reattach an esxi host to vcenter

vmware_host:

```
hostname: "{{ vsphere_host }}"
username: "{{ vsphere_username }}"
password: "{{ vsphere_password }}"
datacenter_name: "my-datacenter"
cluster_name: "my_cluster"
esxi_hostname: "{{ exsi_host }}"
esxi_username: "{{ exsi_username }}"
esxi_password: "{{ exsi_password }}"
state: reconnect
```

You can also remove a host from your vCenter cluster: - name: remove an esxi host to vcenter

vmware_host:

```
hostname: "{{ vsphere_host }}"
username: "{{ vsphere_username }}"
password: "{{ vsphere_password }}"
datacenter_name: "my-datacenter"
```

```
cluster_name: "my_cluster"
esxi_hostname: "{{ exsi_host }}"
esxi_username: "{{ exsi_username }}"
esxi_password: "{{ exsi_password }}"
state: absent
```

The vmware_host_facts module

As you may have guessed, this module gathers facts about the VMware ESXi hosts within your vSphere or vCenter cluster:

```
- name: Find out facts on the esxi hosts
  vmware_host_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
  register: host_facts
  delegate_to: localhost
```

The vmware_host_acceptance module

With this module, you can manage the acceptance level of your VMware ESXi host. There are four acceptance levels supported by VMware, and these are:

- VMwareCertified
- VMwareAccepted
- PartnerSupported
- CommunitySupported

These levels control the VIBs which can be installed on the ESXi host; a VIB is an ESXi software package. This typically dictates the level of support you will receive from either VMware or a VMware partner. The following task will set the acceptance level to CommunitySupported for all of the ESXi hosts in the named cluster:

```
- name: Set acceptance level for all esxi hosts in the cluster
  vmware_host_acceptance:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    cluster_name: "my_cluster"
    acceptance_level: "community"
    state: present
  register: cluster_acceptance_level
```

The vmware_host_config_manager module

Using this module, you can set configuration options on your individual VMware ESXi hosts, for example:

```
- name: Set some options on our esxi host
  vmware_host_config_manager:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    esxi_hostname: "{{ exsi_host }}"
    options:
      "Config.HostAgent.log.level": "verbose"
      "Annotations.WelcomeMessage": "Welcome to my awesome Ansible managed ESXi host"
      "Config.HostAgent.plugins.solo.enableMob": "false"
```

Ansible maps the advanced configurations options from your VMware host, so for more information on the available options, please consult your documentation.

The vmware_host_datastore module

This module gives you the power to mount and dismount datastores on your VMware ESXi hosts; in the following example, we are mounting three datastores on all of the VMware ESXi hosts within our inventory:

```
- name: Mount datastores on our cluster
  vmware_host_datastore:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter_name: "my-datacenter"
    datastore_name: "{{ item.name }}"
    datastore_type: "{{ item.type }}"
    nfs_server: "{{ item.server }}"
    nfs_path: "{{ item.path }}"
    nfs_ro: "no"
    esxi_hostname: "{{ inventory_hostname }}"
    state: present
    delegate_to: localhost
    with_items:
      - { "name": "ds_vol01", "server": "nas", "path": "/mnt/ds_vol01", 'type': "nfs"}
      - { "name": "ds_vol02", "server": "nas", "path": "/mnt/ds_vol02", 'type': "nfs"}
      - { "name": "ds_vol03", "server": "nas", "path": "/mnt/ds_vol03", 'type': "nfs"}
```

The vmware_host_firewall_manager module

This module lets you configure the firewall rules on your VMware ESXi hosts:

```
- name: set some firewall rules on the esxi hosts
  vmware_host_firewall_manager:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    esxi_hostname: "{{ inventory_hostname }}"
    rules:
      - name: "vvold"
        enabled: "True"
      - name: "CIMHttpServer"
        enabled: "False"
```

The previous example enables `vvold` and disables `CIMHttpServer` on each of the VMware ESXi hosts in the host inventory.

The vmware_host_firewall_facts module

As you will have already guessed, this module, like the other facts modules, is used to gather facts on the firewall configuration of all of the hosts within our cluster:

```
- name: Get facts on all cluster hosts
  vmware_host_firewall_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    cluster_name: "my_cluster"
```

It can also gather for just a single host:

```
- name: Get facts on a single host
  vmware_host_firewall_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    esxi_hostname: "{{ exsi_host }}"
```

The vmware_host_lockdown module

This module comes with a warning which reads: This module is destructive as administrator permission are managed using APIs used, please read options carefully and proceed..

You can lock down a host with the following code:

```
- name: Lockdown an ESXi host
  vmware_host_lockdown:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    esxi_hostname: "{{ exsi_host }}"
    state: "present"
```

You can take a host out of lockdown using:

```
- name: Remove the lockdown on an ESXi host
  vmware_host_lockdown:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    esxi_hostname: "{{ exsi_host }}"
    state: "absent"
```

As mentioned previously, this module may have some unexpected side effects, so you may want to do this on a per-host basis, rather than using the following option, which will put all hosts in the named cluster into lockdown:

```
- name: Lockdown all the ESXi hosts
  vmware_host_lockdown:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    cluster_name: "my_cluster"
    state: "present"
```

The vmware_host_ntp module

With this module, you can manage the NTP settings for each of the VMware ESXi hosts. The following example configures all of the hosts to use the same set of NTP servers:

```
- name: Set NTP servers for all hosts
  vmware_host_ntp:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    cluster_name: "my_cluster"
    state: present
    ntp_servers:
      - 0.pool.ntp.org
      - 1.pool.ntp.org
      - 2.pool.ntp.org
```

The vmware_host_package_facts module

This module can be used to gather facts on all of the VMware ESXi hosts within your cluster:

```
- name: Find out facts about the packages on all the ESXi hosts
  vmware_host_package_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    cluster_name: "my_cluster"
  register: cluster_packages
```

Like the rest of the facts modules, it can also gather for just a single host:

```
- name: Find out facts about the packages on a single ESXi host
  vmware_host_package_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    esxi_hostname: "{{ exsi_host }}"
  register: host_packages
```

The vmware_host_service_manager module

This module lets you manage ESXi servers on either all of your cluster members or individual hosts:

```
- name: Start the ntp service on all esxi hosts
  vmware_host_service_manager:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    cluster_name: "my_cluster"
    service_name: "ntpd"
    service_policy: "automatic"
    state: "present"
```

In this example, we are starting the NTP service (`service_name`) on all hosts within the cluster; as we have the `service_policy` defined as `automatic`, the service will only start if the services corresponding to the firewall rule have been configured. If we wanted the service to start regardless of the firewall rule, then we could set `service_policy` to `on` or if wanted to the service to be stopped, then `service_policy` should be set to `off`.

The vmware_host_service_facts module

With this module, you can find out facts on the services configured on each of the VMware ESXi hosts within the cluster:

```
- name: Find out facts about the services on all the ESXi hosts
  vmware_host_service_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    cluster_name: "my_cluster"
  register: cluster_services
```

The vmware_datastore_facts module

This is one of the old-style facts modules which can be used to gather information on the data stores configured in the data center:

```
- name: Find out facts about the datastores
  vmware_datastore_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter: "my_datacenter"
  delegate_to: localhost
  register: datastore_facts
```

You may notice that there is a little bit of a difference in the syntax between this and the previous facts modules.

The vmware_host_vmnics_facts module

From an old-style fact module back to a new one, this module can be used to gather information on the physical network interfaces on your VMware ESXi hosts:

```
- name: Find out facts about the vmnics on all the ESXi hosts
  vmware_host_vmnics_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter: "my_datacenter"
  register: cluster_vmnics
```

For a single ESXi host, we could use the following task:

```
- name: Find out facts about the vmnics on a single ESXi host
  vmware_host_vmnics_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    esxi_hostname: "{{ exsi_host }}"
  register: host_vmnics
```

The vmware_local_role_manager module

Using this module, you can configure roles on your cluster; these roles can be used to assign privileges. In the following example, we are assigning a few privileges to the `vmware_qa` role:

```
- name: Add a local role
  vmware_local_role_manager:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    local_role_name: "vmware_qa"
    local_privilege_ids: [ "Folder.Create", "Folder.Delete" ]
    state: "present"
```

The vmware_local_user_manager module

With this module, you can manage local users by adding users and setting their passwords:

```
- name: Add local user to ESXi
  vmware_local_user_manager:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    local_user_name: "myuser"
    local_user_password: "my-super-secret-password"
    local_user_description: "An example user added by Ansible"
  delegate_to: "localhost"
```

The vmware_cfg_backup module

Using this module, you can create a backup of a VMware ESXi host configuration:

```
- name: Create an esxi host configuration backup
  vmware_cfg_backup:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    state: "saved"
    dest: "/tmp/"
    esxi_hostname: "{{ exsi_host }}"
    delegate_to: "localhost"
    register: cfg_backup
```

Please note that this module will automatically put the host into maintenance and then save the configuration too. In the preceding example, from `/tmp`, you can use the information registered to grab a copy of the backup using the `fetch` module.

You can also use this module to restore a configuration:

```
- name: Restore an esxi host configuration backup
  vmware_cfg_backup:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    state: "loaded"
    dest: "/tmp/my-host-backup.tar.gz"
    esxi_hostname: "{{ exsi_host }}"
    delegate_to: "localhost"
```

Finally, you can also reset a host configuration back to its default settings by running the following code:

```
- name: Reset a host configuration to the default values
  vmware_cfg_backup:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}
```

```
password: "{{ vsphere_password }}"
validate_certs: "no"
state: "absent"
esxi_hostname: "{{ exsi_host }}"
delegate_to: "localhost"
```

The vmware_vmkernel module

This module allows you to add VMkernel interfaces, which are also known as Virtual NICs on a host:

```
- name: Add management port with a static ip
  vmware_vmkernel:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    esxi_hostname: "{{ exsi_host }}"
    vswitch_name: "my_vSwitch"
    portgroup_name: "my_portgroup"
    vlan_id: "the_vlan_id"
    network:
      type: "static"
      ip_address: "192.168.127.10"
      subnet_mask: "255.255.255.0"
    state: "present"
    enable_mgmt: "True"
```

In the previous example, we added a management interface; there are also the following options:

- `enable_ft`: Enables an interface for fault tolerance traffic
- `enable_mgmt`: Enables an interface for management traffic
- `enable_vmotion`: Enables an interface for VMotion traffic
- `enable_vsan`: Enables an interface for VSAN traffic

The vmware_vmkernel_facts module

Yet another facts module, this is a new-style module; you have probably already guessed what the task will look like:

```
- name: Find out facts about the vmkernel on all the ESXi hosts
  vmware_vmkernel_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    cluster_name: "my_cluster"
    register: cluster_vmk

- name: Find out facts about the vmkernel on a single ESXi host
  vmware_vmkernel_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    esxi_hostname: "{{ exsi_host }}"
    register: host_vmk
```

The vmware_target_canonical_facts module

Using this module, you can find out the canonical name of an SCSI target; all you need to know is the ID of the target device:

```
- name: Get Canonical name of SCSI device
  vmware_target_canonical_facts"
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    target_id: "6"
    register: canonical_name
```

The vmware_vmotion module

You can use this module to perform a vMotion of a virtual machine from one VMware ESXi host to another:

```
- name: Perform vMotion of VM
  vmware_vmotion
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    vm_name: "example_vm"
    destination_host: "esxi-host-02"
  delegate_to: "localhost"
  register: vmotion_results
```

The vmware_vsan_cluster module

You can use this module to register a VSAN cluster; this module works slightly differently to the other modules in this chapter in that you first need to generate a cluster UUID on a single host before deploying the VSAN on the remaining hosts using the UUIDs generated.

The following tasks assume you have a host group called `esxi_hosts`, which contains more than one host. The first task assigns the VSAN to the first host in the group, then registers the results:

```
- name: Configure VSAN on first host in the group
  vmware_vsan_cluster:
    hostname: "{{ groups['esxi_hosts'][0] }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
  register: vsan_cluster
```

The results, registered as `vsan_cluster`, contain the VSAN cluster UUID which we will need for the rest of the hosts in the group. The following code configures the cluster on the remaining hosts, skipping the original host:

```
- name: Configure VSAN on the remaining hosts in the group
  vmware_vsan_cluster:
    hostname: "{{ item }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    cluster_uuid: "{{ vsan_cluster.cluster_uuid }}"
  with_items: "{{ groups['esxi_hosts'][1:] }}"
```

The vmware_vswitch module

Using this module, you can add or remove a **VMware Standard Switch (vSwitch)** to an ESXi host:

```
- name: Add a vSwitch
  vmware_vswitch:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    switch: "vswitch_name"
    nics:
      - "vmnic1"
      - "vmnic2"
    mtu: "9000"
    delegate_to: "localhost"
```

In this example, we have added a vSwitch that's attached to multiple vmnics.

The vmware_drs_rule_facts module

You can use this module to gather facts on the DRS configured in your entire cluster or single data center:

```
- name: Find out facts about drs on all the hosts in the cluster
  vmware_drs_rule_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    cluster_name: "my_cluster"
    delegate_to: "localhost"
    register: cluster_drs

- name: Find out facts about drs in a single data center
  vmware_drs_rule_facts:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter: "my_datacenter"
    delegate_to: "localhost"
    register: datacenter_drs
```

The vmware_dvswitch module

This module allows you to create and remove distributed vSwitches:

```
- name: Create dvswitch
  vmware_dvswitch:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter: "my_datacenter"
    switch_name: "my_dvSwitch"
    switch_version: "6.0.0"
    mtu: "9000"
    uplink_quantity: "2"
    discovery_proto: "lldp"
    discovery_operation: "both"
    state: present
    delegate_to: "localhost"
```

The vmware_dvs_host module

Using this module, you can add or remove a host from the distributed virtual switch:

```
- name: Add host to dvs
  vmware_dvs_host:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    esxi_hostname: "{{ exsi_host }}"
    switch_name: "my_dvSwitch"
    vmnics:
      - "vmnic1"
      - "vmnic2"
    state: "present"
  delegate_to: "localhost"
```

The vmware_dvs_portgroup module

With this module, you can manage your DVS port groups:

```
- name: Create a portgroup with vlan
  vmware_dvs_portgroup:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    portgroup_name: "my_portgroup_vlan123"
    switch_name: "my_dvSwitch"
    vlan_id: "123"
    num_ports: "120"
    portgroup_type: "earlyBinding"
    state: "present"
    delegate_to: "localhost"
```

The vmware_maintenancemode module

Using this module, you can place a host into maintenance mode. The following example shows you how to put a host into maintenance while maintaining object availability on the VSAN:

```
- name: Put host into maintenance mode
  vmware_maintenancemode:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    esxi_hostname: "{{ exsi_host }}"
    vsan: "ensureObjectAccessibility"
    evacuate: "yes"
    timeout: "3600"
    state: "present"
    delegate_to: "localhost"
```

The vmware_portgroup module

This module lets you create a VMware port group on the hosts in a given cluster:

```
- name: Create a portgroup with vlan
  vmware_portgroup:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    cluster_name: "my_cluster"
    switch_name: "my_switch"
    portgroup_name: "my_portgroup_vlan123"
    vlan_id: "123"
    delegate_to: "localhost"
```

The vmware_resource_pool module

With this, the final module we are going to look at, you can create a resource pool. An example of how to do this follows:

```
- name: Add resource pool
  vmware_resource_pool:
    hostname: "{{ vsphere_host }}"
    username: "{{ vsphere_username }}"
    password: "{{ vsphere_password }}"
    validate_certs: "no"
    datacenter: "my_datacenter"
    cluster: "my_new_cluster"
    resource_pool: "my_resource_pool"
    mem_shares: "normal"
    mem_limit: "-1"
    mem_reservation: "0"
    mem_expandable_reservations: "True"
    cpu_shares: "normal"
    cpu_limit: "-1"
    cpu_reservation: "0"
    cpu_expandable_reservations: "True"
    state: present
  delegate_to: "localhost"
```

An example playbook

Before we complete this chapter, I am going to share an example playbook I wrote for deploying a small number of virtual machines in a VMware cluster. The idea with the project was to launch seven virtual machines into a customer's network which are as follows:

- One Linux jump host
- One NTP server
- One load balancer
- Two web servers
- Two database servers

The VMs all had to be built from an existing template; unfortunately, this template was built with a hardcoded gateway IP address of 192.168.1.254 in the `/etc/sysconfig/network` file. This meant that for these machines to correctly appear on the network, I had to make changes to each of the virtual machines once they had been launched.

I started off by setting up a file called `vmware.yml` in my `group_vars` folder; this contained the information needed to connect to my VMware installation and also the default credentials for the VMs:

```
vcenter:
  host: "cluster.cloud.local"
  username: "svc_ansible@cloud.local"
  password: "mymegasecretpassword"

  wait_for_ip_address: "yes"
  machine_state: "poweredon"

deploy:
  datacenter: "Cloud DC4"
  folder: "/vm/Ansible"
  resource_pool: "/Resources/"

vm_shell:
  username: "root"
  password: "hushdonttell"
  cwd: "/tmp"
  cmd: "/bin/sed"
  args: "-i 's/GATEWAY=192.168.1.254/GATEWAY={{ item.gateway }}/g' /etc/sysconfig/network"
```

I will be using the variables defined in both of the roles I will be running. Next up, there is the `group_vars/vms.yml` file; this contains all of the information needed to launch the virtual machines in my VMWare environment:

```
vm:
  - name: "NTPSERVER01"
    machine_name: "ntpserver01"
    machine_template: "RHEL6_TEMPLATE"
    guest_id: "rhel6_64Guest"
    host: "compute-host-01.cloud.local"
    cpu: "1"
    ram: "1024"
    networks:
      - name: "CLOUD-CUST|Customer|MANGMENT"
        ip: "192.168.99.10"
        netmask: "255.255.255.0"
        device_type: "vmxnet3"
    gateway: "192.168.99.254"
    disk:
      - size_gb: "30"
        type: "thin"
        datastore: "cust_sas_esx_nfs_01"
  - name: "JUMPHOST01"
    machine_name: "jumphost01"
    machine_template: "RHEL6_TEMPLATE"
    guest_id: "rhel6_64Guest"
    host: "compute-host-02.cloud.local"
    cpu: "1"
    ram: "1024"
    networks:
      - name: "CLOUD-CUST|Customer|MANGMENT"
        ip: "192.168.99.20"
        netmask: "255.255.255.0"
        device_type: "vmxnet3"
    gateway: "192.168.99.254"
    disk:
      - size_gb: "30"
        type: "thin"
        datastore: "cust_sas_esx_nfs_01"
  - name: "LOADBALANCER01"
    machine_name: "loadbalancer01"
    machine_template: "LB_TEMPLATE"
    guest_id: "rhel6_64Guest"
    host: "compute-host-03.cloud.local"
    cpu: "4"
    ram: "4048"
    networks:
      - name: "CLOUD-CUST|Customer|DMZ"
        ip: "192.168.98.100"
        netmask: "255.255.255.0"
        device_type: "vmxnet3"
    gateway: "192.168.99.254"
    disk:
      - size_gb: "30"
        type: "thin"
        datastore: "cust_sas_esx_nfs_02"
  - name: "WEBSERVER01"
    machine_name: "webserver01"
    machine_template: "RHEL6_TEMPLATE"
    guest_id: "rhel6_64Guest"
    host: "compute-host-01.cloud.local"
```

```

cpu: "1"
ram: "1024"
networks:
  - name: "CLOUD-CUST|Customer|APP"
    ip: "192.168.100.10"
    netmask: "255.255.255.0"
    device_type: "vmxnet3"
gateway: "192.168.100.254"
disk:
  - size_gb: "30"
    type: "thin"
    datastore: "cust_sas_esx_nfs_01"
- name: "WEB SERVER02"
  machine_name: "webserver02"
  machine_template: "RHEL6_TEMPLATE"
  guest_id: "rhel6_64Guest"
  host: "compute-host-02.cloud.local"
  cpu: "1"
  ram: "1024"
  networks:
    - name: "CLOUD-CUST|Customer|APP"
      ip: "192.168.100.20"
      netmask: "255.255.255.0"
      device_type: "vmxnet3"
    gateway: "192.168.100.254"
  disk:
    - size_gb: "30"
      type: "thin"
      datastore: "cust_sas_esx_nfs_02"
- name: "DB SERVER01"
  machine_name: "dbserver01"
  machine_template: "RHEL6_TEMPLATE"
  guest_id: "rhel6_64Guest"
  host: "compute-host-10.cloud.local"
  cpu: "8"
  ram: "32000"
  networks:
    - name: "CLOUD-CUST|Customer|DB"
      ip: "192.168.101.10"
      netmask: "255.255.255.0"
      device_type: "vmxnet3"
    gateway: "192.168.101.254"
  disk:
    - size_gb: "30"
      type: "thin"
      datastore: "cust_sas_esx_nfs_01"
    - size_gb: "250"
      type: "thick"
      datastore: "cust_ssd_esx_nfs_01"
    - size_gb: "250"
      type: "thick"
      datastore: "cust_ssd_esx_nfs_01"
    - size_gb: "250"
      type: "thick"
      datastore: "cust_ssd_esx_nfs_01"
- name: "DB SERVER02"
  machine_name: "dbserver02"
  machine_template: "RHEL6_TEMPLATE"
  guest_id: "rhel6_64Guest"
  host: "compute-host-11.cloud.local"
  cpu: "8"
  ram: "32000"
  networks:
    - name: "CLOUD-CUST|Customer|DB"

```

```

ip: "192.168.101.11"
netmask: "255.255.255.0"
device_type: "vmxnet3"
gateway: "192.168.101.254"
disk:
  - size_gb: "30"
    type: "thin"
    datastore: "cust_sas_esx_nfs_02"
  - size_gb: "250"
    type: "thick"
    datastore: "cust_ssd_esx_nfs_02"
  - size_gb: "250"
    type: "thick"
    datastore: "cust_ssd_esx_nfs_02"
  - size_gb: "250"
    type: "thick"
    datastore: "cust_ssd_esx_nfs_02"

```

As you can see, I am defining the specs, networks, and storage for all seven VMs; where possible, I am thin provisioning storage and also making sure that where there is more than one virtual machine in a role I am using different storage pools.

Now that I have all of the details needed for my virtual machines, I can create the roles. First of all, there is `roles/vmware/tasks/main.yml`:

```

- name: Launch the VMs
  vmware_guest:
    hostname: "{{vcenter.host}}"
    username: "{{ vcenter.username }}"
    password: "{{ vcenter.password }}"
    validate_certs: no
    datacenter: "{{ deploy.datacenter }}"
    folder: "{{ deploy.folder }}"
    name: "{{ item.machine_name | upper }}"
    state: "{{ machine_state }}"
    guest_id: "{{ item.guest_id }}"
    esxi_hostname: "{{ item.host }}"
    hardware:
      memory_mb: "{{ item.ram }}"
      num_cpus: "{{ item.cpu }}"
    networks: "{{ item.networks }}"
    disk: "{{ item.disk }}"
    template: "{{ item.machine_template }}"
    wait_for_ip_address: "{{ wait_for_ip_address }}"
    customization:
      hostname: "{{ item.machine_name | lower }}"
  with_items: "{{ vm }}"

```

As you can see, this task loops through the items in the `vm` variable; once the virtual machine has been launched, it will wait for the IP address I have assigned to be available within VMware. This makes sure that the virtual machine is correctly launched before proceeding with either launching the next virtual machine or moving on to the next role.

The next role resolves the problem of `192.168.1.254` being hardcoded as the gateway in the virtual machine template; it can be found in `roles/fix/tasks/main.yml`. There are two tasks in the role; the first one updates the gateway to the correct one for the network the virtual machine has been launched in:

```
- name: Sort out the wrong IP address in the /etc/sysconfig/network file on the vms
  vmware_vm_shell:
    hostname: "{{vcenter.host}}"
    username: "{{ vcenter.username }}"
    password: "{{ vcenter.password }}"
    validate_certs: no
    vm_id: "{{ item.machine_name | upper }}"
    vm_username: "{{ vm_shell.username }}"
    vm_password: "{{ vm_shell.password }}"
    vm_shell: "{{ vm_shell.cmd }}"
    vm_shell_args: " {{ vm_shell.args }}"
    vm_shell_cwd: "{{ vm_shell.cwd }}"
  with_items: "{{ vm }}"
```

As you can see, this loops through the list of virtual machines defined as `vm` and executes the `sed` command we defined in the `group_vars/vmware.yml` file. Once this task has run, we need to run one more task. This one restarts networking on all of the virtual machines so that the changes to the gateway are picked up:

```
- name: Restart networking on all VMs
  vmware_vm_shell:
    hostname: "{{vcenter.host}}"
    username: "{{ vcenter.username }}"
    password: "{{ vcenter.password }}"
    validate_certs: no
    vm_id: "{{ item.machine_name | upper }}"
    vm_username: "{{ vm_shell.username }}"
    vm_password: "{{ vm_shell.password }}"
    vm_shell: "/sbin/service"
    vm_shell_args: "network restart"
  with_items: "{{ vm }}"
```

When I ran the playbook, it took about 30 minutes to run, but at the end of it I had the seven virtual machines launched and available to work, so I was then able to run the set of playbooks which bootstrapped the environment ready so that I could hand them over to the customer for them to deploy their application.

Summary

As you have seen from the very long list of modules, you can do pretty much most of the common tasks you would be doing as a VMware administrator using Ansible. Add to this the modules we looked at in [Chapter 7, *The Core Network Modules*](#), for managing network equipment, and also modules such as the ones that support NetApp storage devices, and you can build some quite complex playbooks which span the physical devices, VMware elements, and even the virtual machines running within your virtualized infrastructure.

In the next chapter, we will see how to build our Windows servers locally using Vagrant and then take our playbooks to the public cloud.

Questions

1. Which Python module do you need to install on your Ansible controller to be able to interact with vSphere?
2. True or false: `vmware_dns_config` only allows you to set the DNS resolvers on your ESXi hosts.
3. Name two of the modules we have covered that can be used to launch virtual machines; there are three, but one is deprecated.
4. Which of the modules we have looked at would you use to ensure that a virtual machine is fully available before progressing to a task that interacts with the VM via VMware?
5. True or false: It is possible to schedule the change of a power state using Ansible.

Further reading

For a good overview of VMware vSphere, I would recommend the following video: <https://www.youtube.com/watch?v=30vrKZYnzjM>.

Ansible Windows Modules

So far, we have been targeting Linux servers. In this chapter, we will take a look at the ever-growing collection of core Ansible modules that support and interact with Windows-based servers. Personally, coming from an almost exclusively macOS and Linux background, it felt a little odd to be using a tool that is not natively supported on Windows to manage Windows.

However, as I am sure you will agree by the end of the chapter, its developers have made the process of introducing Windows workloads into your playbook as seamless and familiar as possible.

In this chapter, we will learn how to build our Windows servers locally using Vagrant and then take our playbooks to the public cloud. We will cover:

- Enabling features in Windows
- Launching Windows instances in AWS
- Creating users
- Installing third-party packages using Chocolatey

Technical requirements

As in the previous chapter, we are going to be using Vagrant and also AWS. The Vagrant box we will be using contains an evaluation copy of Windows 2016. The Windows EC2 instance we will be launching in AWS will be fully licensed and therefore will carry an additional cost on top of the EC2 resource costs. As always, you can find complete copies of the playbooks in the accompanying repository at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter12>.

Up-and-running

For this section, we are going to use Vagrant to launch a Windows 2016 server just like we originally did in [Chapter 2, *Installing and Running Ansible*](#). Let's start by looking at the Vagrantfile we will be using to launch our host.

Vagrantfile

This `vagrantfile` doesn't look too dissimilar from the ones we have been using to launch Linux hosts:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

API_VERSION  = "2"
BOX_NAME      = "StefanScherer/windows_2016"
COMMUNICATOR = "winrm"
USERNAME      = "vagrant"
PASSWORD      = "vagrant"

Vagrant.configure(API_VERSION) do |config|
  config.vm.define "vagrant-windows-2016"
  config.vm.box = BOX_NAME
  config.vm.synced_folder ".", "/vagrant", disabled: true
  config.vm.network "forwarded_port", guest: 80, host: 8080
  config.vm.communicator = COMMUNICATOR
  config.winrm.username = USERNAME
  config.winrm.password = PASSWORD

  config.vm.provider "virtualbox" do |v|
    v.memory = "4048"
    v.cpus = "4"
    v.gui = true
  end

  config.vm.provider "vmware_fusion" do |v|
    v.vmx["memsize"] = "4048"
    v.vmx["numvcpus"] = "4"
  end
end
```

As you can see, we are replacing references to SSH Vagrant. We will be using the **Windows Remote Management (WinRM)** protocol, as well as Ansible, to interact with the virtual machine. By default, the `config.vm.communicator` is SSH, so overriding this with `winrm` means that we have to provide `config.winrm.username` and `config.winrm.password`.

Also, we are instructing Vagrant not to attempt to try and mount our local filesystem on the virtual machine, nor to add any additional IP addresses or network interfaces; instead, it should just forward the port from our localhost machine to the host.

Finally, we are mapping port `8080` on our local machine to port `80` on the Windows

host; more on that later in the chapter.

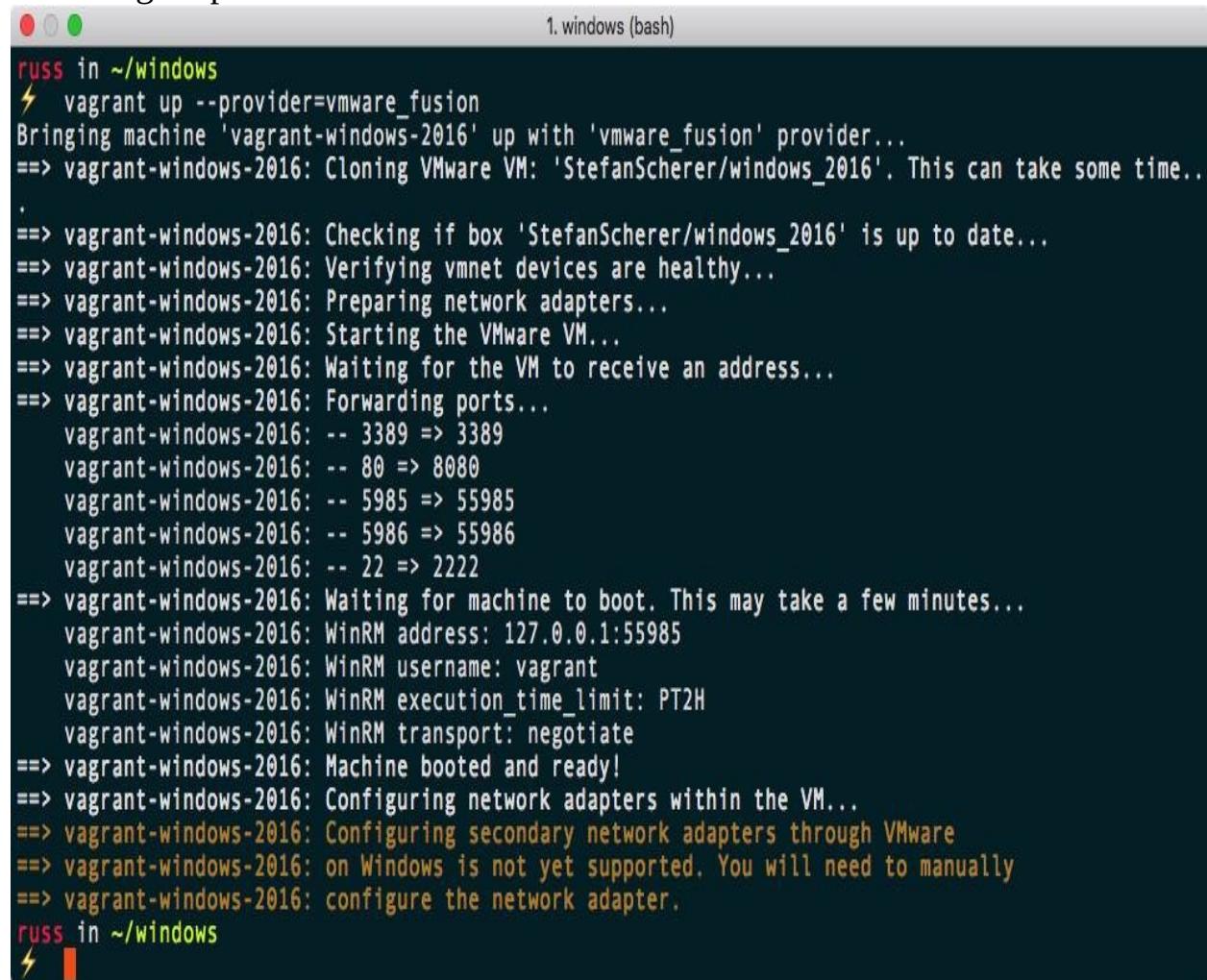
We can launch the host using one of the following commands:

```
| $ vagrant up
```

This will use VirtualBox, or we can use VMWare by running:

```
| $ vagrant up --provider=vmware_fusion
```

The Vagrant box we are using is several gigabytes in size so it will take a little while to download, but once downloaded you should see something like the following output:

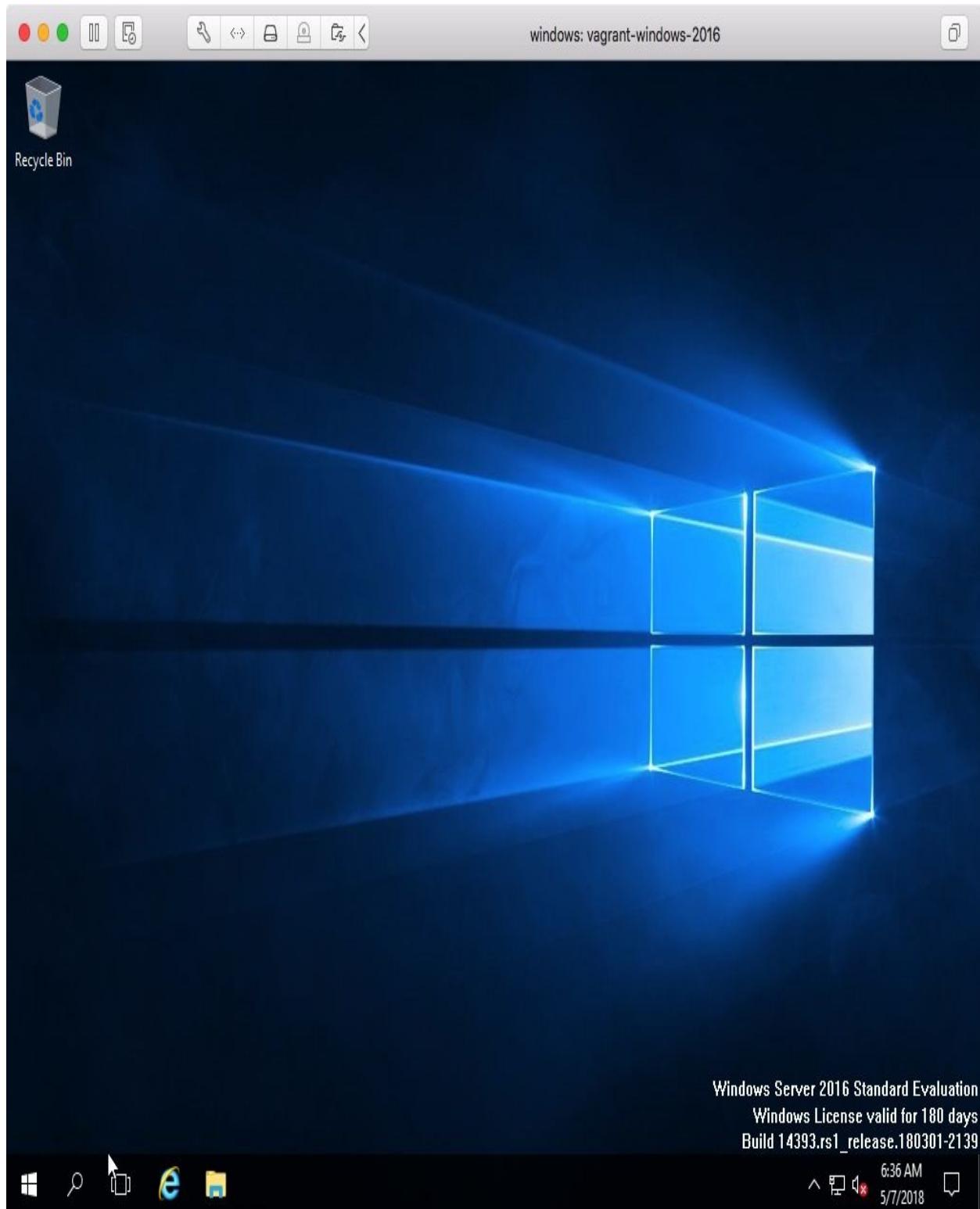


A screenshot of a terminal window titled "1. windows (bash)". The window shows the command "vagrant up" being run, followed by a series of status messages indicating the provisioning process. The messages include cloning the VM, checking for updates, verifying devices, preparing network adapters, starting the VM, and forwarding ports. It also notes that WinRM is active and provides the address. The terminal ends with a prompt "russ in ~/windows".

```
russ in ~/windows
⚡ vagrant up --provider=vmware_fusion
Bringing machine 'vagrant-windows-2016' up with 'vmware_fusion' provider...
==> vagrant-windows-2016: Cloning VMware VM: 'StefanScherer/windows_2016'. This can take some time...
.
==> vagrant-windows-2016: Checking if box 'StefanScherer/windows_2016' is up to date...
==> vagrant-windows-2016: Verifying vmnet devices are healthy...
==> vagrant-windows-2016: Preparing network adapters...
==> vagrant-windows-2016: Starting the VMware VM...
==> vagrant-windows-2016: Waiting for the VM to receive an address...
==> vagrant-windows-2016: Forwarding ports...
    vagrant-windows-2016: -- 3389 => 3389
    vagrant-windows-2016: -- 80 => 8080
    vagrant-windows-2016: -- 5985 => 55985
    vagrant-windows-2016: -- 5986 => 55986
    vagrant-windows-2016: -- 22 => 2222
==> vagrant-windows-2016: Waiting for machine to boot. This may take a few minutes...
    vagrant-windows-2016: WinRM address: 127.0.0.1:55985
    vagrant-windows-2016: WinRM username: vagrant
    vagrant-windows-2016: WinRM execution_time_limit: PT2H
    vagrant-windows-2016: WinRM transport: negotiate
==> vagrant-windows-2016: Machine booted and ready!
==> vagrant-windows-2016: Configuring network adapters within the VM...
==> vagrant-windows-2016: Configuring secondary network adapters through VMWare
==> vagrant-windows-2016: on Windows is not yet supported. You will need to manually
==> vagrant-windows-2016: configure the network adapter.

russ in ~/windows
⚡
```

Once the machine has launched, you will find that your virtual machine has opened a window and that the Windows desktop is accessible, as follows:



Just minimize this window for now as we will not be interacting with Windows directly. Closing the window may suspend and power down the virtual machine.

Now that we have our Windows host up-and-running, we need to install a few supporting Python modules, to allow Ansible to interact with it.

Ansible preparation

As already mentioned, Ansible will be using WinRM to interact with our Windows host.



WinRM provides access to a SOAP-like protocol called WS-Management. Unlike SSH, which provides the user with an interactive shell to manage the host, WinRM accepts scripts that are executed and then the results are passed back to you.

To be able to use WinRM, Ansible requires us to install a few different Python modules, Linux users can use the following command to install them: **\$ sudo pip install pywinrm[credssp]**

macOS users may need to perform the following command if they get errors about `pyopenssl` that can't be updated, as it is part of the core operating system: **\$ sudo pip install pywinrm[credssp] --ignore-installed pyOpenSSL**

Once installed, we should now be able to interact with our Windows host once we configure a host inventory file. The file, called `production`, looks like the following: `box1 ansible_host=localhost`

```
[windows]
box1
```

```
[windows:vars]
ansible_connection=winrm
ansible_user=vagrant
ansible_password=vagrant
ansible_port=55985
ansible_winrm_scheme=http
ansible_winrm_server_cert_validation=ignore
```

As you can see, we have removed all references to SSH and replaced them with WinRM (`ansible_connection`). Again, we have to provide a username (`ansible_user`) and password (`ansible_password`). Due to the way the Vagrant box we are using was built, we are not using the default HTTPS scheme and are instead using the

HTTP scheme (`ansible_winrm_scheme`). This means that we have to use port 55985 (`ansible_port`) rather than port 99586. Both of these ports are mapped from our Ansible controller to ports 9585 and 5986 on the Windows host.

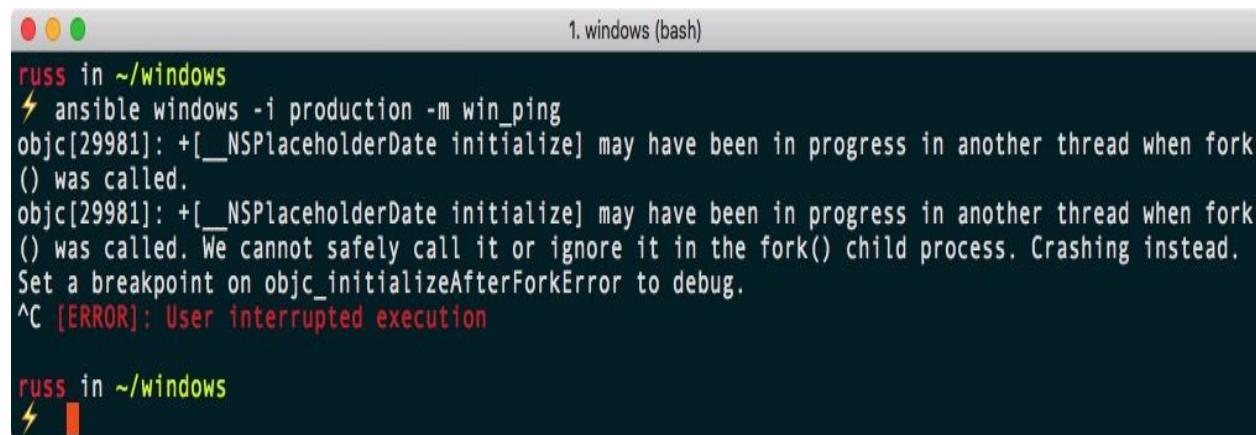
Now we have Windows up-and-running and Ansible configured, we can make a start on interacting with it.

The ping module

Not all Ansible modules work with Windows hosts and ping is one of them. There is a module provided for Windows called `win_ping` and we will be using that here.

The command we need to run follows; as you can see, other than the module name it is exactly the way we executed it against a Linux host: **\$ ansible windows -i production -m win_ping**

If you are a macOS user and you receive an error like this one, then don't worry; there is a workaround for it:



```
russ in ~/windows
⚡ ansible windows -i production -m win_ping
objc[29981]: +[__NSPlaceholderDate initialize] may have been in progress in another thread when fork()
() was called.
objc[29981]: +[__NSPlaceholderDate initialize] may have been in progress in another thread when fork()
() was called. We cannot safely call it or ignore it in the fork() child process. Crashing instead.
Set a breakpoint on objc_initializeAfterForkError to debug.
^C [ERROR]: User interrupted execution

russ in ~/windows
⚡ |
```

The error is a known issue that is being worked on by the Ansible team. In the meantime, run the following command, or add it to your `~/.bash_profile` file: **\$ export OBJC_DISABLE_INITIALIZE_FORK_SAFETY=YES**

Once you have run the command, you should see the following results:



1. windows (bash)

```
russ in ~/windows
⚡ ansible windows -i production -m win_ping
box1 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
russ in ~/windows
⚡
```

The next module we are going to run has been designed to work with Windows or Linux hosts.

The setup module

As we found out in [Chapter 2, *Installing and Running Ansible*](#), the setup module gathers facts on our target host; if we call the module directly using the `ansible` command, the facts are printed directly on the screen. To call the module, we need to run the following command:

```
| $ ansible windows -i production -m setup
```

As you can see from the following screen, the information displayed is pretty much identical to when we ran the module against a Linux host:

```
1. windows (bash)
russ in ~/windows
⚡ ansible windows -i production -m setup
box1 | SUCCESS => {
    "ansible_facts": {
        "ansible_architecture": "64-bit",
        "ansible_bios_date": "05/18/2017",
        "ansible_bios_version": "6.00",
        "ansible_date_time": {
            "date": "2018-05-07",
            "day": "07",
            "epoch": "1525680058.96767",
            "hour": "08",
            "iso8601": "2018-05-07T15:00:58Z",
            "iso8601_basic": "20180507T080058951799",
            "iso8601_basic_short": "20180507T080058",
            "iso8601_micro": "2018-05-07T15:00:58.951799Z",
            "minute": "00",
            "month": "05",
            "second": "58",
            "time": "08:00:58",
            "tz": "Pacific Standard Time",
            "tz_offset": "-07:00",
            "weekday": "Monday",
            "weekday_number": "1",
            "weeknumber": "18",
            "year": "2018"
        },
        "ansible_distribution": "Microsoft Windows Server 2016 Standard Evaluation",
        "ansible_distribution_major_version": "10",
        "ansible_distribution_version": "10.0.14393.0",
        "ansible_domain": "",
        "ansible_env": {
            "ALLUSERSPROFILE": "C:\\\\ProgramData"
        }
    }
}
```

We can use one of the playbooks from [Chapter 2](#), *Installing and Running Ansible*, to see this. In `playbook01.yml`, we used the facts gathered by Ansible when it first connects to the hosts to display a message. Let's update that playbook to interact with our Windows host:

```
---
- hosts: windows
  gather_facts: true

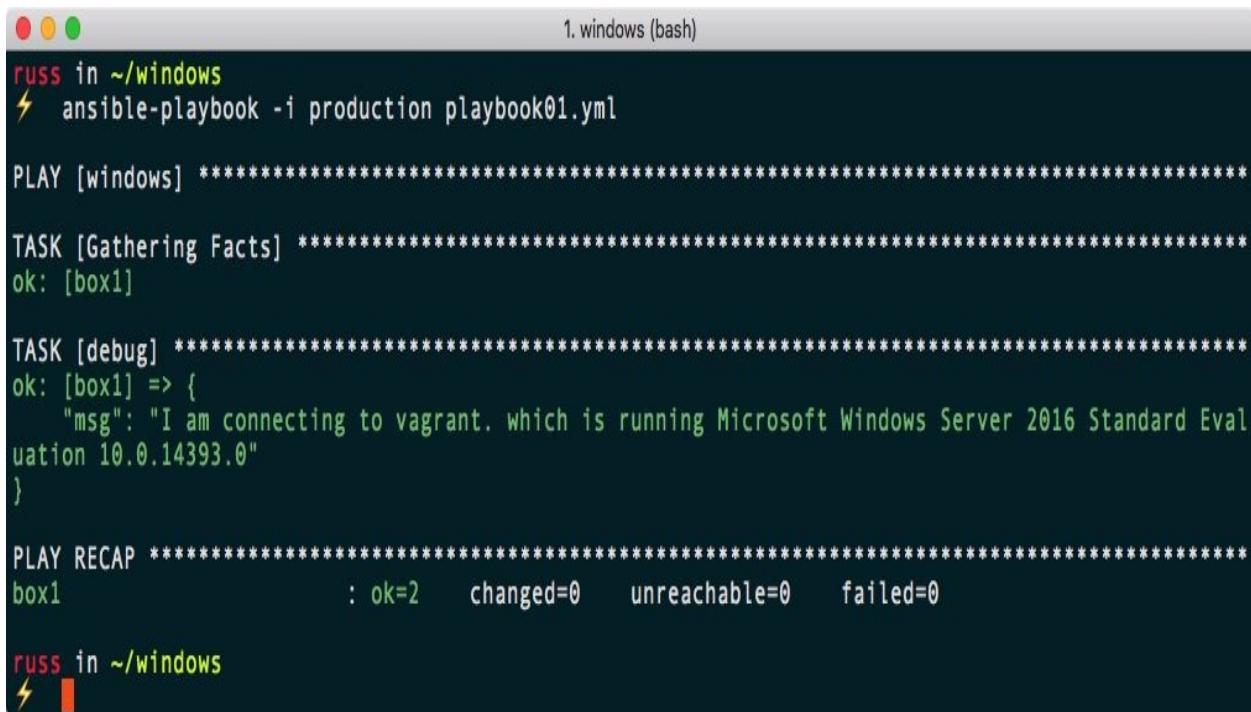
  tasks:
    - debug:
        msg: "I am connecting to {{ ansible_nodename }} which is running {{ ansible_distribution }} {{ ansible_distribution_version }}"
```

As you can see, we have updated the host group to use `windows` rather than `boxes`, and we also removed the `become` and `become_method` options as the user we will be connecting with has enough permissions to run the tasks we need.

We can run the playbook using the following command:

```
| $ ansible-playbook -i production playbook01.yml
```

The following screen gives the expected output:



A screenshot of a terminal window titled "1. windows (bash)". The terminal shows the command `ansible-playbook -i production playbook01.yml` being run. The output indicates a successful run on a Windows host named "box1". The "debug" task outputs a message about connecting to a Vagrant box running Microsoft Windows Server 2016 Standard Evaluation 10.0.14393.0. The play recap shows 2 tasks completed successfully (ok=2) with 0 changes, 0 unreachable hosts, and 0 failed tasks.

```
russ in ~/windows
⚡ ansible-playbook -i production playbook01.yml

PLAY [windows] ****
TASK [Gathering Facts] ****
ok: [box1]

TASK [debug] ****
ok: [box1] => {
    "msg": "I am connecting to vagrant, which is running Microsoft Windows Server 2016 Standard Evaluation 10.0.14393.0"
}

PLAY RECAP ****
box1 : ok=2    changed=0    unreachable=0    failed=0

russ in ~/windows
⚡
```

Now we have quickly covered the basics, we can look at doing something useful and install a few different software packages.

Installing a web server

One of the first things we did when we had our Linux host up-and-running was to install a web server, so let's look at repeating that process by installing and enabling **Internet Information Services (IIS)** on our Windows host.



IIS is the default web server that ships with Windows Server, and it supports the following protocols: HTTP, HTTPS, and HTTP/2, as well as FTP, FTPS, SMTP, and NNTP. It was first released 22 years ago as part of Windows NT.

Like all of the playbooks we have covered so far, let's create the basic skeleton by running the following commands:

```
$ mkdir web web/group_vars web/roles  
$ touch web/production web/site.yml web/group_vars/common.yml
```

Now we can make a start on writing our playbook.

IIS role

The first role we are going to look at installs and configures IIS and then, like our previous playbook, uploads an HTML file that is generated by Ansible using a template. First of all, change to the `web` folder and create the role by running:

```
$ cd web  
$ ansible-galaxy init roles/iis
```

Starting with the default variable in `roles/iis/defaults/main.yml`, we can see that our role is going to be really similar to our Apache role we created when setting up the LAMP stack:

```
---  
# defaults file for web/roles/iis  
  
document_root: 'C:\inetpub\wwwroot\'  
html_file: ansible.html  
  
html_heading: "Success !!"  
html_body: |  
    This HTML page has been deployed using Ansible to a <b>{{ ansible_distribution }}</b>  
host.<br><br>  
    The webroot is <b>{{ document_root }}</b> this file is called <b>{{ html_file }}</b>.  
<br>
```

As you can see, we are providing the path to the document root, a name for our HTML file, and also some content for our HTML file, for which the template can be found at `roles/iis/templates/index.html.j2`:

```
<!--{{ ansible_managed }}-->  
<!doctype html>  
<title>{{ html_heading }}</title>  
<style>  
    body { text-align: center; padding: 150px; }  
    h1 { font-size: 50px; }  
    body { font: 20px Helvetica, sans-serif; color: #333; }  
    article { display: block; text-align: left; width: 650px; margin: 0 auto; }  
</style>  
<article>  
    <h1>{{ html_heading }}</h1>  
    <div>  
        <p>{{ html_body }}</p>  
    </div>  
</article>
```

This is the exact template we used in our Apache role earlier on. Deploying IIS is quite simple, we just need two tasks in `roles/iis/tasks/main.yml` to accomplish

this. Our first task can be found here:

```
- name: enable IIS
  win_feature:
    name:
      - "Web-Server"
      - "Web-Common-Http"
    state: "present"
```

This uses the `win_feature` module to enable and start the `Web-Server` and `Web-Common-Http` features. The next and final task deploys our HTML page using the `win_template` module:

```
- name: create an html file from a template
  win_template:
    src: "index.html.j2"
    dest: "{{ document_root }}{{ html_file }}"
```

As you can see, the syntax is pretty much the same as the standard `template` module. Now that we have our role completed, we can run the playbook, copy the content of the host inventory file, which we used in the previous section, into the `production` file, and update `site.yml` so it contains the following content:

```
---
- hosts: windows
  gather_facts: true

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/iis
```

You can then run the playbook using the following command:

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

The output of the playbook run should look something like the following Terminal output:

```
1. web (bash)

russ in ~/windows/web
⚡ ansible-playbook -i production site.yml

PLAY [windows] ****
TASK [Gathering Facts] ****
ok: [box1]

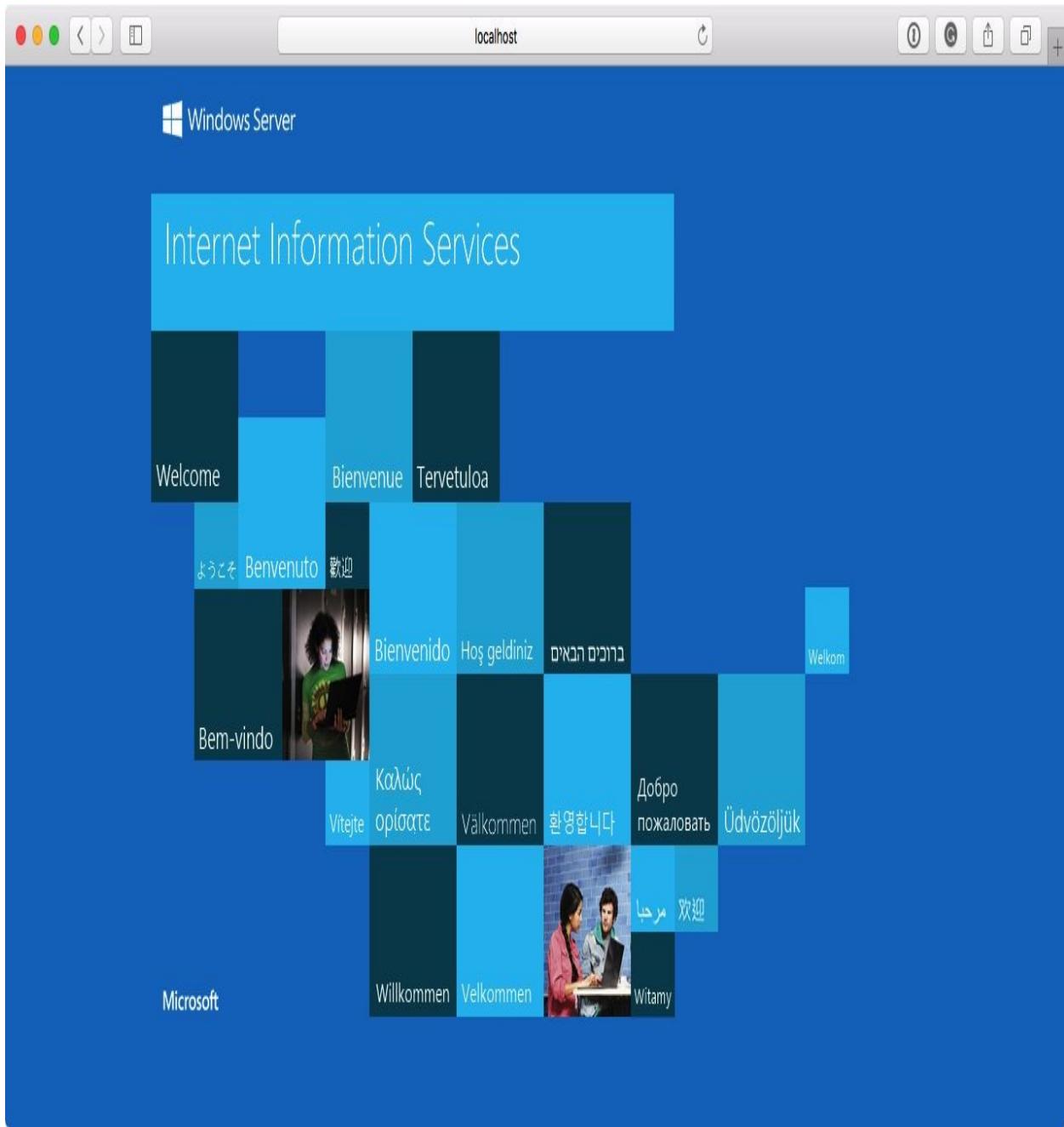
TASK [roles/iis : enable IIS] ****
changed: [box1]

TASK [roles/iis : create an html file from a template] ****
changed: [box1]

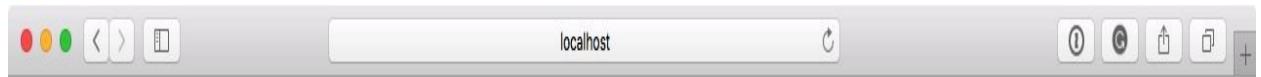
PLAY RECAP ****
box1 : ok=3    changed=2    unreachable=0    failed=0

russ in ~/windows/web
⚡
```

Once complete, you should be able to open the web browser on your local machine and go to `http://localhost:8080/`, which should display the default IIS page:



Opening `http://localhost:8080/ansible.html` will show the page we uploaded:



Success !!!

This HTML page has been deployed using Ansible to a **Microsoft Windows Server 2016 Standard Evaluation host**.

The webroot is **C:\inetpub\wwwroot**, this file is called **ansible.html**.

ASP.NET role

Now we have IIS up-and-running, let's look at enabling ASP.NET support. Again, let's start by creating the role:

```
| $ ansible-galaxy init roles/asp
```

Starting with the variables in `roles/asp/defaults/main.yml`, you can see that they look similar to the HTML ones except that we have prefixed them with `.aspx` so that they do not clash with the variables from the `iis` role:

```
aspx_document_root: 'C:\inetpub\wwwroot\ansible\'  
aspx_file: default.aspx  
  
aspx_heading: "Success !!!"  
aspx_body: |  
    This HTML page has been deployed using Ansible to a <b>{{ ansible_distribution }}</b>  
host.<br><br>  
    The webroot is <b>{{ aspx_document_root }}</b> this file is called <b>{{ aspx_file }}</b>.  
<br><br>  
    The output below is from ASP.NET<br><br>  
    Hello from <%= Environment.MachineName %> at <%= DateTime.UtcNow %><br><br>
```

As you can see from the bottom of the page, we are including a function that prints the machine name, which in our case should be Vagrant, and also the date and time.

Next up, we have the template in `roles/asp/templates/default.aspx.j2`. Apart from the updated variables and filename, the content remains more or less the same as the one used in the `iis` role:

```
<!--{{ ansible_managed }}-->  
<!doctype html>  
<title>{{ html_heading }}</title>  
<style>  
    body { text-align: center; padding: 150px; }  
    h1 { font-size: 50px; }  
    body { font: 20px Helvetica, sans-serif; color: #333; }  
    article { display: block; text-align: left; width: 650px; margin: 0 auto; }  
</style>  
<article>  
    <h1>{{ aspx_heading }}</h1>  
    <div>  
        <p>{{ aspx_body }}</p>  
    </div>  
</article>
```

Next, we have the tasks that should be placed in `roles/asp/tasks/main.yml`. First of all, we are using the `win_feature` module to enable the components needed to get our basic page up-and-running:

```
- name: enable .net
  win_feature:
    name:
      - "Net-Framework-Features"
      - "Web-Asp-Net45"
      - "Web-Net-Ext45"
    state: "present"
  notify: restart iis
```

Next, we need to create a folder to serve our page from and copy the rendered template:

```
- name: create the folder for our asp.net app
  win_file:
    path: "{{ aspx_document_root }}"
    state: "directory"

- name: create an aspx file from a template
  win_template:
    src: "default.aspx.j2"
    dest: "{{ aspx_document_root }}{{ aspx_file }}"
```

As you can see, we are using the `win_template` module again. As well as using the `win_file` module, the syntax for the file module is extremely close to that of the `file` module we have been using in other chapters. The final task checks that the site is configured correctly in IIS:

```
- name: ensure the default web application exists
  win_iis_webapplication:
    name: "Default"
    state: "present"
    physical_path: "{{ aspx_document_root }}"
    application_pool: "DefaultAppPool"
    site: "Default Web Site"
```

The `win_iis_webapplication` module is used for, as the name says, configuring web applications within IIS. This is not strictly required for our example, but it gives you an idea of what is possible.

You may have noticed that, when we enabled the additional features, we sent a notification to restart IIS. This means we have to add a task to the `roles/asp/handlers/main.yml` file. This task uses the `win_service` module to restart the webserver:

```
| - name: restart iis
```

```
|   win_service:  
|     name: w3svc  
|     state: restarted
```

Now that we have our completed role, we can look at running the playbook again. First, we need to add the new role to the `site.yml` file:

```
---
```

```
- hosts: windows  
  gather_facts: true
```

```
  vars_files:  
    - group_vars/common.yml
```

```
  roles:  
    - roles/iis  
    - roles/asp
```

Then, we can run the playbook using the following command:

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

This should give you something along the lines of the following output:

```
● ● ● 1. web (bash)
⚡ ansible-playbook -i production site.yml

PLAY [windows] ****
TASK [Gathering Facts] ****
ok: [box1]

TASK [roles/iis : enable IIS] ****
ok: [box1]

TASK [roles/iis : create an html file from a template] ****
ok: [box1]

TASK [roles/asp : enable .net] ****
changed: [box1]

TASK [roles/asp : create the folder for our asp.net app] ****
changed: [box1]

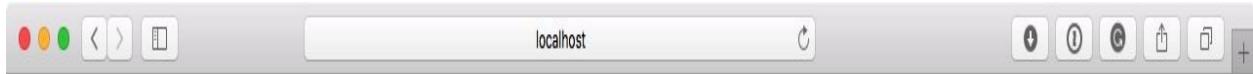
TASK [roles/asp : create an aspx file from a template] ****
changed: [box1]

TASK [roles/asp : ensure the default web application exists] ****
changed: [box1]

RUNNING HANDLER [roles/asp : restart iis] ****
changed: [box1]

PLAY RECAP ****
box1 : ok=8    changed=5    unreachable=0    failed=0
```

Opening your browser and going to `http://localhost:8080/ansible/` should present you with something that looks similar to the following web page:



This HTML page has been deployed using Ansible to a **Microsoft Windows Server 2016 Standard Evaluation** host.

The webroot is **C:\inetpub\wwwroot\ansible** this file is called **default.aspx**.

The output below is from ASP.NET

Hello from VAGRANT at 5/11/2018 12:19:14 PM

Let's remove our Vagrant box and look at a few more modules. To remove the box, run:

```
| $ vagrant destroy
```

We can now look at creating a user using Ansible, and also install a few desktop applications using Chocolatey on a server host in AWS.

Interacting with AWS Windows instances

When we interacted with our local Windows Vagrant box, it was not using a secure connection; let's look at launching a Windows EC2 instance in an AWS instance and then interacting with it like we did with the CentOS 7 instance in [Chapter 10, Highly Available Cloud Deployments](#).

First of all, we need to create the folder structure for our new playbook:

```
$ mkdir cloud cloud/group_vars cloud/roles  
$ touch cloud/production cloud/site.yml cloud/group_vars/common.yml
```

Once we have the structure, we need to create four roles, starting with the AWS one.

AWS role

Our first role will create the VPC and launch the EC2 instance. To bootstrap the role change, go to the cloud folder and run:

```
$ cd cloud  
$ ansible-galaxy init roles/aws
```

Let's start with the contents of `roles/aws/default/main.yml` first:

```
vpc_cidr_block: "10.0.0.0/16"  
the_subnets:  
  - { use: 'ec2', az: 'a', subnet: '10.0.10.0/24' }  
  
ec2:  
  instance_type: "t2.large"  
  wait_port: "5986"  
  
image:  
  base: Windows_Server-2016-English-Full-Base-*  
  owner: amazon  
  architecture: x86_64  
  root_device: ebs  
  
win_initial_password: "{{ lookup('password', 'group_vars/generated_administrator  
chars=ascii_letters,digits length=30') }}"
```

As you can see, we are only going to be using a single subnet and we are going to be looking for a Windows Server 2016 AMI during the playbook run. Finally, we are setting a variable called `win_initial_password`, which will be used to set our Administrator password later in the playbook run.

Most of the tasks in `roles/aws/tasks/main.yml` are as you would expect. First, we set up the VPC, create the subnet, and find out your current IP addresses for use with the security group:

```
- name: ensure that the VPC is present  
  ec2_vpc_net:  
    region: "{{ ec2_region }}"  
    name: "{{ environment_name }}"  
    state: present  
    cidr_block: "{{ vpc_cidr_block }}"  
    resource_tags: { "Name" : "{{ environment_name }}", "Environment" : "{{  
environment_name }}"}  
    register: vpc_info  
  
- name: ensure that the subnets are present  
  ec2_vpc_subnet:  
    region: "{{ ec2_region }}"
```

```

state: present
vpc_id: "{{ vpc_info.vpc.id }}"
cidr: "{{ item.subnet }}"
az: "{{ ec2_region }}{{ item.az }}"
resource_tags:
    "Name" : "{{ environment_name }}_{{ item.use }}_{{ ec2_region }}{{ item.az }}"
    "Environment" : "{{ environment_name }}"
    "Use" : "{{ item.use }}"
with_items: "{{ the_subnets }}"

```

- name: gather information about the ec2 subnets
 ec2_vpc_subnet_facts:
 region: "{{ ec2_region }}"
 filters:
 "tag:Use": "ec2"
 "tag:Environment": "{{ environment_name }}"
 register: subnets_ec2
- name: register just the IDs for each of the subnets
 set_fact:
 subnet_ec2_ids: "{{ subnets_ec2.subnets | map(attribute='id') | list }}"
- name: find out your current public IP address using https://ipify.org/
 ipify_facts:
 register: public_ip
- name: set your public ip as a fact
 set_fact:
 your_public_ip: "{{ public_ip.ansible_facts.ipify_public_ip }}/32"

The security group has been updated so, rather than port 22, we are opening ports for remote desktop (port 3389) and WinRM (ports 5985 and 5986):

```

- name: provision ec2 security group
  ec2_group:
    region: "{{ ec2_region }}"
    vpc_id: "{{ vpc_info.vpc.id }}"
    name: "{{ environment_name }}-ec2"
    description: "Opens the RDP and WinRM ports to a trusted IP"
    tags:
      "Name": "{{ environment_name }}-ec2"
      "Environment": "{{ environment_name }}"
    rules:
      - proto: "tcp"
        from_port: "3389"
        to_port: "3389"
        cidr_ip: "{{ your_public_ip }}"
        rule_desc: "allow {{ your_public_ip }} access to port RDP"
      - proto: "tcp"
        from_port: "5985"
        to_port: "5985"
        cidr_ip: "{{ your_public_ip }}"
        rule_desc: "allow {{ your_public_ip }} access to WinRM"
      - proto: "tcp"
        from_port: "5986"
        to_port: "5986"
        cidr_ip: "{{ your_public_ip }}"
        rule_desc: "allow {{ your_public_ip }} access to WinRM"
  register: sg_ec2

```

We then continue to build out our network by adding an internet gateway and routing before finding the right AMI ID to use:

```
- name: ensure that there is an internet gateway
  ec2_vpc_igw:
    region: "{{ ec2_region }}"
    vpc_id: "{{ vpc_info.vpc.id }}"
    state: present
    tags:
      "Name": "{{ environment_name }}_internet_gateway"
      "Environment": "{{ environment_name }}"
      "Use": "gateway"
    register: igw_info

- name: check that we can route through internet gateway
  ec2_vpc_route_table:
    region: "{{ ec2_region }}"
    vpc_id: "{{ vpc_info.vpc.id }}"
    subnets: "{{ subnet_ec2_ids }}"
    routes:
      - dest: 0.0.0.0/0
        gateway_id: "{{ igw_info.gateway_id }}"
  resource_tags:
    "Name": "{{ environment_name }}_outbound"
    "Environment": "{{ environment_name }}"

- name: search for all of the AMIs in the defined region which match our selection
  ec2_ami_facts:
    region: "{{ ec2_region }}"
    owners: "{{ image.owner }}"
    filters:
      name: "{{ image.base }}"
      architecture: "{{ image.architecture }}"
      root-device-type: "{{ image.root_device }}"
  register: amiFind

- name: filter the list of AMIs to find the latest one with an EBS backed volume
  set_fact:
    amiSortFilter: "{{ amiFind.images | sort(attribute='creation_date') | last }}"

- name: finally grab AMI ID of the most recent result which matches our base image
  which is backed by an EBS volume
  set_fact:
    our_ami_id: "{{ amiSortFilter.image_id }}"
```

Now it is time to launch the EC2 instance; you may have noticed that we have not had to upload a key or any credentials. This is because we will actually inject a PowerShell script that executes when the instance is first booted. This script will set the administrator password and configure the instance so that Ansible can be run against it:

```
- name: launch an instance
  ec2_instance:
    region: "{{ ec2_region }}"
    state: "present"
    instance_type: "{{ ec2.instance_type }}"
    image_id: "{{ our_ami_id }}"
    wait: yes
```

```

    security_groups: [ "{{ sg_ec2.group_id }}" ]
network:
  assign_public_ip: true
filters:
  instance-state-name: "running"
  "tag:Name": "{{ environment_name }}"
  "tag:environment": "{{ environment_name }}"
vpc_subnet_id: "{{ subnet_ec2_ids[0] }}"
user_data: "{{ lookup('template', 'userdata.j2') }}"
tags:
  Name: "{{ environment_name }}"
  environment: "{{ environment_name }}"

```

The script is a template called `userdata.j2`, which is injected into the instance at boot using the `user_data` key. We will take a look at the template in a moment; all that remains in this role is to add the instance to a host group and then wait for WinRM to be accessible:

```

- name: gather facts on the instance we just launched using the AWS API
  ec2_instance_facts:
    region: "{{ ec2_region }}"
    filters:
      instance-state-name: "running"
      "tag:Name": "{{ environment_name }}"
      "tag:environment": "{{ environment_name }}"
  register: singleinstance

- name: add our temporary instance to a host group for use in the next step
  add_host:
    name: "{{ item.public_dns_name }}"
    ansible_ssh_host: "{{ item.public_dns_name }}"
    groups: "ec2_instance"
    with_items: "{{ singleinstance.instances }}"

- name: wait until WinRM is available before moving onto the next step
  wait_for:
    host: "{{ item.public_dns_name }}"
    port: "{{ ec2.wait_port }}"
    delay: 2
    timeout: 320
    state: "started"
  with_items: "{{ singleinstance.instances }}"

```

The `userdata.j2` template in `roles/aws/templates/` looks like the following:

```

<powershell>
$admin = [adsi]("WinNT://./administrator, user")
$admin.PSBase.Invoke("SetPassword", "{{ win_initial_password }})
Invoke-Expression ((New-Object
System.Net.Webclient).DownloadString('https://raw.githubusercontent.com/ansible/ansible/
</powershell>

```

The first part of the script sets the password (`win_initial_password`) for the administrator user; the script then downloads and executes a PowerShell script directly from Ansible's GitHub repository. This script runs checks against the

current WinRM configuration on the target instance and then makes the changes needed for Ansible to be able to securely connect. The script also configures all actions in WinRM to be logged to the instances event log.

User role

Next up, we have the user role, which we can run the following command to create:

```
| $ ansible-galaxy init roles/user
```

This role creates a user for us to connect to our instance with. The defaults that can be found in `roles/user/defaults/main.yml` are as follows:

```
ansible:  
  username: "ansible"  
  password: "{{ lookup('password', 'group_vars/generated_ansible  
chars=ascii_letters,digits length=30') }}"  
  groups:  
    - "Users"  
    - "Administrators"
```

As you can see, here we are defining a user called `ansible` that has a 30-character random password. The `ansible` user will be a member of the `Users` and `Administrators` groups. There is a single task in `roles/user/tasks/main.yml` using the `win_user` module, which looks like:

```
- name: ensure that the ansible created users are present  
  win_user:  
    name: "{{ ansible.username }}"  
    fullname: "{{ ansible.username | capitalize }}"  
    password: "{{ ansible.password }}"  
    state: "present"  
    groups: "{{ ansible.groups }}"
```

Like all Windows modules, the syntax is similar to the Linux equivalent so you should have a good idea of what each key means. As you can see from the previous task, we are using a Jinja2 transformation to capitalize the first letter of the `ansible.username` variable.

Chocolatey role

The next role uses Chocolatey to install a few bits of software on the machine.



Chocolatey is a package manager for Windows, similar in principle and functionality to Homebrew, which we used in earlier chapters to install the software we needed on macOS with a single command. Chocolatey simplifies the installation of packages on the command line by wrapping the installation process for most common Windows installers into a common set of PowerShell commands—perfect for an orchestration tool such as Ansible.

To add the files needed for the role, run the following command:

```
| $ ansible-galaxy init roles/choc
```

In `roles/choc/defaults/main.yml`, we have a list of the packages we want to install:

```
apps:
  - "notepadplusplus.install"
  - "putty.install"
  - "googlechrome"
```

As you can see, we want to install Notepad++, PuTTY, and Google Chrome. The task itself, which needs to be added to `roles/choc/tasks/main.yml`, looks like the following:

```
- name: install software using chocolatey
  win_chocolatey:
    name: "{{ item }}"
    state: "present"
  with_items: "{{ apps }}
```

Again, the `win_chocolatey` module takes a similar input to the package manager modules we have used in previous chapters when targeting a Linux-based host.

Information role

The final role we are creating is called `info`, and its only purpose is to output information on our newly launched and configured Windows Server 2016 EC2 instance. As you may have already guessed, we need to run the following command: `$ ansible-galaxy init roles/info`

Once we have the files, add the following task to `roles/info/tasks/main.yml`:

```
- name: print out information on the host
  debug:
    msg: "You can connect to '{{ inventory_hostname }}' using the username of '{{ ansible.username }}' with a password of '{{ ansible.password }}'."
```

As you can see, this will provide us with the host to connect to, along with the username and password.

Running the playbook

Before we run the playbook, we need to add the following to `group_vars/common.yml`:

```
| environment_name: "windows_example"
| ec2_region: "eu-west-1"
```

The host inventory file called `production` should contain the following:

```
[ec2_instance]

[ec2_instance:vars]
ansible_connection=winrm
ansible_user="Administrator"
ansible_password="{{ lookup('password', 'group_vars/generated_administrator
chars=ascii_letters,digits_length=30') }}"
ansible_winrm_server_cert_validation=ignore
```

As you can see, we are using the WinRM connector to connect to our Windows instance using the administrator username and the password we set when running the user data script when launching the instance. The `site.yml` file should have the following content:

```
---
- name: Create the AWS environment and launch an EC2 instance
  hosts: localhost
  connection: local
  gather_facts: True

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/aws

- name: Bootstrap the EC2 instance
  hosts: ec2_instance
  gather_facts: true

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/user
    - roles/choc
    - roles/info
```

We can run the playbook using the following command, after first exporting our AWS credentials:

```
$ export AWS_ACCESS_KEY=AKIAI5KECPOTNTTVM3EDA
$ export AWS_SECRET_KEY=Y4B7FFiSwl0Am3VIFc07lgnc/TAtK5+RpxzIGTr
$ ansible-playbook -i production site.yml
```

A slightly edited output of the playbook run follows:

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the
implicit
localhost does not match 'all'

PLAY [Create the AWS environment and launch an EC2 instance]
*****
TASK [Gathering Facts]
*****
ok: [localhost]

TASK [roles/aws : ensure that the VPC is present]
*****
changed: [localhost]

TASK [roles/aws : ensure that the subnets are present]
*****
changed: [localhost] => (item={u'subnet': u'10.0.10.0/24', u'use': u'ec2', u'az': u'a'})

TASK [roles/aws : gather information about the ec2 subnets]
*****
ok: [localhost]

TASK [roles/aws : register just the IDs for each of the subnets]
*****
ok: [localhost]

TASK [roles/aws : find out your current public IP address using https://ipify.org/]
*****
ok: [localhost]

TASK [roles/aws : set your public ip as a fact]
*****
ok: [localhost]

TASK [roles/aws : provision ec2 security group]
*****
changed: [localhost]

TASK [roles/aws : ensure that there is an internet gateway]
*****
changed: [localhost]

TASK [roles/aws : check that we can route through internet gateway]
*****
changed: [localhost]

TASK [roles/aws : search for all of the AMIs in the defined region which match our
selection] ***
ok: [localhost]

TASK [roles/aws : filter the list of AMIs to find the latest one with an EBS backed
volume] ****
ok: [localhost]
```

```

TASK [roles/aws : finally grab AMI ID of the most recent result which matches our base
image which is backed by an EBS volume]
*****
ok: [localhost]

TASK [roles/aws : launch an instance]
*****
changed: [localhost]

TASK [roles/aws : gather facts on the instance we just launched using the AWS API]
*****
ok: [localhost]

TASK [roles/aws : add our temporary instance to a host group for use in the next step]
*****
changed: [localhost] =>

TASK [roles/aws : wait until WinRM is available before moving onto the next step]
*****
ok: [localhost] =>

PLAY [Bootstrap the EC2 instance]
*****

TASK [Gathering Facts]
*****
ok: [ec2-34-245-2-119.eu-west-1.compute.amazonaws.com]

TASK [roles/user : ensure that the ansible created users are present]
*****
changed: [ec2-34-245-2-119.eu-west-1.compute.amazonaws.com]

TASK [roles/choc : install software using chocolatey]
*****
changed: [ec2-34-245-2-119.eu-west-1.compute.amazonaws.com] =>
(item=notepadplusplus.install)
changed: [ec2-34-245-2-119.eu-west-1.compute.amazonaws.com] => (item=putty.install)
changed: [ec2-34-245-2-119.eu-west-1.compute.amazonaws.com] => (item=googlechrome)
[WARNING]: Chocolatey was missing from this system, so it was installed during this
task run.

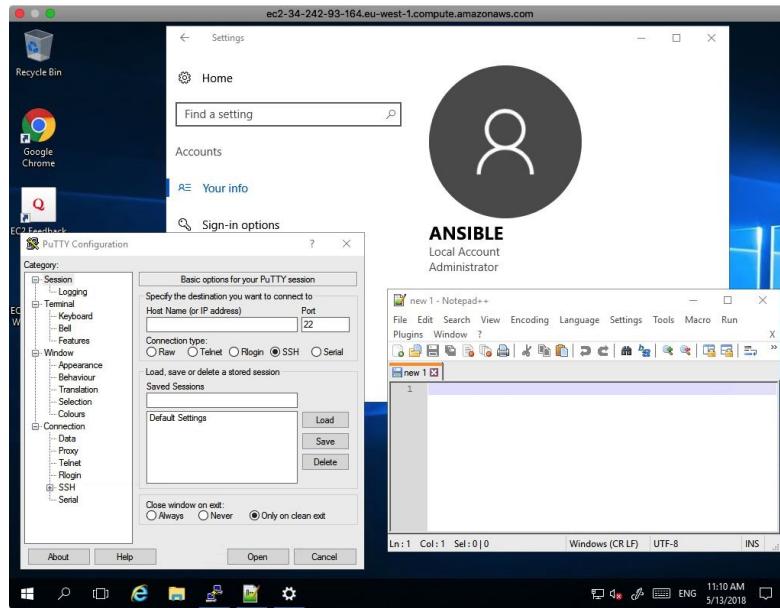
TASK [roles/info : print out information on the host]
*****
ok: [ec2-34-245-2-119.eu-west-1.compute.amazonaws.com] => {
  "msg": "You can connect to 'ec2-34-245-2-119.eu-west-1.compute.amazonaws.com' using
the username of 'ansible' with a password of 'Qb9LVPkUeZFRx5HLFgV1lFrkqK7HHN'."
}

PLAY RECAP
*****
ec2-34-245-2-119.eu-west-1.compute.amazonaws.com : ok=4 changed=2 unreachable=0
failed=0
localhost : ok=17 changed=7 unreachable=0 failed=0

```

As you can see from the output, my EC2 instance has a hostname of `ec2-34-245-2-119.eu-west-1.compute.amazonaws.com` and the `ansible` user has a password of `Qb9LVPkUeZFRx5HLFgV1lFrkqK7HHN`. I can connect to the instance using Microsoft RDP with those details (remember it is locked down to your IP address). As you can

see from the following screenshot, I am connected as the Ansible user and have PuTTY and Notepad ++ open; you can also make out the shortcut to Google Chrome on the desktop:



The other thing you may have noticed is that we never had to install Chocolatey. As stated during the playbook run, if `win_chocolatey` doesn't find a Chocolatey installation on the target machine it will install and configure it automatically.

There is a playbook in the `chapter12/cloud` folder in the GitHub repository that removes the resources we created here. To run this, use the following command:

```
| $ ansible-playbook -i production remove.yml
```

Make sure you double-check that everything has been removed as expected to ensure you do not get any unexpected bills.

Summary

As mentioned at the start of the chapter, using a traditional Linux tool such as Ansible on Windows always feels a little strange. However, I am sure you will agree that the experience is as Linux-like as possible. When I first started experimenting with the Windows modules, I was surprised that I managed to launch an EC2 Windows Server instance and managed to deploy a simple web application without ever having to remote-desktop into the target instance.

With each new release, Ansible gets more and more support for Windows-based hosts, making it easy to manage mixed workloads from your playbooks.

In the next chapter, we are going to return to more familiar territory, for me at least, and look at how we can harden our Linux installation.

Questions

1. Which of the following two modules can be used on both a Windows and Linux host, setup or file?
2. True or false: You can use SSH to access your Windows target.
3. Explain the type of interface WinRM uses.
4. Which Python module do you need to install to be able to interact with WinRM on macOS and Linux?
5. True or false: You can have a separate task to install Chocolatey before you use the `win_chocolatey` module.
6. Update the playbook to install additional packages.

Further reading

You can find more information on the excellent Chocolatey at <http://chocolatey.org/>.

Hardening Your Servers Using Ansible and OpenSCAP

One of the advantages of using an orchestration and configuration tool like Ansible is that it can be used to generate and deploy a complex set of configurations in a repeatable task across many hosts. In this chapter, we are going to a look at a tool that actually generates the configuration for you to then apply.

In this chapter, we will learn how to harden a Red Hat-based CentOS 7.5.1804 host using Ansible and OpenSCAP.

Technical requirements

We are going to be targeting a Vagrant box running CentOS Linux release 7.5.1804; we are using this box because it comes with the latest version of OpenSCAP. A copy of the final playbooks can be found in the repository that accompanies this book; the repository can be found at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter13/scap>.

OpenSCAP

We are going to be looking at one of a set of tools maintained by Red Hat called OpenSCAP. Before we continue, I feel I should warn you that the next section is going to contain a lot of abbreviations, starting with SCAP.

So, what is SCAP? The **Security Content Automation Protocol (SCAP)** is an open standard that encompasses several components, all of which are open standards themselves, to build a framework that allows you to automatically assess and remediate your hosts against the **National Institute of Standards and Technology (NIST) Special Publication 800-53**.

This publication is a catalog of controls that is applied to all U.S. federal IT systems, apart from those maintained by the **National Security Agency (NSA)**. These controls have been put in place to help implement the **Federal Information Security Management Act of 2002 (FISMA)** across U.S federal departments.

SCAP is made up of the following components:

- **Asset Identification (AID)** is a data model used for asset identification.
- **Asset Reporting Format (ARF)** is a vendor-neutral and technology agnostic data model for transporting information on assets between different reporting applications and services.
- **Common Configuration Enumeration (CCE)** is a standard database of recommended configuration for common software. Each recommendation has a unique identifier. At the time of writing, the database has not been updated since 2013.
- **Common Configuration Scoring System (CCSS)** is the continuation of CCE. It is used for generating a score for various software and hardware configurations across all types of deployments.
- **Common Platform Enumeration (CPE)** is a method of identifying hardware assets, operating systems, and software present in an organization's infrastructure. Once identified, this data can then be used to search other databases to threat assess the asset.
- **Common Weakness Enumeration (CWE)** is a common language for

dealing with and discussing the causes of weaknesses in system architecture, design, and code that may lead to vulnerabilities.

- **Common Vulnerabilities and Exposures (CVE)** is a database of publicly acknowledged vulnerabilities. Most system administrators and IT professionals will have come across the CVE database at some point. Each vulnerability receives a unique ID; for example, most people will know CVE-2014-0160, which is also known as **Heartbleed**.
- **Common Vulnerability Scoring System (CVSS)** is a method that helps capture the characteristics of a vulnerability to produce a normalized numerical score, which can then be used to describe the impact of a vulnerability, for example, low, medium, high, and critical.
- **Extensible Configuration Checklist Description Format (XCCDF)** is an XML format for describing security checklists. It can also be used for configuration and benchmarks and provides a common language for all the parts of SCAP.
- **Open Checklist Interactive Language (OCIL)** is a framework for expressing questions to an end user and also the procedures to process the responses in a standardized way.
- **Open Vulnerability and Assessment Language (OVAL)** is defined in XML and aims to standardize the transfer of security content across all of the tools and services offered by NIST, the MITRE Corporation, the **United States Computer Emergency Readiness Team (US-CERT)**, and the **United States Department of Homeland Security (DHS)**.
- **Trust Model for Security Automation Data (TMSAD)** is an XML document that aims to define a common trust model which can be applied to the data being exchanged by all of the components that make up SCAP.

As you can imagine, there have been thousands of man-years that have gone into producing SCAP and the components that go to make its foundation. Some of the projects have been around in one form or another since the mid 90s, so they are well-established and considered the de facto standard when it comes to security best practices; however, I am sure you are thinking that it all sounds very complicated—after all, these are standards that have been defined and are being maintained by scholars, security professionals, and government departments.

This is where OpenSCAP comes in. The OpenSCAP project, maintained by Red Hat and also certificated by NIST for its support of the SCAP 1.2 standard,

allows you to apply all of the best practices we have discussed using a command-line client.

OpenSCAP, like a lot of Red Hat projects, is gaining support for Ansible and the current release introduces support for automatically generating Ansible playbooks to remediate non-conformance discovered during an OpenSCAP scan.



The automatic remediation scripts in the current version of OpenSCAP are a work in progress and there are known issues, which we will address toward the end of the chapter. Because of this, your output may differ from that covered in this chapter.

In the sections that follow, we will launch a CentOS 7.5.1804 Vagrant box, scan it, and generate the remediation playbook. As playbook support has only just been introduced, there is not yet 100% coverage of the fixes, so we will then scan the host a second time and then, using Ansible, generate a remediation bash script, and execute it on our host, before executing another scan, so we can compare the results of all three scans.

Preparing the host

Before we start scanning, we need a host to target, so let's quickly create the folder structure and `Vagrantfile`. To create the structure, run the following commands:

```
$ mkdir scap scap/group_vars scap/roles  
$ touch scap/Vagrantfile scap/production scap/site.yml scap/group_vars/common.yml
```

The `Vagrantfile` we created should contain the following code:

```
# -*- mode: ruby -*-  
# vi: set ft=ruby :  
  
API_VERSION = "2"  
BOX_NAME = "russmckendrick/centos75"  
BOX_IP = "10.20.30.40"  
DOMAIN = "nip.io"  
PRIVATE_KEY = "~/.ssh/id_rsa"  
PUBLIC_KEY = '~/.ssh/id_rsa.pub'  
  
Vagrant.configure(API_VERSION) do |config|  
  config.vm.box = BOX_NAME  
  config.vm.network "private_network", ip: BOX_IP  
  config.vm.host_name = BOX_IP + '.' + DOMAIN  
  config.vm.synced_folder ".", "/vagrant", disabled: true  
  config.ssh.insert_key = false  
  config.ssh.private_key_path = [PRIVATE_KEY, "~/.vagrant.d/insecure_private_key"]  
  config.vm.provision "file", source: PUBLIC_KEY, destination: "~/.ssh/authorized_keys"  
  
  config.vm.provider "virtualbox" do |v|  
    v.memory = "2024"  
    v.cpus = "2"  
  end  
  
  config.vm.provider "vmware_fusion" do |v|  
    v.vmx["memsize"] = "2024"  
    v.vmx["numvcpus"] = "2"  
  end  
end
```

This means that the host inventory file, `scap/production`, should contain the following:

```
box1 ansible_host=10.20.30.40.nip.io  
  
[scap]  
box1  
  
[scap:vars]  
ansible_connection=ssh
```

```
| ansible_user=vagrant  
| ansible_private_key_file=~/ssh/id_rsa  
| host_key_checking=False
```

We can launch the Vagrant box with one of the following commands:

```
| $ vagrant up  
| $ vagrant up --provider=vmware_fusion
```

Now that we have our target host ready, we can perform our initial scan.

The playbook

We are going to split the playbook into a few different roles. Unlike previous chapters, we are going to make a few of the roles reusable and pass parameters to them as they are executed. Our first role is a simple one, which installs the packages we need to run our OpenSCAP scan.

Install role

As mentioned previously, this first role is a simple one that installs the packages we need to run a scan:

```
| $ ansible-galaxy init roles/install
```

There are a few defaults we need to set in `roles/install/default/main.yml`; these are:

```
install:
  packages:
    - "openscap-scanner"
    - "scap-security-guide"
```

There is a task in `roles/install/tasks/main.yml` that installs the packages and also performs a `yum` update:

```
- name: update all of the installed packages
  yum:
    name: "*"
    state: "latest"
    update_cache: "yes"

- name: install the packages needed
  package:
    name: "{{ item }}"
    state: latest
  with_items: "{{ install.packages }}"
```

That is it for this role; we will be calling it each time we run a scan to ensure that we have the correct packages installed to be running the scan itself.

Scan role

Now that we have the OpenSCAP packages installed, we can create a role that performs the scan:

```
| $ ansible-galaxy init roles/scan
```

As already mentioned, we will be reusing this role throughout the playbook, which presents us with an easily resolved problem. By default, even if you define the role several times, Ansible will only execute a role once during a playbook run. To allow the role to execute more than once, we need to add the following line to the top of the `roles/scan/meta/main.yml` file:

```
| allow_duplicates: true
```

This instructs Ansible to execute this one role multiple times during the playbook run. Next up, we need to add some variables to the `group_vars/common.yml` file. These key values will be shared across all of the roles we will be using in our playbook. The first set of nested variables looks like the following:

```
oscap:  
  profile: "xccdf_org.ssgproject.content_profile_pci-dss"  
  policy: "ssg-centos7-ds.xml"  
  policy_path: "/usr/share/xml/scap/ssg/content/"
```

These define which profile we want to use and also which policy we want to apply. By default, OpenSCAP doesn't ship with any policies; these were installed with the `scap-security-guide` package. This package provides several policies, which can all be found in `/usr/share/xml/scap/ssg/content/`; the following terminal screenshot shows a directory listing of that folder:

```
vagrant@10:~ (ssh)
[vagrant@10 ~]$ ls /usr/share/xml/scap/ssg/content/
ssg-centos6-ds.xml      ssg-jre-cpe-dictionary.xml    ssg-rhel6-oval.xml
ssg-centos6-xccdf.xml   ssg-jre-cpe-oval.xml       ssg-rhel6-xccdf.xml
ssg-centos7-ds.xml      ssg-jre-ds.xml            ssg-rhel7-cpe-dictionary.xml
ssg-centos7-xccdf.xml   ssg-jre-ocil.xml          ssg-rhel7-cpe-oval.xml
ssg-firefox-cpe-dictionary.xml ssg-jre-oval.xml        ssg-rhel7-ds.xml
ssg-firefox-cpe-oval.xml ssg-jre-xccdf.xml         ssg-rhel7-ocil.xml
ssg-firefox-ds.xml       ssg-rhel6-cpe-dictionary.xml ssg-rhel7-oval.xml
ssg-firefox-ocil.xml     ssg-rhel6-cpe-oval.xml      ssg-rhel7-xccdf.xml
ssg-firefox-oval.xml    ssg-rhel6-ds.xml
ssg-firefox-xccdf.xml   ssg-rhel6-ocil.xml
[vagrant@10 ~]$
```

For our playbook, we are going to be using the `ssg-centos7-ds.xml` policy, or to give it its proper title, `PCI-DSS v3 Control Baseline for CentOS Linux 7`.



The Payment Card Industry Data Security Standard (PCI-DSS) is a standard that's been agreed upon by all of the major credit card operators that anyone handling cardholder data must adhere to. The standard is a set of security controls that are audited either by an external auditor or via a self-assessment questionnaire, depending on the number of transactions you handle.

The following set of nested variables define whereabouts we will be storing the various files that are generated by the scans:

```
report:
  report_remote_path: "/tmp/{{ inventory_hostname }}_report_{{ report_name }}.html"
  report_local_path: "generated/{{ inventory_hostname }}_report_{{ report_name }}.html"
  results: "/tmp/{{ inventory_hostname }}_results_{{ report_name }}.xml"
```

As you can see, we have remote and local paths for the HTML reports. This is because we will be copying the reports to our Ansible controller as part of the playbook run.

Now that we have the shared variables, we need to add a single default variable to the `roles/scan/defaults/main.yml` file:

```
scan_command: >
  oscap xccdf eval --profile {{ oscap.profile }}
    --fetch-remote-resources
    --results-arf {{ report.results }}
    --report {{ report.report_remote_path }}
    {{ oscap.policy_path }}{{ oscap.policy }}
```

This is the command we will be running to initiate the scan. At the time of writing, there are not any OpenSCAP modules, so we will need to execute the

`oscap` commands using the `command` module. The interesting thing to note is that I have split the command over multiple lines in the variable so that it is easy to read.

Because I used `>`, Ansible will actually render the command on a single line when it comes to applying the variable to the task, which means that we do not have to add `\` at the end of each line like we would need to if we were to run the multiline command on the command line.

The final part of the role is the tasks themselves. We will be putting all of the tasks in the `roles/scan/tasks/main.yml` file, starting with the task that executes the command we defined:

```
- name: run the openscap scan
  command: "{{ scan_command }}"
  args:
    creates: "{{ report.report_remote_path }}"
  ignore_errors: yes
```

The `ignore_errors` is extremely important here. As far as Ansible is concerned, this task will always run unless we get a 100% clean bill of health from the scan. The next task copies the HTML report generated by the scan from our target host to our Ansible controller:

```
- name: download the html report
  fetch:
    src: "{{ report.report_remote_path }}"
    dest: "{{ report.report_local_path }}"
    flat: yes
```

Now that we have two roles in place, we can look at running our first scan.

Running the initial scan

Now that we have both the install and scan roles completed, we can run our first scan. The only file we have not covered yet is the `site.yml` one; this one looks like slightly different to the ones we have been using in other chapters:

```
---
- hosts: scap
  gather_facts: true
  become: yes
  become_method: sudo

  vars_files:
    - group_vars/common.yml

  roles:
    - { role: install, tags: [ "scan" ] }
    - { role: scan, tags: [ "scan" ], report_name: "01-initial-scan" }
```

As you can see, we are tagging the roles as well as passing a parameter when running the scan. For now, we are just going to run playbook without using any tags. To run the playbook, issue the following command:

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

This will give us the following results:

```
PLAY [scap]
*****
TASK [Gathering Facts]
*****
ok: [box1]

TASK [install : install the packages needed]
*****
changed: [box1] => (item=openscap-scanner)
changed: [box1] => (item=scap-security-guide)

TASK [scan : run the openscap scan]
*****
fatal: [box1]: FAILED! => {"changed": true, "cmd": ["oscap", "xccdf", "eval", "--profile", "xccdf_org.ssgproject.content_profile_pci-dss", "--fetch-remote-resources", "--results-arf", "/tmp/box1_results_01-initial-scan.xml", "--report", "/tmp/box1_report_01-initial-scan.html", "/usr/share/xml/scap/ssg/content/ssg-centos7-ds.xml"], "delta": "0:01:03.459407", "end": "2018-05-16 08:17:50.970321", "msg": "non-zero return code", "rc": 2, "start": "2018-05-16 08:16:47.510914", "stderr": "Downloading: https://www.redhat.com/security/data/oval/com.redhat.rhsa-RHEL7.xml.bz2 ... ok", "stderr_lines": ["Downloading: https://www.redhat.com/security/data/oval/com.redhat.rhsa-RHEL7.xml.bz2 ... ok"], "stdout": "", "stdout_lines": []}
```

```
"stdout": "Title\r\tEnsure Red Hat GPG Key\nInstalled\nRule\r\ttxccdf_org.ssgproject.content_rule_ensure_redhat_gpgkey_installed\nRes\n  gpgcheck Enabled In Main Yum\n  \"\\txccdf_org.ssgproject.content_rule_chronyd_or_ntpd_specify_multiple_servers\",\n\"Result\", \"\\tpass\"]}\n..ignoring\n\nTASK [scan : download the html report]\n*****\nchanged: [box1]\n\nPLAY RECAP\n*****\nbox1 : ok=4 changed=3 unreachable=0 failed=0
```

I have truncated the scan results in this output, but when you run it, you will see a large section of failed output colored red. As already mentioned, this is to be expected and is nothing to worry about.

A copy of the HTML report for our initial scan should be on your Ansible controller now; you can open it in your browser with the following command:

```
| $ open generated/box1_report_01-initial-scan.html
```

Or, open the `generated` folder and double-click on `box1_report_01-initial-scan.html`:



Evaluation Characteristics

Evaluation target	10.20.30.40.nip.io	CPE Platforms	Addresses
Benchmark URL	/usr/share/xml/ssg/content/ssg-centos7.ds.xml	<ul style="list-style-type: none">cpe:/o:centos:centos:7cpe:/o:redhat:enterprise_linux:7::crcpe:/o:redhat:enterprise_linux:7::clcpe:/o:redhat:enterprise_linux:7::ce	<ul style="list-style-type: none">IPv4 127.0.0.1IPv4 172.16.117.132IPv4 10.20.30.40IPv6 0:0:0:0:0:0:1IPv6 fe80:0:0:0:455f:df89:ba88:9280IPv6 fe80:0:0:0:20c:29ff:febb:a365MAC 00:00:00:00:00:00MAC 00:0C:29:BB:A3:5BMAC 00:0C:29:BB:A3:65
Benchmark ID	xccdf_org.ssgproject.content_benchmark_RH7		
Profile ID	xccdf_org.ssgproject.content_profile_pci-dss		
Started at	2018-05-16T08:16:52		
Finished at	2018-05-16T08:17:48		
Performed by	vagrant		

Compliance and Scoring

The target system did not satisfy the conditions of 51 rules! Please review rule results and consider applying remediation.

Rule results



Severity of failed rules



Score

Scoring system	Score	Maximum	Percent
urn:xccdf:scoring:default	52.762619	100.000000	<div style="width: 52.76%; background-color: #28a745; height: 10px;"></div> 52.76%

As you can see from the example, our host failed 51 of the 94 checks OpenSCAP ran. Let's look at doing something about getting that number of failed checks down.

Generating the remediation Ansible playbook

Before we continue, I must first draw your attention to the fact the report gives the following warning:

Do not attempt to implement any of the settings in this guide without first testing them in a non-operational environment. The creators of this guidance assume no responsibility whatsoever for its use by other parties and makes no guarantees, expressed or implied, about its quality, reliability, or any other characteristic.

While we are only targeting a test host here, if you like what you see and decide to look at implementing OpenSCAP against other workloads, please ensure that you take it slowly and test thoroughly before running against anything that is in use, even if it is just by developers—the changes made during the remediation we are about to carry out could have serious consequences on the running of your target host.

Now that we have that warning out of the way, we can continue to look at securing our host using an automatically generated Ansible playbook:

```
| $ ansible-galaxy init roles/fix-ansible
```

With this role, we need a few defaults that define whereabouts our generated playbook will be sorted, and again we need to define the command that needs to run. These values can be found in `roles/fix-ansible/defaults/main.yml`.

The first block deals with where the files we are going to be generating are stored on both the target host and locally:

```
playbook_file:
  remote: "/tmp/{{ inventory_hostname }}_ansible.yml"
  local: "generated/{{ inventory_hostname }}_ansible.yml"
  log: "generated/{{ inventory_hostname }}_ansible.log"
```

Next up, we have the command that needs to be executed to generate the playbook file:

```

ansible_fix_command: >
  oscap xccdf generate fix
    --profile {{ oscap.profile }}
    --template urn:xccdf:fix:script:ansible
    --output {{ playbook_file.remote }}
    {{ report.results }}

```

Then, we have the locations of some folders and files that need to be in place before the playbook runs; otherwise, it will result in an error and a fail:

```

missing_folders:
  - "/etc/dconf/db/local.d/locks/"

missing_files:
  - "/etc/dconf/db/local.d/locks/00-security-settings-lock"
  - "/etc/sysconfig/prelink"

```

Now that we have the default variables in place, we can start adding tasks to `roles/fix-ansible/tasks/main.yml`, starting with one that uses the `file` module to put the missing folders and files in place:

```

- name: fix missing folders
  file:
    path: "{{ item }}"
    state: "directory"
  with_items: "{{ missing_folders }}"

- name: fix missing files
  file:
    path: "{{ item }}"
    state: "touch"
  with_items: "{{ missing_files }}"

```

Next, we are going to add a check to see whether the playbook file already exists on the target machine:

```

- name: do we already have the playbook?
  stat:
    path: "{{ playbook_file.remote }}"
  register: playbook_check

```

We are doing this so that we have a way of skipping running the playbook that has been generated. Next up, we run the command to generate the playbook:

```

- name: generate the ansible playbook with the fixes
  command: "{{ ansible_fix_command }}"
  args:
    creates: "{{ playbook_file.remote }}"
    ignore_errors: yes

```

As you can see, we are passing arguments that tell Ansible the command that creates the playbook file; if the file is there, then the command will not execute

again. Now that we have the playbook on the machine, we need to copy it to our Ansible controller. Here, we are using the `fetch` module again:

```
- name: download the ansible playbook
  fetch:
    src: "{{ playbook_file.remote }}"
    dest: "{{ playbook_file.local }}"
    flat: yes
  when: playbook_check.stat.exists == False
```

As you can see, we are using `when` so that the task only runs if the playbook file did not exist at the start of the role being run. Now that we have a copy of the playbook locally, we can run it. To do that, we are going to be using the `local_action` module in combination with the `command` module to run Ansible within Ansible:

```
- name: run the ansible playbook locally
  local_action:
    module: "command ansible-playbook -i production --become --become-method sudo {{ playbook_file.local }}"
    become: no
    register: playbook_run
  when: playbook_check.stat.exists == False
```

There are a few different things happening here, so let's break it down a little more, starting with the command we are running, which translates to:

```
$ ansible-playbook -i production --become --become-method sudo
generated/box1_ansible.yml
```

As you can see, we are having to pass the instructions for using `become` with the method of `sudo` as part of the command. This is because the Ansible playbook that is being generated does not account for you connecting externally using a user other than root.

The final task in this role writes the results of the previous task to a file on our Ansible controller:

```
- name: write the results to a log file
  local_action:
    module: "copy content={{ playbook_run.stdout }} dest={{ playbook_file.log }}"
    become: no
  when: playbook_check.stat.exists == False
```

That completes the role. We can run the playbook again to apply the fixes and remediation, then run another scan so that we can update the `site.yml` file so it reads:

```

---
- hosts: scap
  gather_facts: true
  become: yes
  become_method: sudo

  vars_files:
    - group_vars/common.yml

  roles:
    - { role: install, tags: [ "scan" ] }
    - { role: scan, tags: [ "scan" ], report_name: "01-initial-scan" }
    - { role: fix-ansible, report_name: "01-initial-scan" }
    - { role: scan, report_name: "02-post-ansible-fix" }

```

As you can see, we have removed the tags for the `fix-ansible` role and we have also updated the report name for the second `scan`. We can start the playbook by running this:

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

This gives us the following output:

```

PLAY [scap]
*****
TASK [Gathering Facts]
*****
ok: [box1]

TASK [install : update all of the installed packages]
*****
ok: [box1]

TASK [install : install the packages needed]
*****
ok: [box1] => (item=openscap-scanner)
ok: [box1] => (item=scap-security-guide)

TASK [scan : run the openscap scan]
*****
ok: [box1]

TASK [scan : download the html report]
*****
ok: [box1]

TASK [fix-ansible : fix missing folders]
*****
changed: [box1] => (item=/etc/dconf/db/local.d/locks/)

TASK [fix-ansible : fix missing files]
*****
changed: [box1] => (item=/etc/dconf/db/local.d/locks/00-security-settings-lock)
changed: [box1] => (item=/etc/sysconfig/prelink)

TASK [fix-ansible : do we already have the playbook?]
*****
```

```
ok: [box1]

TASK [fix-ansible : generate the ansible playbook with the fixes]
*****
changed: [box1]

TASK [fix-ansible : download the ansible playbook]
*****
changed: [box1]

TASK [fix-ansible : run the ansible playbook locally]
*****
changed: [box1 -> localhost]

TASK [fix-ansible : write the results to a log file]
*****
changed: [box1 -> localhost]

TASK [scan : run the openscap scan]
*****
fatal: [box1]: FAILED! =>
...ignoring

TASK [scan : download the html report]
*****
changed: [box1]

PLAY RECAP
*****
box1 : ok=14 changed=8 unreachable=0 failed=0
```

Let's take a look at the report and see what difference running the Ansible playbook has made:

```
| $ open generated/box1_report_02-post-ansible-fix.html
```

The output can be seen as follows:



Compliance and Scoring

The target system did not satisfy the conditions of 25 rules! Please review rule results and consider applying remediation.

Rule results



Severity of failed rules



Score

Scoring system	Score	Maximum	Percent
urn:xccdf:scoring:default	89.522018	100.000000	<div style="width: 89.52%; background-color: #28a745; height: 10px;"></div> 89.52%

That's a little better than before; however, we are still failing 25 rules—why is that? Well, as already mentioned, work is still ongoing with porting all of the remediation rules over to Ansible; for example, if you open up the original scan results and scroll to the bottom, you should see that the Set SSH Idle Timeout Interval check failed.

Clicking on it will show you information on what OpenSCAP is checking, why they are checking it, and also why it should be fixed. And finally, at the bottom, you will notice that there are options to show both the shell and Ansible remediation solutions:

Set SSH Idle Timeout Interval

Rule ID	xccdf_org.ssgproject.content_rule_sshd_set_idle_timeout
Result	fail
Time	2018-05-16T15:49:46
Severity	low
Identifiers and References	References: SV-86861r2_rule, AC-2(5), SA-8(l), AC-12, CCI-001133, CCI-002361, SRG-OS-000163-GPOS-00072, SRG-OS-000279-GPOS-00109, Req-8.1.8, 5.2.12, 5.5.6, 3.1.11
Description	<p>SSH allows administrators to set an idle timeout interval. After this interval has passed, the idle user will be automatically logged out.</p> <p>To set an idle timeout interval, edit the following line in <code>/etc/ssh/sshd_config</code> as follows:</p> <div style="border: 1px solid #ccc; padding: 5px; background-color: #f9f9f9;"><code>ClientAliveInterval interval</code></div> <p>The timeout <code>interval</code> is given in seconds. To have a timeout of 10 minutes, set <code>interval</code> to 600.</p> <p>If a shorter timeout has already been set for the login shell, that value will preempt any SSH setting made here. Keep in mind that some processes may stop SSH from correctly detecting that the user is idle.</p>
Rationale	Terminating an idle ssh session within a short time period reduces the window of opportunity for unauthorized personnel to take control of a management session enabled on the console or console port that has been left unattended.
Remediation Shell script:	(show)
Remediation Ansible snippet:	(show)

Now, click on one of the remaining failures from the second report. You should notice that there is only an option for remediation using a shell script. We will be generating this in the next role, but before we move on, let's quickly look at the

playbook that was generated.

The playbook I generated at the time of writing contained over 3,200 lines of code, so I am not going to cover them all here, but as we have already mentioned the Set SSH Idle Timeout Interval check, let's take a look at the task in the playbook that applies the fix:

```
- name: Set SSH Idle Timeout Interval
  lineinfile:
    create: yes
    dest: /etc/ssh/sshd_config
    regexp: ^ClientAliveInterval
    line: "ClientAliveInterval {{ sshd_idle_timeout_value }}"
    validate: sshd -t -f %s
  #notify: restart sshd
  tags:
    - sshd_set_idle_timeout
    - low_severity
    - restrict_strategy
    - low_complexity
    - low_disruption
    - CCE-27433-2
    - NIST-800-53-AC-2(5)
    - NIST-800-53-SA-8(i)
    - NIST-800-53-AC-12
    - NIST-800-171-3.1.11
    - PCI-DSS-Req-8.1.8
    - CJIS-5.5.6
    - DISA-STIG-RHEL-07-040320
```

As you can see, it uses the lineinfile module to apply a variable that is defined at the very top of the playbook. Also, each of the tasks is tagged with quite a lot of information about which areas of the standard the fix comes under, and also the severity. This means we can get quite granular on which parts of the playbook run; for example, you could only run the low disruption changes by using the following command:

```
$ ansible-playbook -i production --become --become-method sudo --tags "low_disruption"
generated/box1_ansible.yml
```

Finally, at the bottom of the `box1_ansible.log` file, we can see that the playbook run made the following changes:

```
PLAY RECAP
*****
box1 : ok=151 changed=85 unreachable=0 failed=0
```

Generating the remediation bash script

To remediate the remaining issues, we should generate and execute the bash script:

```
| $ ansible-galaxy init roles/fix-bash
```

As this is a nice-to-have, I am not going to go into any detail about the ins and outs of what we are adding here. The contents of `roles/fix-bash/defaults/main.yml` are similar to those in the `fix-ansible` role:

```
bash_file:
  remote: "/tmp/{{ inventory_hostname }}_bash.sh"
  log: "generated/{{ inventory_hostname }}_bash.log"

bash_fix_command: >
  oscap xccdf generate fix
  --profile {{ oscap.profile }}
  --output {{ bash_file.remote }}
  {{ report.results }}
```

The tasks in `roles/fix-bash/tasks/main.yml` are also similar and shouldn't need any explanation:

```
- name: do we already have the bash script?
  stat:
    path: "{{ bash_file.remote }}"
  register: bash_script_check

- name: generate the bash script
  command: "{{ bash_fix_command }}"
  args:
    creates: "{{ bash_file.remote }}"
  ignore_errors: yes

- name: run the bash script
  command: "bash {{ bash_file.remote }}"
  ignore_errors: yes
  register: bash_run
  when: bash_script_check.stat.exists == False

- name: write the results to a log file
  local_action:
    module: "copy content={{ bash_run.stdout }} dest={{ bash_file.log }}"
  become: no
  when: bash_script_check.stat.exists == False
```

Update the `site.yml` file so it reads:

```
- hosts: scap
  gather_facts: true
  become: yes
  become_method: sudo

  vars_files:
    - group_vars/common.yml

  roles:
    - { role: install, tags: [ "scan" ] }
    - { role: scan, tags: [ "scan" ], report_name: "01-initial-scan" }
    - { role: fix-ansible, report_name: "01-initial-scan" }
    - { role: scan, report_name: "02-post-ansible-fix" }
    - { role: fix-bash, report_name: "02-post-ansible-fix" }
    - { role: scan, report_name: "03-post-bash-fix" }
```

This means we can take the results of the scan that ran after the Ansible fix to generate the bash script that contains the remaining fixes; we are then doing one final scan. To apply the final batch of fixes, run the following:

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

This gives the following output:

```
PLAY [scap]
*****
TASK [Gathering Facts]
*****
ok: [box1]

TASK [install : update all of the installed packages]
*****
ok: [box1]

TASK [install : install the packages needed]
*****
ok: [box1] => (item=openscap-scanner)
ok: [box1] => (item=scap-security-guide)

TASK [scan : run the openscap scan]
*****
ok: [box1]

TASK [scan : download the html report]
*****
ok: [box1]

TASK [fix-ansible : fix missing folders]
*****
ok: [box1] => (item=/etc/dconf/db/local.d/locks/)

TASK [fix-ansible : fix missing files]
*****
changed: [box1] => (item=/etc/dconf/db/local.d/locks/00-security-settings-lock)
changed: [box1] => (item=/etc/sysconfig/prelink)
```

```

TASK [fix-ansible : do we already have the playbook?]
*****
ok: [box1]

TASK [fix-ansible : generate the ansible playbook with the fixes]
*****
skipping: [box1]

TASK [fix-ansible : download the ansible playbook]
*****
skipping: [box1]

TASK [fix-ansible : run the ansible playbook locally]
*****
skipping: [box1]

TASK [fix-ansible : write the results to a log file]
*****
skipping: [box1]

TASK [scan : run the openscap scan]
*****
ok: [box1]

TASK [scan : download the html report]
*****
ok: [box1]

TASK [fix-bash : do we already have the bash script?]
*****
ok: [box1]

TASK [fix-bash : generate the bash script]
*****
changed: [box1]

TASK [fix-bash : run the bash script]
*****
changed: [box1]

TASK [fix-bash : write the results to a log file]
*****
changed: [box1 -> localhost]

TASK [scan : run the openscap scan]
*****
fatal: [box1]: FAILED! =>
...ignoring

TASK [scan : download the html report]
*****
changed: [box1]

PLAY RECAP
*****
box1 : ok=16 changed=6 unreachable=0 failed=0

```

Check the final report by running:

```
| $ open generated/box1_report_03-post-bash-fix.html
```

This should show that the overall number of failed checks has reduced to just five:

Compliance and Scoring

The target system did not satisfy the conditions of 5 rules! Please review rule results and consider applying remediation.

Rule results

89 passed 5 failed

Severity of failed rules

1 low 4 medium

Score

Scoring system	Score	Maximum	Percent
urn:xccdf:scoring:default	95.312500	100.000000	95.31%

Running a standalone scan

When we created the scan role, it was mentioned that the role should be reusable. We also added tags to the role when we defined it in the `site.yml` file. Let's take a quick look at how we can run just a scan outside of a full playbook run. To kick off the scan, run the following command:

```
$ ansible-playbook -i production --tags "scan" --extra-vars "report_name=scan-only" site.yml
```

This will run only the parts of the playbook that are tagged `scan`, and we are also overriding the `report_name` variable that we are setting as part of calling the role in the `site.yml` file to call our `report_box1_report_scan-only.html`:

```
1. scap (bash)
russ in ~/scap
⚡ ansible-playbook -i production --tags "scan" --extra-vars "report_name=scan-only" site.yml

PLAY [scap] ****
TASK [Gathering Facts] ****
ok: [box1]

TASK [install : update all of the installed packages] ****
ok: [box1]

TASK [install : install the packages needed] ****
ok: [box1] => (item=openscap-scanner)
ok: [box1] => (item=scap-security-guide)

TASK [scan : run the openscap scan] ****
fatal: [box1]: FAILED! => {"changed": true, "cmd": ["oscap", "xccdf", "eval", "--profile", "xccdf_org.ssgproject.content_profile_pci-dss", "--fetch-remote-resources", "--results-arf", "/tmp/box1_results_scan-only.xml", "--report", "/tmp/box1_report_scan-only.html", "/usr/share/xml/scap/ssg/content/ssg-centos7-ds.xml"], "delta": "0:00:55.750166", "end": "2018-05-16 17:12:22.514364", "msg": "non-zero return code", "rc": 2, "start": "2018-05-16 17:11:26.764198", "stderr": "Downloading: https://www.
```

Fixing the remaining failed checks

So far, we haven't had to put any hardcoded fixes in place to resolve any of the problems found in the scans. We have had to create a few files and folders to allow the fixes to be applied, but that was more to let the automated remediation work, rather than a fix.

At the time of writing, there are known issues with two of the five problems that are currently showing on my scans; these are:

- `xccdf_org.ssaproject.content_rule_audit_rules_privileged_commands`
- `xccdf_org.ssaproject.content_rule_audit_rules_login_events`

There are fixes being worked on. You can find them on Red Hat's Bugzilla at:

- https://bugzilla.redhat.com/show_bug.cgi?id=1570802
- https://bugzilla.redhat.com/show_bug.cgi?id=1574586

So, leaving these two to one side, there are three I can fix now. To do this, I am going to create a separate role and playbook, as by the time you read this, the following fixes may not be needed:

```
| $ ansible-galaxy init roles/final-fixes
```

Jumping straight into `roles/final-fixes/tasks/main.yml`, our first fix is to rotate the logs daily rather than weekly, which is the default. To do this, we will use the `lineinfile` module to replace `weekly` with `daily`:

```
- name: sort out the logrotate
  lineinfile:
    path: "/etc/logrotate.conf"
    regexp: "^weekly"
    line: "daily"
```

The next task adds a fix that should make its way through to the `scap-security-guide` package at some point:

```
- name: add the missing line to the modules.rules
  lineinfile:
    path: "/etc/audit/rules.d/modules.rules"
    line: "-a always,exit -F arch=b32 -S init_module -S delete_module -k modules"
```

As you can see, here, we are using the `lineinfile` module again. This time, we are adding a line to `/etc/audit/rules.d/modules.rules` if it is not already present. This adds a rule that takes into account 32-bit kernels, as well as the 64-bit ones, which the remediation scripts already configured.

Next, we are adding a fix for a script that should have been executed during the bash script execution. First of all, we need to create a file using the `file` module:

```
- name: add file for content_rule_file_permissions_var_log_audit
  file:
    path: "/var/log/audit/audit.log.fix"
    state: "touch"
```

We then need to copy and then execute the portion of the bash script that failed when we first ran it:

```
- name: copy the content_rule_file_permissions_var_log_audit.sh script
  copy:
    src: "content_rule_file_permissions_var_log_audit.sh"
    dest: "/tmp/content_rule_file_permissions_var_log_audit.sh"

- name: run the content_rule_file_permissions_var_log_audit.sh script
  command: "bash /tmp/content_rule_file_permissions_var_log_audit.sh"
```

The bash script itself can be found at `roles/final-fixes/files/content_rule_file_permissions_var_log_audit.sh`, and it looks like this:

```
if `grep -q ^log_group /etc/audit/auditd.conf` ; then
  GROUP=$(awk -F "=" '/log_group/ {print $2}' /etc/audit/auditd.conf | tr -d ' ')
  if ! [ "${GROUP}" == 'root' ] ; then
    chmod 0640 /var/log/audit/audit.log
    chmod 0440 /var/log/audit/audit.log.*
  else
    chmod 0600 /var/log/audit/audit.log
    chmod 0400 /var/log/audit/audit.log.*
  fi

  chmod 0640 /etc/audit/audit*
  chmod 0640 /etc/audit/rules.d/*
else
  chmod 0600 /var/log/audit/audit.log
  chmod 0400 /var/log/audit/audit.log.*
  chmod 0640 /etc/audit/audit*
  chmod 0640 /etc/audit/rules.d/*
fi
```

Finally, we need to create a playbook file called `final-fixes.yml`. This should run the role we have just created and then run a final scan:

```
---
- hosts: scap
```

```

gather_facts: true
become: yes
become_method: sudo

vars_files:
  - group_vars/common.yml

roles:
  - { role: final-fixes }
  - { role: scan, report_name: "04-final-fixes" }

```

To run the playbook, use the following command:

```
| $ ansible-playbook -i production final-fixes.yml
```

This will give the following results:

```

PLAY [scap]
*****
TASK [Gathering Facts]
*****
ok: [box1]

TASK [final-fixes : sort out the logrotate]
*****
changed: [box1]

TASK [final-fixes : add the missing line to the modules.rules]
*****
changed: [box1]

TASK [final-fixes : add file for content_rule_file_permissions_var_log_audit]
*****
changed: [box1]

TASK [final-fixes : copy the content_rule_file_permissions_var_log_audit.sh script]
*****
changed: [box1]

TASK [final-fixes : run the content_rule_file_permissions_var_log_audit.sh script]
*****
changed: [box1]

TASK [scan : run the openscap scan]
*****
fatal: [box1]: FAILED! =>
...ignoring

TASK [scan : download the html report]
*****
changed: [box1]

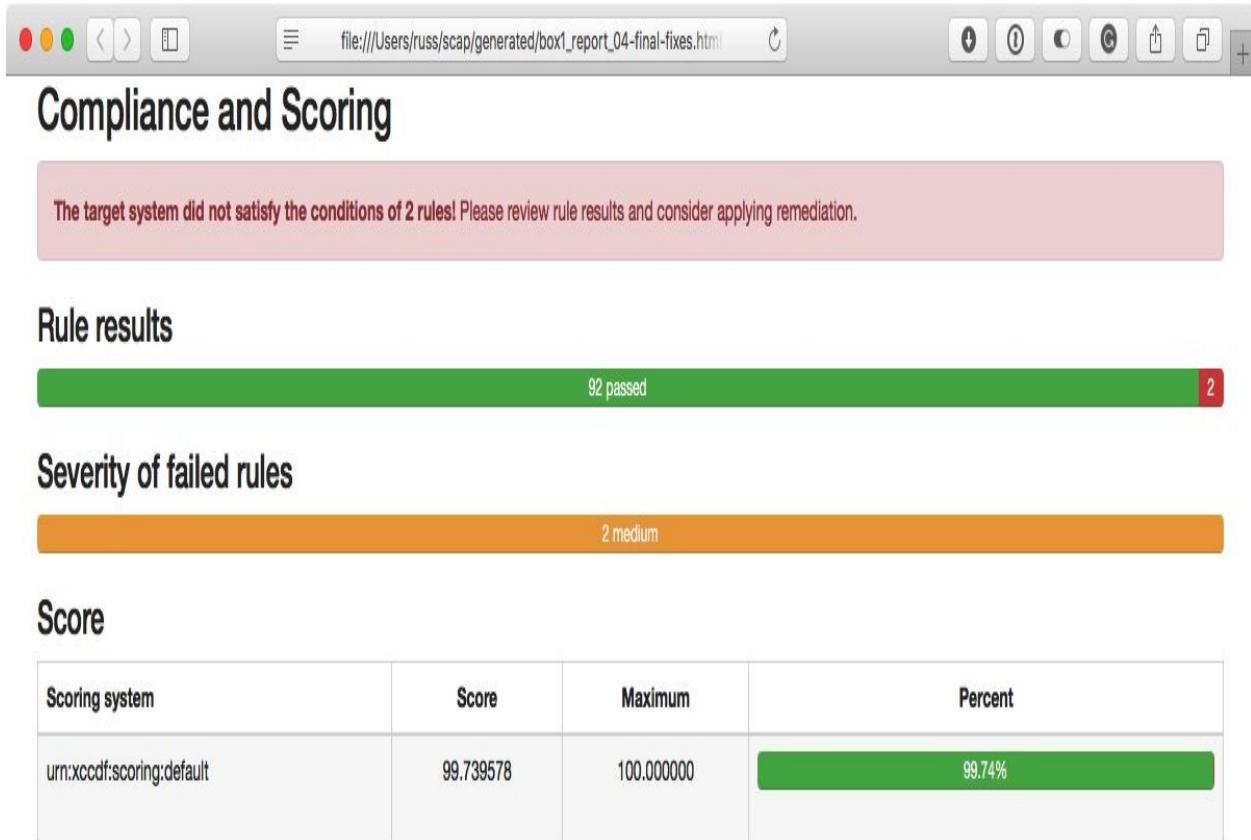
PLAY RECAP
*****
box1 : ok=8 changed=7 unreachable=0 failed=0

```

Open the report that was generated using the following command:

```
| $ open generated/box1_report_04-final-fixes.html
```

This shows us that there are just the two medium checks with known issues that are still failing:



Hopefully, by the time you are reading this, your hosts will be getting a clean bill of health and this last section won't be needed, which is why I split it off from the main `site.yml` playbook.

Destroying the Vagrant box

Don't forget to destroy the Vagrant box once you have finished; you don't want an idle virtual machine running on your host. To do this, run: **\$ vagrant destroy**

Once the box has gone, I would recommend running through the scan and remediation a few times on a clean installation to get an idea of how this can be implemented on new hosts.

Summary

In this chapter, we created a playbook that generated a playbook to remediate any PCI-DSS non-compliance errors found during a scan. As well as being really cool, it is also really practical if you imagine you are running a few dozen servers that all need to be compliant, and also that they all need a full audit history.

You now have the foundations of a playbook that you can use to target those hosts daily, to both audit them and store the results away from the host itself, but also depending on your configuration, you have a way of automatically resolving any non-conformance found during the scan.

The scans we have been doing in this chapter have all been host-based; in the next chapter, we are going to look at how we can scan hosts remotely.

Questions

1. What effect does adding > to a multiline variable have?
2. True or false: OpenSCAP is certified by NIST.
3. Why are we telling Ansible to continue if the `scan` command is marked as failed?
4. Explain why we are using tags for certain roles.
5. True or false: We are using the `copy` command to copy the HTML reports from the remote host to the Ansible controller.

Further reading

You can find out more information on the technologies and organizations we have covered in this chapter at the following links:

- **OpenSCAP:** <https://www.open-scap.org/>
- **Security Content Automation Protocol (SCAP):** <https://scap.nist.gov/>
- **NIST:** <https://www.nist.gov/>
- **MITRE Corporation:** <https://www.mitre.org/>
- **Asset Identification (AID):** <https://csrc.nist.gov/Projects/Security-Content-Automation-Protocol/Specifications/aid>
- **Asset Reporting Format (ARF):** <https://csrc.nist.gov/Projects/Security-Content-Automation-Protocol/Specifications/arf>
- **Common Configuration Enumeration (CCE):** <https://cce.mitre.org>
- **Common Configuration Scoring System (CCSS):** <https://www.nist.gov/publications/common-configuration-scoring-system-ccss-metrics-software-security-configuration>
- **Common Platform Enumeration (CPE):** <https://nvd.nist.gov/products/cpe>
- **Common Weakness Enumeration (CWE):** <https://cwe.mitre.org/>
- **Common Vulnerabilities and Exposures (CVE):** <https://cve.mitre.org>
- **Common Vulnerability Scoring System (CVSS):** <https://www.first.org/cvss/>
- **Extensible Configuration Checklist Description Format (XCCDF):** <https://csrc.nist.gov/Projects/Security-Content-Automation-Protocol/Specifications/xccdf>
- **Open Checklist Interactive Language (OCIL):** <https://csrc.nist.gov/Projects/Security-Content-Automation-Protocol/Specifications/ocil>
- **Open Vulnerability and Assessment Language (OVAL):** <https://oval.mitre.org>
- **Trust Model for Security Automation Data (TMSAD):** <https://www.nist.gov/publications/trust-model-security-automation-data-10-tmsad>

Deploying WPScan and OWASP ZAP

In this chapter, we will look at creating a playbook that deploys and runs two security tools, WPScan and OWASP ZAP. Then, using the playbooks from previous chapters, we will launch a WordPress installation for us to scan.

As with other chapters, we will be using Vagrant and one of the boxes we have already downloaded. You can find a complete copy of the playbook at [https://git
hub.com/PacktPublishing/Learn-Ansible/tree/master/Chapter14](https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter14).

Preparing the boxes

We are going to be launching two Vagrant boxes for this chapter, the first of which we will be using to install the scanning tools. This host will have Docker installed, and we will be using the Docker Ansible modules to interact with the software. The second box will contain or host the WordPress installation, which will be targeted by the scanning tools.

Create a `Vagrantfile` with the following content:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

API_VERSION = "2"
BOX_NAME = "centos/7"
BOX_IP_SCAN = "10.20.30.40"
BOX_IP_WP = "10.20.30.41"
DOMAIN = "nip.io"
PRIVATE_KEY = "~/.ssh/id_rsa"
PUBLIC_KEY = '~/.ssh/id_rsa.pub'

Vagrant.configure(API_VERSION) do |config|
  config.vm.define :scan do |scan|
    scan.vm.box = BOX_NAME
    scan.vm.network "private_network", ip: BOX_IP_SCAN
    scan.vm.host_name = BOX_IP_SCAN + '.' + DOMAIN
    scan.ssh.insert_key = false
    scan.ssh.private_key_path = [PRIVATE_KEY, "~/.vagrant.d/insecure_private_key"]
    scan.vm.provision "file", source: PUBLIC_KEY, destination: "~/.ssh/authorized_keys"
  end

  config.vm.define :wp do |wp|
    wp.vm.box = BOX_NAME
    wp.vm.network "private_network", ip: BOX_IP_WP
    wp.vm.host_name = BOX_IP_WP + '.' + DOMAIN
    wp.ssh.insert_key = false
    wp.ssh.private_key_path = [PRIVATE_KEY, "~/.vagrant.d/insecure_private_key"]
    wp.vm.provision "file", source: PUBLIC_KEY, destination: "~/.ssh/authorized_keys"
  end

  config.vm.provider "virtualbox" do |v|
    v.memory = "2024"
    v.cpus = "2"
  end

  config.vm.provider "vmware_fusion" do |v|
    v.vmx["memsize"] = "2024"
    v.vmx["numvcpus"] = "2"
  end
end
```

As you can see, we are going to be launching two CentOS 7 boxes, one labelled `scan`, which has a hostname of `10.20.30.40.nip.io`, and the other `wp`, which has a hostname of `10.20.30.41.nip.io`.

The inventory host file, which is always called `production`, contains the following:

```
box1 ansible_host=10.20.30.40.nip.io
box2 ansible_host=10.20.30.41.nip.io

[scan]
box1

[wordpress]
box2

[boxes]
box1
box2

[boxes:vars]
ansible_connection=ssh
ansible_user=vagrant
ansible_private_key_file=~/ssh/id_rsa
host_key_checking=False
```

As you can see, we have defined three host groups; the first group called `scan` includes the single host which we will use to run the scanning tools. The second group, `wordpress`, while only containing a single host, could have more than one hosted listed, and the scans should target them all. The third group, called `boxes`, has been defined as a way of applying the connection configuration to all the hosts we have added to the playbook.

You can launch the two boxes using one of the following two commands:

```
$ vagrant up
$ vagrant up --provider=vmware_fusion
```

Now we have our Vagrant boxes up and running, we can take a look at what our playbook looks like.

The WordPress playbook

As you have already guessed, this is going to be extremely simple, as we already have written a playbook that deploys WordPress on a CentOS 7 host. In fact, all we need to do is copy the `group_vars`, `roles` folders, plus their contents, and also the `site.yml` file from the `chapter05/lemp` folder in the repository and we are done.

This is one of the great advantages of using a tool such as Ansible: Write once and use often; the only change we will be making is the `site.yml` file when we add in the plays that deploy the software.

The scan playbook

As already mentioned, we are going to be using Docker to run both WPScan and also OWASP ZAP. The reason for this is we would end up deploying quite a bit of supporting software if we were to install both packages directly on the host—while this is not a problem, using a tool such as Docker allows to simplify the installation process as well as giving us an excuse to cover the Docker Ansible modules.

The Docker role

As with all of the roles we have created so far, we are going to use the `ansible-galaxy` command to generate the structure for our role:

```
| $ ansible-galaxy init roles/docker
```

For our Docker installation, we will be using the `yum` repository provided by Docker itself; this means that before we install, Docker needs to enable the repository. Once enabled, we will be able to install the latest stable release. Let's make a start by populating some defaults in `roles/docker/defaults/main.yml`:

```
docker:
  gpg_key: "https://download.docker.com/linux/centos/gpg"
  repo_url: "https://download.docker.com/linux/centos/docker-ce.repo"
  repo_path: "/etc/yum.repos.d/docker-ce.repo"
  packages:
    - "docker-ce"
    - "device-mapper-persistent-data"
    - "lvm2"
    - "python-setuptools"
    - "libselinux-python"
  pip:
    - "docker"
```

As you can see, we are defining the URL to where the repo's GPG key is stored, the URL for the repository file, and where on the host the repository file should be copied. We also have a list of the packages that need to be installed to get Docker up and running. Finally, we have the Python package for Docker that will allow Ansible to interact with the Docker API on our Vagrant box.

Before we use any of the variables defined, we need to make sure that the host we are running is up to date with its packages, so the first task in `roles/docker/tasks/main.yml` should perform a `yum update`:

```
- name: update all of the installed packages
  yum:
    name: "*"
    state: "latest"
    update_cache: "yes"
```

Now our host is up to date, we can add the GPG key; for this we will use the `rpm_key` module where we simply have to supply the URL, or file path, to the key

we want to install:

```
- name: add the gpg key for the docker repo
  rpm_key:
    key: "{{ docker.gpg_key }}"
    state: "present"
```

Now we have the GPG key installed, we can download the `docker-ce.repo` file from Docker and store it where `yum` will pick it up when it is next executed:

```
- name: add docker repo from the remote url
  get_url:
    url: "{{ docker.repo_url }}"
    dest: "{{ docker.repo_path }}"
    mode: "0644"
```

As you can see, we are using the `get_url` module to download the file and place it in `/etc/yum.repos.d/` on our host machine; we are also setting the read, write, and execute permissions on the file to `0644`.

Now that we have the Docker repository configured, we can install the packages we defined by adding the following task:

```
- name: install the docker packages
  yum:
    name: "{{ item }}"
    state: "installed"
    update_cache: "yes"
  with_items: "{{ docker.packages }}"
```

We have added the `update_cache` option as we have just added a new repository and want to make sure that it is picked up. Next, we have to install the Docker Python package using `pip`; by default `pip` is not installed so we need to make sure it is available first by using `easy_install`, which in turn was installed by the `python-setuptools` package, which was installed with the previous task. There is an `easy_install` module, so this task is simple:

```
- name: install pip
  easy_install:
    name: pip
    state: latest
```

Now that `pip` is available, we can use the `pip` module to install the Docker Python library:

```
- name: install the python packages
  pip:
    name: "{{ item }}"
```

```
|   with_items: "{{ docker.pip }}"
```

The penultimate task is to disable SELinux on the Vagrant box:

```
- name: put selinux into permissive mode
  selinux:
    policy: targeted
    state: permissive
```

By default, the version of Docker provided by Docker does not automatically start on CentOS/Red Hat servers, so the final task in this role is to start the Docker service and also make sure that it is configured to start on boot:

```
- name: start docker and configure to start on boot
  service:
    name: "docker"
    state: "started"
    enabled: "yes"
```

We have done this at this part of the playbook run rather than using a handler, as the playbook needs to interact with Docker before the playbook completes. As handlers are only called at the end of a playbook run, that would mean that the next part of our playbook would fail. Before we start to download and run containers, let's quickly run the playbook.

Testing the playbook

As we have all of the basic roles in place, we can try running the playbook; before doing so, we need to update the `site.yml` to include a play for our scan host:

```
---
- hosts: scan
  gather_facts: true
  become: yes
  become_method: sudo

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/docker

- hosts: wordpress
  gather_facts: true
  become: yes
  become_method: sudo

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/stack-install
    - roles/stack-config
    - roles/wordpress
```

Once updated, we can run our playbook using the code:

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

This should give us something like the following output:

```
PLAY [scan]
*****
TASK [Gathering Facts]
*****
ok: [box1]

TASK [roles/docker : update all of the installed packages]
*****
changed: [box1]

TASK [roles/docker : add the gpg key for the docker repo]
*****
changed: [box1]
```

```
TASK [roles/docker : add docker repo from the remote url]
*****
changed: [box1]

TASK [roles/docker : install the docker packages]
*****
changed: [box1] => (item=[u'docker-ce', u'device-mapper-persistent-data', u'lvm2',
u'python-setuptools', u'libselinux-python'])

TASK [roles/docker : install pip]
*****
changed: [box1]

TASK [roles/docker : install the python packages]
*****
changed: [box1] => (item=docker)

TASK [roles/docker : put selinux into permissive mode]
*****
changed: [box1]

TASK [roles/docker : start docker and configure to start on boot]
*****
changed: [box1]

PLAY [wordpress]
*****

TASK [Gathering Facts]
*****
ok: [box2]

TASK [roles/stack-install : install the repo packages]
*****
changed: [box2] => (item=[u'epel-release', u'https://centos7.iuscommunity.org/ius-
release.rpm'])

TASK [roles/stack-install : add the NGINX mainline repo]
*****
changed: [box2]

TASK [roles/stack-install : update all of the installed packages]
*****
changed: [box2]

TASK [roles/stack-install : remove the packages so that they can be replaced]
*****
changed: [box2] => (item=[u'mariadb-libs.x86_64'])

TASK [roles/stack-install : install the stack packages]
*****
changed: [box2] => (item=[u'postfix', u'MySQL-python', u'policycoreutils-python',
u'nginx', u'mariadb101u', u'mariadb101u-server', u'mariadb101u-config', u'mariadb101u-
common', u'mariadb101u-libs', u'php72u', u'php72u-bcmath', u'php72u-cli', u'php72u-
common', u'php72u-dba', u'php72u-fpm', u'php72u-fpm-nginx', u'php72u-gd', u'php72u-
intl', u'php72u-json', u'php72u-mbstring', u'php72u-mysqld', u'php72u-process',
u'php72u-snmp', u'php72u-soap', u'php72u-xml', u'php72u-xmlrpc', u'vim-enhanced',
u'git', u'unzip'])

TASK [roles/stack-config : add the wordpress user]
*****
changed: [box2]

TASK [roles/stack-config : copy the nginx.conf to /etc/nginx/]
```

```
*****
changed: [box2]

TASK [roles/stack-config : create the global directory in /etc/nginx/]
*****
changed: [box2]

TASK [roles/stack-config : copy the restrictions.conf to /etc/nginx/global/]
*****
changed: [box2]

TASK [roles/stack-config : copy the wordpress_shared.conf to /etc/nginx/global/]
*****
changed: [box2]

TASK [roles/stack-config : copy the default.conf to /etc/nginx/conf.d/]
*****
changed: [box2]

TASK [roles/stack-config : copy the www.conf to /etc/php-fpm.d/]
*****
changed: [box2]

TASK [roles/stack-config : configure php.ini]
*****
changed: [box2] => (item={u'regexp': u'^date.timezone =', u'replace': u'date.timezone = Europe/London'})
changed: [box2] => (item={u'regexp': u'^expose_php = On', u'replace': u'expose_php = Off'})
changed: [box2] => (item={u'regexp': u'^upload_max_filesize = 2M', u'replace': u'upload_max_filesize = 20M'})

TASK [roles/stack-config : start php-fpm]
*****
changed: [box2]

TASK [roles/stack-config : start nginx]
*****
changed: [box2]

TASK [roles/stack-config : configure the mariadb bind address]
*****
changed: [box2]

TASK [roles/stack-config : start mariadb]
*****
changed: [box2]

TASK [roles/stack-config : change mysql root password]
*****
changed: [box2] => (item=127.0.0.1)
changed: [box2] => (item=:1)
changed: [box2] => (item=10.20.30.41.nip.io)
changed: [box2] => (item=localhost)

TASK [roles/stack-config : set up .my.cnf file]
*****
changed: [box2]

TASK [roles/stack-config : delete anonymous MySQL user]
*****
ok: [box2] => (item=127.0.0.1)
ok: [box2] => (item=:1)
changed: [box2] => (item=10.20.30.41.nip.io)
```

```
changed: [box2] => (item=localhost)

TASK [roles/stack-config : remove the MySQL test database]
*****
changed: [box2]

TASK [roles/stack-config : set the selinux allowing httpd_t to be permissive is
required] *****
changed: [box2]

TASK [roles/wordpress : download wp-cli]
*****
changed: [box2]

TASK [roles/wordpress : update permissions of wp-cli to allow anyone to execute it]
*****
changed: [box2]

TASK [roles/wordpress : create the wordpress database]
*****
changed: [box2]

TASK [roles/wordpress : create the user for the wordpress database]
*****
changed: [box2] => (item=127.0.0.1)
ok: [box2] => (item=:1)
ok: [box2] => (item=10.20.30.41.nip.io)
ok: [box2] => (item=localhost)

TASK [roles/wordpress : are the wordpress files already there?]
*****
ok: [box2]

TASK [roles/wordpress : download wordpresss]
*****
changed: [box2]

TASK [roles/wordpress : set the correct permissions on the homedir]
*****
changed: [box2]

TASK [roles/wordpress : is wordpress already configured?]
*****
ok: [box2]

TASK [roles/wordpress : configure wordpress]
*****
changed: [box2]

TASK [roles/wordpress : do we need to install wordpress?]
*****
fatal: [box2]: FAILED! =>
...ignoring

TASK [roles/wordpress : install wordpress if needed]
*****
changed: [box2]

TASK [roles/wordpress : do we need to install the plugins?]
*****
failed: [box2] (item=jetpack) =>
failed: [box2] (item=wp-super-cache) =>
failed: [box2] (item=wordpress-seo) =>
failed: [box2] (item=wordfence) =>
```

```

failed: [box2] (item=nginx-helper) =>
...ignoring

TASK [roles/wordpress : set a fact if we don't need to install the plugins]
*****
skipping: [box2]

TASK [roles/wordpress : set a fact if we need to install the plugins]
*****
ok: [box2]

TASK [roles/wordpress : install the plugins if we need to or ignore if not]
*****
changed: [box2] => (item=jetpack)
changed: [box2] => (item=wp-super-cache)
changed: [box2] => (item=wordpress-seo)
changed: [box2] => (item=wordfence)
changed: [box2] => (item=nginx-helper)

TASK [roles/wordpress : do we need to install the theme?]
*****
fatal: [box2]: FAILED! =>
...ignoring

TASK [roles/wordpress : set a fact if we don't need to install the theme]
*****
skipping: [box2]

TASK [roles/wordpress : set a fact if we need to install the theme]
*****
ok: [box2]

TASK [roles/wordpress : install the theme if we need to or ignore if not]
*****
changed: [box2]

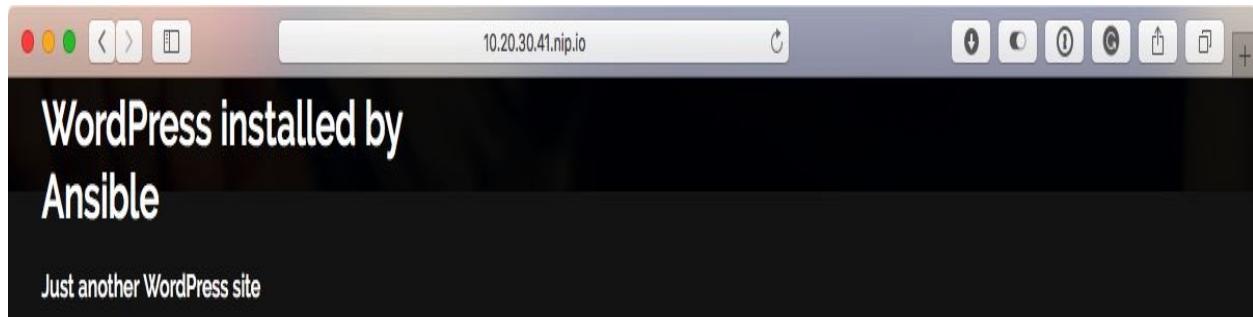
RUNNING HANDLER [roles/stack-config : restart nginx]
*****
changed: [box2]

RUNNING HANDLER [roles/stack-config : restart php-fpm]
*****
changed: [box2]

PLAY RECAP
*****
box1 : ok=9 changed=8 unreachable=0 failed=0
box2 : ok=42 changed=37 unreachable=0 failed=0

```

As you can see, this has performed the full Docker and WordPress installations; opening <http://10.20.30.41.nip.io/> will take you to the WordPress site:



UNCATEGORIZED

Hello world!

POSTED ON MAY 18, 2018

Welcome to WordPress. This is your first post. Edit or delete it, then start writing!

RECENT POSTS

Hello world!

Now we have our WordPress site up and running, we can make a start on the roles that perform a scan of the site.

The WPScan role

The first role we are going to create is one that runs WPScan. WPScan is a tool that performs a scan of a WordPress site; it tries to figure out the version of WordPress that is running as well as checking for any plugins that have known vulnerabilities. It can also try and brute force the admin user account; however, we are going to skip that.

As always, we can bootstrap the role using this:

```
| $ ansible-galaxy init roles/wpscan
```

Once the files are in place, we need to add the following to `roles/wpscan/default/main.yml`:

```
image: "wpscanteam/wpscan"
log:
  remote_folder: /tmp/wpscan/
  local_folder: "generated/"
  file: "{{ ansible_date_time.date }}-{{ ansible_date_time.hour }}-{{ ansible_date_time.minute }}.txt"
```

This sets the image we want to download from the Docker Hub; in this case, it is the official WPScan image from the WPScan team. We then set the variables we wish to use for the logs; you may notice that we are defining a folder and file name for the logs.

Next up, we need to add the tasks to `roles/wpscan/tasks/main.yml`, the first of which uses the `docker_image` module to pull a copy of the `wpscanteam/wpscan` image:

```
- name: pull the image
  docker_image:
    name: "{{ image }}"
```

Next up, we need to create the folder where the logs will be written to our Vagrant box:

```
- name: create the folder which we will mount inside the container
  file:
    path: "{{ log.remote_folder }}"
    state: "directory"
    mode: "0777"
```

The reason we are doing this is that we will be mounting this folder inside of the container we are going to be launching in the next task. As the logs are the only bits of data we want to keep from each scan, we will write them to the mounted folder, meaning that the logs will be available for us to copy to our Ansible controller once the container has exited and removed.

Before we look at the next task, let's take a quick look at the command we would need to run to launch the scan if we were using Docker directly on the command line:

```
$ docker container run -it --rm --name wpscan -v /tmp/wpscan/:/tmp/wpscan/
wpscanteam/wpscan \
-u http://10.20.30.41.nip.io/ --update --enumerate --log
/tmp/wpscan/10.20.30.41.nip.io-2018-05-19-12-16.txt
```

The first line of the command is where all of the Docker logic happens; what we are asking Docker to do is to launch (`run`) a container called `wpscan` (`--name`) in the foreground (`-it`), mounting `/tmp/wpscan/` from the host to `/tmp/wpscan/` inside the container (`-v`) using the specified image (`wpscanteam/wpscan`). Once the process exits, we remove the container (`--rm`).

Everything on the second line is passed to the container's default entry point, which in the case of the `wpscanteam/wpscan` image is `/wpscan/wpscan.rb`, meaning that the command we are using to run the scan within the container is actually this:

```
$ /wpscan/wpscan.rb -u http://10.20.30.41.nip.io/ --update --enumerate --log
/tmp/wpscan/10.20.30.41.nip.io-2018-05-19-12-16.txt
```

Now we have an idea of the command we would run using Docker, we can look at how that would look in our task:

```
- name: run the scan
  docker_container:
    detach: false
    auto_remove: true
    name: "wpscan"
    volumes: "{{ log.remote_folder }}:{{ log.remote_folder }}"
    image: "{{ image }}"
    command: "--url http://{{ hostvars[item]['ansible_host'] }} --update --enumerate --
log {{ log.remote_folder }}{{ hostvars[item]['ansible_host'] }}-{{ log.file }}"
    register: docker_scan
    failed_when: docker_scan.rc == 0 or docker_scan.rc >= 2
  with_items: "{{ groups['wordpress'] }}"
```

The options in the task are in the same order as they were written in the Docker command:

- `detach: false` is similar to passing `-it` in that it will run the container in the foreground; by default the `docker_container` module runs containers in the background. This introduces a few challenges, which we will discuss shortly.
- `auto_remove: true` is the same as `--rm`.
- `name: "wpscan"` is exactly the same as running `--name wpscan`.
- `volumes:"{{ log.remote_folder }}:{{ log.remote_folder }}"` is the same as what we would pass using the `-v` flag in Docker.
- `image: "{{ image }}"` is the equivalent of just passing the image name, for example, `wpscanteam/wpscan`.
- Finally, `command` contains everything we want to pass to the entry-point; as you can see we have passed a few dynamic variables here.

As mentioned, the `docker_container` module by default runs the containers in the background; typically that would be great in most cases; however, as we are just running the container as a one-off job to perform our scan, then we need to run it in the foreground.

Doing this will actually cause an error as we are instructing Ansible to remain attached to a container, which then terminates and is removed once the scan process completes. To get around this, we are registering the results of the task and, rather than using `ignore_errors`, we are telling the task to fail when (`failed_when`) the return code (`rc`) is equal to `0` or equal or more than `2`, as our task should always have a return code of `1`.

So why not just let the container run in the background? As the next task copies the log file from the Vagrant box to the Ansible Controller, if we were to let the container run in the background, Ansible would immediately move on to the next task and copy a partly written file.

Attaching to the container and waiting for it to exit means that we are waiting for the scan to complete before progressing to the next task, which looks like this:

```

- name: download the html report
  fetch:
    src: "{{ log.remote_folder }}{{ hostvars[item]['ansible_host'] }}-{{ log.file }}"
    dest: "{{ log.local_folder }}{{ hostvars[item]['ansible_host'] }}-{{ log.file }}"
    flat: yes
  with_items: "{{ groups['wordpress'] }}"

```

Now we have our tasks written, we can try running our role.

Running a WPScan

To run a scan, update the `site.yml` file to look like the following code:

```
- hosts:  
  wordpress  
gather_facts: true  
become: yes  
become_method: sudo
```

```
vars_files:  
- group_vars/common.yml
```

```
roles:  
  - roles/stack-install  
  - roles/stack-config  
  - roles/wordpress  
  
- hosts: scan  
gather_facts: true  
become: yes  
become_method: sudo  
  
vars_files:  
  - group_vars/common.yml  
  
roles:  
  - roles/docker  
  - roles/wpscan
```

Then run the following command:

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

This should give you the following results (the screenshot shows just the scan rather than the full playbook run, which you should see):

```
1. chapter14 (bash)
⚡ ansible-playbook -i production site.yml

PLAY [scan] ****
TASK [Gathering Facts] ****
ok: [box1]

TASK [roles/wpscan : pull the image] ****
changed: [box1]

TASK [roles/wpscan : create the folder which we will mount inside the container] ****
changed: [box1]

TASK [roles/wpscan : run the scan] ****
ok: [box1] => (item=box2)

TASK [roles/wpscan : download the html report] ****
changed: [box1] => (item=box2)

PLAY RECAP ****
box1                  : ok=5    changed=3    unreachable=0    failed=0
```

Also, you should find a log file in the generated folder; this contains the results of the WPScan run:

The screenshot shows a terminal window with the following content:

```
FOLDERS
└ chapter14
  └─ generated
    └─ 10.20.30.41.nip.io-2018-05-19-13-04.txt — chapter14

10.20.30.41.nip.io-2018-05-19-13-04.txt
1
2
3
4
5
6
7
8
9
9
10
11
12
13
14
15
15
16
16
17
17
18
18
19
19
20
20
21
21
22
22
23
23
24
24
25
25
26
26
27
27
28
28
29
29
30
31
31
32
32
33
33
34
34
```

WPScan output:

```
WordPress Security Scanner by the WPScan Team
Version 2.9.4-dev
Sponsored by Sucuri - https://Sucuri.net
@WPScan_, @ethicalhack3r, @erwan_lr, @FireFart_
[+] Updating the Database ...
[i] Update completed.
[+] URL: http://10.20.30.41.nip.io/
[+] Started: Sat May 19 13:05:10 2018
[+] Interesting header: LINK: <http://10.20.30.41.nip.io/wp-json/>;
rel="https://api.w.org/"
[+] Interesting header: SERVER: nginx/1.13.12
[+] Interesting header: X-CONTENT-TYPE-OPTIONS: nosniff
[+] Interesting header: X-FRAME-OPTIONS: SAMEORIGIN
[+] Interesting header: X-XSS-PROTECTION: 1; mode=block
[+] XML-RPC Interface available under: http://10.20.30.41.nip.io/xmlrpc.php
[+] WordPress version 4.9.6 (Released on 2018-05-17) identified from advanced
fingerprinting, meta generator, links opml, stylesheets numbers
[+] WordPress theme in use: sydney - v1.45
[+] Name: sydney - v1.45
| Latest version: 1.45 (up to date)
| Last updated: 2018-02-13T00:00:00.000Z
| Location: http://10.20.30.41.nip.io/wp-content/themes/sydney/
```

Bottom status bar: Line 1, Column 1 | Spaces: 4 | Plain Text

As you can see, there is quite a bit of information; however, since we deployed our WordPress installation from scratch, we should have a clean bill of health.

The OWASP ZAP role

Now that we have covered the basics of how to run a container using Ansible in the WPScan role, creating the role that runs OWASP ZAP should be straightforward; we just use this command:

```
| $ ansible-galaxy init roles/zap
```

Open Web Application Security Project Zed Attack Proxy or OWASP ZAP, to give it its full name, is an open source web application security scanner.

The defaults for the role in `roles/zap/defaults/main.yml` should contain this code:

```
image: "owasp/zap2docker-stable"
log:
  remote_folder: /tmp/zap/
  local_folder: "generated/"
  file: "{{ ansible_date_time.date }}-{{ ansible_date_time.hour }}-{{ ansible_date_time.minute }}.html"
```

As you can see, we are using the `owasp/zap2docker-stable` image and also we are using the `/tmp/zap/` folder to store the report files in on the Vagrant box.

Moving on to the tasks in `roles/zap/tasks/main.yml`, we are pulling the image and creating the folder as we did in the WPScan role:

```
- name: pull the image
  docker_image:
    name: "{{ image }}"
- name: create the folder which we will mount inside the container
  file:
    path: "{{ log.remote_folder }}"
    state: "directory"
    mode: "0777"
```

Let's take a look at the `docker` command we would have run to figure out what we needed to put in the next task:

```
$ docker container run -it --rm --name zap -v /tmp/zap/:/zap/wrk/ owasp/zap2docker-stable \
  zap-baseline.py -t http://10.20.30.41.nip.io/ -g gen.conf -r 10.20.30.41.nip.io-2018-05-19-14-26.html
```

As you can see, the command is using all of the options we used before; there

are differences in where we are mounting our folder within the container, as OWASP ZAP is expecting us to write any files we want to save to `/zap/wrk/`. This means we do not have to provide a full filesystem path when giving the report name as the application will write to `/zap/wrk/` by default.

This means that the task which launches the container should look like the following code:

```
- name: run the scan
  docker_container:
    detach: false
    auto_remove: true
    name: "zap"
    volumes: "{{ log.remote_folder }}:/zap/wrk/"
    image: "{{ image }}"
    command: "zap-baseline.py -t http://{{ hostvars[item]['ansible_host'] }} -g
gen.conf -r {{ hostvars[item]['ansible_host'] }}-{{ log.file }}"
    register: docker_scan
    failed_when: docker_scan.rc == 0 or docker_scan.rc >= 2
  with_items: "{{ groups['wordpress'] }}"
```

We are then downloading the report using the following task:

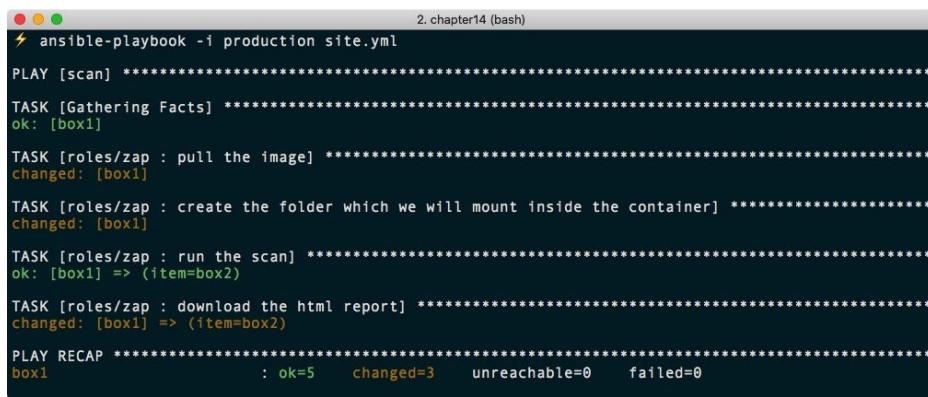
```
- name: download the html report
  fetch:
    src: "{{ log.remote_folder }}{{ hostvars[item]['ansible_host'] }}-{{ log.file }}"
    dest: "{{ log.local_folder }}{{ hostvars[item]['ansible_host'] }}-{{ log.file }}"
    flat: yes
  with_items: "{{ groups['wordpress'] }}"
```

Now we have our tasks in place, we can run the role.

Running OWASP ZAP

To run the scan, we simply need to append the role to the end of our `site.yml` file. Once added, run the following command: **\$ ansible-playbook -i production site.yml**

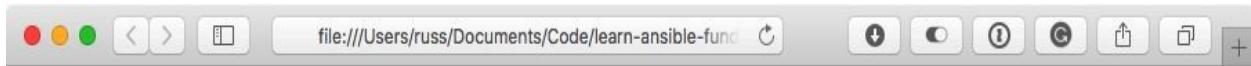
This will run through the playbook; an abridged copy of the output can be found here:



The screenshot shows a terminal window titled "2. chapter14 (bash)". It displays the output of an Ansible playbook named "site.yml". The output shows the execution of various tasks, including gathering facts, pulling the image, creating a folder, running the scan, and downloading the HTML report. The terminal shows the progress with "ok" and "changed" status indicators. Finally, it provides a "PLAY RECAP" summary with statistics: box1 : ok=5 changed=3 unreachable=0 failed=0.

```
ansible-playbook -i production site.yml
PLAY [scan] *****
TASK [Gathering Facts] *****
ok: [box1]
TASK [roles/zap : pull the image] *****
changed: [box1]
TASK [roles/zap : create the folder which we will mount inside the container] *****
changed: [box1]
TASK [roles/zap : run the scan] *****
ok: [box1] => (item=box2)
TASK [roles/zap : download the html report] *****
changed: [box1] => (item=box2)
PLAY RECAP *****
box1 : ok=5    changed=3    unreachable=0    failed=0
```

This will then copy an HTML file to the generated folder; the file should look similar to the following:



ZAP Scanning Report

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	1
Low	3
Informational	2

Alert Detail

Medium (Medium)	Multiple X-Frame-Options Header Entries
Description	X-Frame-Options (XFO) headers were found, a response with multiple XFO header entries may not be predictably treated by all user-agents.
URL	http://10.20.30.41.nip.io/wp-login.php
Method	POST
Parameter	X-Frame-Options
URL	http://10.20.30.41.nip.io/wp-login.php?action=lostpassword
Method	POST
Parameter	X-Frame-Options
URL	http://10.20.30.41.nip.io/wp-login.php

Now you can remove the Vagrant boxes using this command:

```
| $ vagrant destroy
```

Then relaunch the boxes and run through the playbook in its entirety.

Summary

In this chapter, we have looked at using Ansible in combination with Docker to launch two different tools that performed an external vulnerability scan against the WordPress installation we launched using an Ansible playbook. This shows how we can launch some quite complex tools without having to worry about writing a playbook to install, configure, and manage them directly on our hosts.

In the next chapter, we are going to move off the command line and look at the two web-based interfaces for Ansible provided by Red Hat.

Questions

1. Why are we using Docker rather than installing WPScan and OWASP ZAP directly on our Vagrant box?
2. True or false: `pip` is installed on our Vagrant box by default.
3. What is the name of the Python module we need to install for the Ansible Docker modules to function?
4. Update the `vagrantfile` and `production` files to launch a second WordPress Vagrant box and scan them both.

Further reading

For more information on the tools we have used in this chapter, please see the following links:

- **Docker:** <https://docker.com>
- **WPScan:** <https://wpscan.org>
- **OWASP ZAP:** https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

Introducing Ansible Tower and Ansible AWX

In this chapter, we are going to be looking at the two graphical interfaces for Ansible, the commercial Ansible Tower and the open source Ansible AWX. We will discuss how to install them, what the differences are, and also why you would need to use them. After all, we are now 15 chapters into our journey with Ansible and have yet to need to use a graphical interface.

By the end of this chapter, we will have:

- Installed Ansible Tower and Ansible AWX
- Configured both tools
- Deployed our highly available cloud application using Ansible Tower

Technical requirements

We will be looking at using Ansible Tower and Ansible AWX locally using a Vagrant box; we are also going to be using the playbook we covered in [Chapter 10](#), *Highly Available Cloud Deployments*. The final playbooks can be found in the GitHub repository at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter15>.

Web-based Ansible

Before we look at installing the tools, we should first take the time to discuss why we need them and also the differences between them.

I am sure you are starting to notice a common thread between all of the playbooks we have covered so far—where possible, we are allowing the roles we are running to use as many parameters as possible. This makes it easy for us to change the output of the playbook run without having to rewrite or edit the roles directly. For this reason, it should also be easy for us to start using one of the two web-based tools provided by Red Hat for managing your Ansible deployments.

Ansible Tower is a commercially licensed, web-based graphical interface for Ansible. As already mentioned, you might be struggling to see the value in this. Imagine being able to hook Ansible up to your company's active directory and have users, such as developers, use Ansible Tower to deploy their own environments based on your playbooks, providing a controlled way for you to maintain consistency across your estate while allowing self-service.

When Red Hat announced its acquisition of Ansible in October 2015, one of the questions in the FAQ it published on the day of the announcement was: *Will Red Hat open source all of Ansible's technology?* The reason the question was asked was that with other technologies Red Hat has acquired over the years, it has open sourced almost all aspects of them to not only invite community contributions, but to test and build new features, which eventually made their way into Red Hat's commercially supported version.

An example of this is the Fedora project. This project is the open source upstream for Red Hat Enterprise Linux features—which Fedora users are taking advantage of now—including DNF, a YUM replacement. This has been the default package manager in Fedora since 2015 and, if everything goes as planned, it should make its way into Red Hat Enterprise Linux 8.

Other examples of Red Hat open sourcing its technologies include WildFly, which is an upstream for JBoss, and the ManageIQ, which is sponsored by Red

Hat and is the basis for Red Hat CloudForms.

In September 2017, Red Hat announced it would be releasing Ansible AWX, an open source upstream for Ansible Tower. This project would have fortnightly releases with the AWX team, making select releases *stable*, although in this case, stable does not mean production ready as the project is still in its initial development cycle.

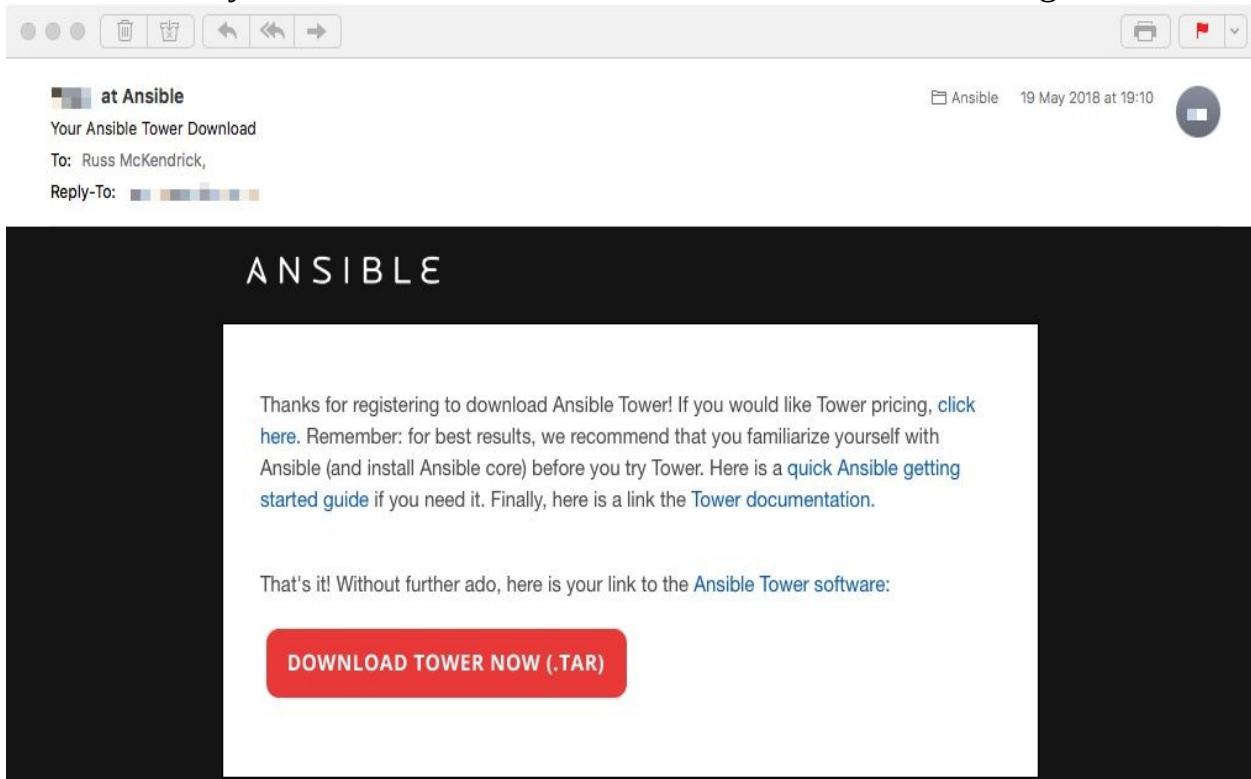
Ansible Tower

We are going to start by looking at Ansible Tower. As you may recall, this is commercial software, so we will need a license; luckily, Red Hat provides a trial license. You can request it by clicking on the Try Tower Free button at <https://www.ansible.com/>.



Please note, you must use a business address, Ansible will not accept requests which originate from an email address such as me.com, icloud.com, gmail.com, and hotmain.com and so on.

After a while, you will receive an email that looks like the following:



Click on the DOWNLOAD TOWER NOW (.TAR) button; this will open your browser and download a TAR file containing the playbooks we will be running to deploy Ansible Tower. Next up, we need a server to host our Ansible Tower installation. Let's use the `vagrantfile` we have used in other chapters: # -*- mode: ruby -*-
vi: set ft=ruby :

```
API_VERSION = "2"
BOX_NAME = "centos/7"
BOX_IP = "10.20.30.40"
DOMAIN = "nip.io"
PRIVATE_KEY = "~/.ssh/id_rsa"
PUBLIC_KEY = '~/.ssh/id_rsa.pub'
```

```
Vagrant.configure(API_VERSION) do |config|
  config.vm.box = BOX_NAME
  config.vm.network "private_network", ip: BOX_IP
  config.vm.host_name = BOX_IP + '.' + DOMAIN
  config.ssh.insert_key = false
  config.ssh.private_key_path = [PRIVATE_KEY,
    "~/.vagrant.d/insecure_private_key"]
  config.vm.provision "file", source: PUBLIC_KEY, destination:
    "~/.ssh/authorized_keys"
```

```
config.vm.provider "virtualbox" do |v|
  v.memory = "2024"
  v.cpus = "2"
end
```

```
config.vm.provider "vmware_fusion" do |v|
  v.vmx["memsize"] = "2024"
  v.vmx["numvcpus"] = "2"
end
```

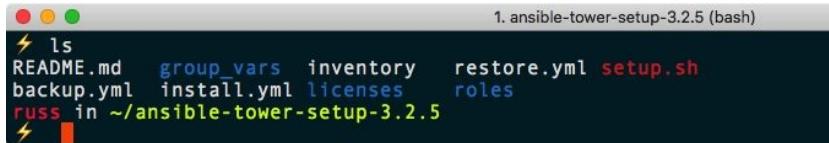
```
end
```

Once the `vagrantfile` is in place, you can launch the Vagrant box using one of the following commands: **\$ vagrant up**
\$ vagrant up --provider=vmware_fusion

Once you have the Vagrant box up and running, you can look at what changes you need to make to the inventory, which is contained within the TAR file we downloaded.

Updating the inventory file

There are several files provided in the top level of the unarchived folder, to unarchive the folder double-click on the TAR file:



```
1. ansible-tower-setup-3.2.5 (bash)
ls
README.md  group_vars  inventory  restore.yml  setup.sh
backup.yml  install.yml  licenses  roles
russ in ~/ansible-tower-setup-3.2.5
```

We just need to worry about the `inventory` file; open the file in a text editor and update it so it looks like the following: [tower]

```
10.20.30.40.nip.io ansible_connection=ssh ansible_user=vagrant
ansible_private_key_file=~/ssh/id_rsa host_key_checking=False
```

```
[database]
```

```
[all:vars]
```

```
admin_password='password'
```

```
pg_host=""
pg_port=""
```

```
pg_database='awx'
pg_username='awx'
pg_password='iHpkiPEAHpGeR8paCoVhwLPH'
```

```
rabbitmq_port=5672
rabbitmq_vhost=tower
rabbitmq_username=tower
rabbitmq_password='WUwTLJK2AtdxCfopcXFQoVYs'
rabbitmq_cookie=cookiemonster
```

```
# Needs to be true for fqdns and ip addresses
rabbitmq_use_long_name=true
```

```
# Isolated Tower nodes automatically generate an RSA key for authentication;
```

```
# To disable this behavior, set this value to false  
# isolated_key_generation=true
```

As you can see, we have updated the host listed under the `[tower]` group to include the details and configuration for our Vagrant box; we have also added a password for the `admin_password`, `pg_password`, and `rabbitmq_password` parameters. Obviously, you can set your own passwords rather than using the ones listed here.

The final change to the file is that `rabbitmq_use_long_name` has been updated from `false` to `true`. A failure to do this would result in the RabbitMQ service failing to start.

Running the playbook

Now that we have updated the `inventory` file, we can run the `install.yml` playbook to kick off the Ansible Tower installation. To do this, run the following command:

```
| $ ansible-playbook -i inventory --become install.yml
```

There are checks built into the playbook to see if the playbook is being run as the root user. In a typical installation, the playbook is expecting you to be running the playbook as the root user on the machine you want Ansible Tower to be installed on. However, we are doing things slightly differently, so we need to use the `--become` flag.

The installation process takes around 20 minutes and, as you can see from the following output, the installer works through a lot of tasks:

```
1. ansible-tower-setup-3.2.5 (bash)
changed: [10.20.30.40.nip.io] => (item=tower)

TASK [misc : Create the default organization if it is needed.] ****
changed: [10.20.30.40.nip.io]

RUNNING HANDLER [supervisor : restart supervisor] ****
changed: [10.20.30.40.nip.io] => {
    "msg": "Restarting supervisor."
}

RUNNING HANDLER [supervisor : Stop supervisor.] ****
changed: [10.20.30.40.nip.io]

RUNNING HANDLER [supervisor : Wait for supervisor to stop.] ****
ok: [10.20.30.40.nip.io]

RUNNING HANDLER [supervisor : Start supervisor.] ****
changed: [10.20.30.40.nip.io]

RUNNING HANDLER [nginx : restart nginx] ****
changed: [10.20.30.40.nip.io]

PLAY [Install Tower isolated node(s)] ****
skipping: no hosts matched

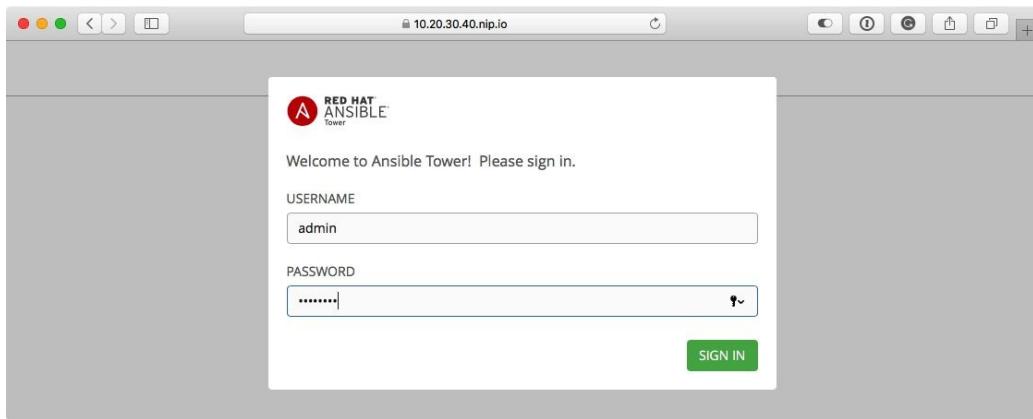
PLAY RECAP ****
10.20.30.40.nip.io      : ok=137  changed=66    unreachable=0    failed=0

russ in ~/ansible-tower-setup-3.2.5
```

Requesting a license

Now that we have Ansible Tower installed, there are a few more steps we need to perform in order to complete the installation. The first step is to log in; to do this, enter the following URL in your browser: <https://10.20.30.40.nip.io/>. When you first open Tower, you will be presented with a warning about the SSL certificate; this is because the certificate that was installed during the deployment is self-signed. It is safe to proceed.

You should now see a login page; enter the USERNAME as `admin` and a PASSWORD of `password`, which is what we set in the `inventory` file earlier:



Then click on the SIGN IN button; this will take you to a page that instructs you to enter a license file:

The screenshot shows a web browser window with the URL 10.20.30.40.nip.io. The title bar says "A TOWER". The main content area is titled "TOWER LICENSE". It instructs the user to visit Ansible's website to get a Tower license key. A "REQUEST LICENSE" button is present. Step 2 indicates to choose a license file, agree to the End User License Agreement, and click submit. A "LICENSE FILE" field contains a file named "license_...txt". A "BROWSE" button is next to it. An "END USER LICENSE AGREEMENT" section contains the Red Hat EULA text and a checkbox for agreeing to it. The checkbox is checked, and the text "I agree to the End User License Agreement" is visible. A "SUBMIT" button is at the bottom right.

Copyright © 2017 Red Hat, Inc.

Clicking the REQUEST LICENSE button will take you to <https://www.ansible.com/license/>; here, you have the option of requesting two types of license for your installation. We are going to be requesting the FREE ANSIBLE TOWER TRIAL - LIMITED FEATURES UP TO 10 NODES license. Select the license type, complete the form, and submit it as prompted.

After a short while, you should receive a few emails, one of which welcomes you to your Ansible Tower installation. The other email contains a license file. Make a copy of the attached license file and use the BROWSE button on the Tower License page to upload it. Also review and agree to the end-user agreement. Once you have uploaded the license file and agreed to the END USER LICENSE AGREEMENT, click on SUBMIT.

After a few seconds, you will be presented with your first look at Ansible Tower:

A TOWER PROJECTS INVENTORIES TEMPLATES JOBS admin 🔍 📁 🖊 🌱

DASHBOARD

HOSTS FAILED HOSTS INVENTORIES INVENTORY SYNC FAILURES PROJECTS PROJECT SYNC FAILURES

JOB STATUS

PERIOD PAST MONTH JOB TYPE ALL VIEW ALL

RECENTLY USED JOB TEMPLATES

No job templates were recently used.
You can create a job template [here](#).

RECENTLY RUN JOBS

No jobs were recently run.

Copyright © 2017 Red Hat, Inc.

Now we have Ansible Tower installed, we can look at running our first playbook.

The hello world demo project

As you can see, we already have a project configured; this is an extremely basic one that downloads the sample playbook from <https://github.com/ansible/ansible-tower-samples/> and displays the message Hello World. Before we can run the playbook, we first need to download a copy from GitHub; to do this, click on PROJECTS in the top menu.

You will be able to see the Demo Project listed. Hovering over the icons under actions will give you a description of what each one will do when you click it; we want to click on the first icon, Start an SCM update. After a short time, you should see that REVISION and **LAST UPDATED** are both populated:

The screenshot shows the Ansible Tower web interface. At the top, there's a navigation bar with icons for red, yellow, and green status, a search bar containing '10.20.30.40.nip.io', and a refresh button. Below the bar, the top menu has tabs for 'TOWER', 'PROJECTS' (which is selected and highlighted in grey), 'INVENTORIES', 'TEMPLATES', and 'JOBS'. On the right side of the header, there are user profile icons for 'admin' and other users, along with settings and system icons. The main content area is titled 'PROJECTS' and contains a table. The table has columns: NAME, TYPE, REVISION, LAST UPDATED, and ACTIONS. One row is visible, showing 'Demo Project' (Type: Git, Revision: 347e44f, Last Updated: 2/6/2018 15:16:10). The 'ACTIONS' column for this row includes icons for delete, edit, and copy. At the bottom right of the table, it says 'ITEMS 1 · 1'.

NAME	TYPE	REVISION	LAST UPDATED	ACTIONS
Demo Project	Git	347e44f	2/6/2018 15:16:10	

This means that Ansible Tower has now downloaded the demo playbook from GitHub; we can now run the playbook. To do this, click on TEMPLATES in the top menu.

Again, you should see that there is a template called Demo Job Template and towards the right-hand side of the line there are several icons. The one we want to click is the one that looks like a rocket. Clicking Start a job using this template will run the demo job; you will be taken to a screen where you can monitor the progress of the job.

Once complete, you should see something that looks like the following:

S 10.20.30.40.nip.io C

TOWER PROJECTS INVENTORIES TEMPLATES JOBS admin 🔍 🚧 📁 🛡️

JOB / 2 - Demo Job Template

DETAILS	
STATUS	Successful
STARTED	2/6/2018 15:22:02
FINISHED	2/6/2018 15:22:10
TEMPLATE	Demo Job Template
JOB TYPE	Run
LAUNCHED BY	admin
INVENTORY	Demo Inventory
PROJECT	Demo Project
REVISION	347e44f 🔍
PLAYBOOK	hello_world.yml
MACHINE CREDENTIAL	Demo Credential
FORKS	0
VERBOSITY	0 (Normal)
INSTANCE GROUP	tower
EXTRA VARIABLES	0

1 ---

Demo Job Template

PLAYS 1 TASKS 2 HOSTS 1 ELAPSED 00:00:08 X 🔍

SEARCH 🔍 KEY

#	Line	Time
1	PLAY [Hello World Sample] *****	15:22:08
2	*****	
3	*****	
4	TASK [Gathering Facts] *****	15:22:08
5	ok: [localhost]	
6	*****	
7	TASK [Hello Message] *****	15:22:10
8	ok: [localhost] => {	
9	"msg": "Hello World!"	
10	}	
11	*****	
12	PLAY RECAP *****	15:22:10
13	localhost : ok=2 changed=0 unreachable=0 failed=0	
14	*****	

^ TOP

Copyright © 2017 Red Hat, Inc.

As you can see, on the left-hand side, you have an overview of the job itself; this tells you the status, when it started and finished, and which user requested the job to be executed. The section on the right-hand side of the page shows the playbook output, which is exactly the same as what we see when executing the playbook from the command line.

Let's take a look at running something more complicated.

Launching the AWS playbook

In [Chapter 10](#), *Highly Available Cloud Deployments*, we worked through a playbook which used the AWS core Ansible modules to launch a cluster running WordPress; there is a standalone version of the `aws-wordpress` playbook hosted on GitHub at <https://github.com/russmckendrick/aws-wordpress/>. Let's use this to deploy our AWS cluster using Ansible Tower.

Before we move on to configuring the playbook within Ansible Tower, we need to do a little housekeeping on the versions of some of the Python modules that were deployed as part of the Ansible Tower installation. This is because there are parts of our playbook that require later versions of the Boto modules.

To do this, we need to SSH into our Ansible Tower host by running the following command:

```
| $ vagrant ssh
```

Now that we are logged in as the Vagrant user, we can change the root with:

```
| $ sudo -i
```

Next, we change to the same Python environment as Ansible Tower uses; to do this, we run the following:

```
| $ source /var/lib/awx/venv/ansible/bin/activate
```

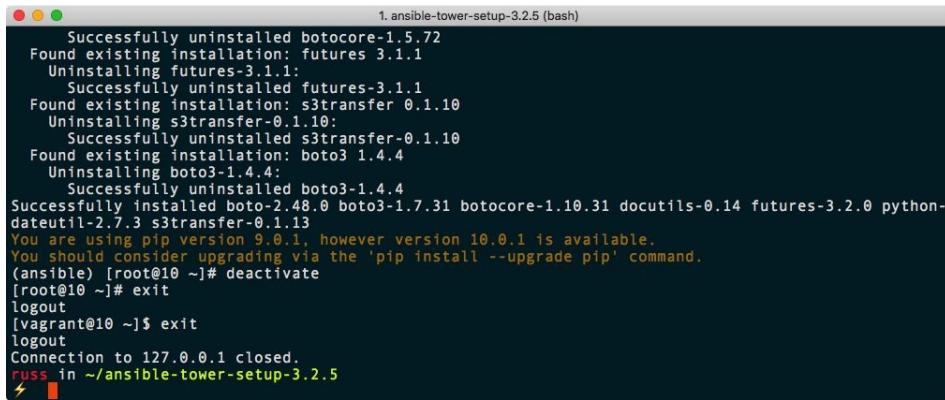
Now we are using the right environment, we need to upgrade the `boto` libraries using the following command:

```
| $ pip install boto boto3 botocore --upgrade
```

Once updated, we can exit the Ansible Tower Python environment by running:

```
| $ deactivate
```

Then we use the `exit` command to log out:



```
Successfully uninstalled botocore-1.5.72
Found existing installation: futures 3.1.1
Uninstalling futures-3.1.1:
  Successfully uninstalled futures-3.1.1
Found existing installation: s3transfer 0.1.10
Uninstalling s3transfer-0.1.10:
  Successfully uninstalled s3transfer-0.1.10
Found existing installation: boto3 1.4.4
Uninstalling boto3-1.4.4:
  Successfully uninstalled boto3-1.4.4
Successfully installed boto-2.48.0 boto3-1.7.31 botocore-1.10.31 docutils-0.14 futures-3.2.0 python-dateutil-2.7.3 s3transfer-0.1.13
You are using pip version 9.0.1, however version 10.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
(ansible) [root@10 ~]# deactivate
[root@10 ~]# exit
logout
[vagrant@10 ~]$ exit
logout
Connection to 127.0.0.1 closed.
russ in ~/ansible-tower-setup-3.2.5
```

Now that we have our environment updated, we can proceed to adding a new project.

Adding a new project

The first thing we need to do is add a new project; this is where we let Ansible Tower know about the repository hosting our playbook. As already mentioned, we will be using a GitHub repository that houses the code. To add a new project, click on PROJECTS in the top menu and then click on the +ADD button, which can be found on the right just under the row of icons in the top menu.

Here, you will be asked for several bits of information; enter the following:

- NAME: AWS Project
- DESCRIPTION: AWS WordPress Cluster
- ORGANIZATION: Default
- SCM TYPE: GIT

When you select the SCM TYPE, a second section will appear that asks for details about where your source is hosted:

- SCM URL: <https://github.com/russmckendrick/aws-wordpress.git>
- SCM BRANCH/TAG/COMMIT: master
- SCM CREDENTIAL: Leave blank as this is a publicly accessible repository
- Clean: Tick
- Delete on Update: Tick
- Update on Launch: Tick
- CACHE TIMEOUT (SECONDS): Leave at zero

Once you have entered the details, click on SAVE. If you now return to the PROJECTS page, you should see that Ansible has already downloaded the source for the playbook:

Screenshot of a web-based application interface titled "TOWER". The browser address bar shows the URL `10.20.30.40.nip.io`. The top navigation bar includes links for "PROJECTS", "INVENTORIES", "TEMPLATES", and "JOBS", along with user authentication ("admin") and configuration icons.

The main content area is titled "PROJECTS" and displays a table of two projects:

NAME	TYPE	REVISION	LAST UPDATED	ACTIONS
AWS Project	Git	2c54f2e	3/6/2018 10:23:07	
Demo Project	Git	347e44f	2/6/2018 15:22:02	

Below the table, a message indicates "ITEMS 1 - 2".

Adding credentials

Next up, we have to let Ansible Tower know the credentials to use when accessing our AWS account; to add these, click on the Settings icon (the cog in the top menu) and you will be taken to a screen which looks like the following:

The screenshot shows the Ansible Tower Settings interface. At the top, there's a navigation bar with icons for dashboard, projects, inventories, templates, and jobs, followed by a search bar containing '10.20.30.40.nip.io' and a refresh button. On the right of the bar are user profile and settings icons. Below the bar, the main header reads 'SETTINGS'. The page is divided into a grid of 12 cards arranged in three rows of four. The cards are: ORGANIZATIONS (Group all of your content to manage permissions across departments in your company.), USERS (Allow others to sign into Tower and own the content they create.), TEAMS (Split up your organization to associate content and control permissions for groups.), CREDENTIALS (Add passwords, SSH keys, and other credentials to use when launching jobs against machines, or when syncing inventories or projects.), CREDENTIAL TYPES (Create custom credential types to be used for authenticating to network hosts and cloud sources.), MANAGEMENT JOBS (Manage the cleanup of old job history, activity streams, data marked for deletion, and system tracking info.), INVENTORY SCRIPTS (Create and edit scripts to dynamically load hosts from any source.), NOTIFICATIONS (Create templates for sending notifications with Email, HipChat, Slack, and SMS.), INSTANCE GROUPS (View list and capacity of Tower instances.), CONFIGURE TOWER (Edit Tower's configuration.), ABOUT TOWER (View information about this version of Ansible Tower.), and VIEW YOUR LICENSE (View and edit your license information.).

As you can see, there are a lot of different options here. As you may have already guessed, the option we are interested in is the CREDENTIALS one. Clicking on it will take you to a page that gives you an overview of the existing credentials; we want to add some new ones, so click on the +ADD button.

This should take you to a page that is similar in layout to the one where we added the project. Fill out the following information:

- **NAME:** AWS API Credentials
- **DESCRIPTION:** AWS API Credentials
- **ORGANIZATION:** Default
- **CREDENTIAL TYPE:** Click on the magnifying glass icon and select Amazon Web Services

Once the CREDENTIAL TYPE has been selected, the second section will be added; here, you can enter the following:

- **ACCESS KEY:** Add your access key from the previous AWS chapters, for example, AKIAI5KECPOTNTTVM3EDA
- **SECRET KEY:** Add your secret key from the previous AWS chapters, for example, Y4B7FFiSW10Am3VIFc071gnc/TAtK5+RpxZIGTr
- **STS TOKEN:** Leave blank

Once the form is completed, click on SAVE. Once saved, you will notice that the SECRET KEY is marked as ENCRYPTED:

The screenshot shows the 'Edit Credential' dialog for an 'AWS API Credentials' type. The 'DETAILS' tab is active. In the 'NAME' field, 'AWS API Credentials' is entered. In the 'DESCRIPTION' field, 'AWS API Credentials' is also entered. Under 'ORGANIZATION', 'Default' is selected. In the 'CREDENTIAL TYPE' section, a search bar shows 'Amazon Web Services'. The 'TYPE DETAILS' section contains fields for 'ACCESS KEY' (with a redacted value), 'SECRET KEY' (with a redacted value and a note 'REPLACE ENCRYPTED'), and 'STS TOKEN' (with a redacted value). At the bottom right are 'CANCEL' and 'SAVE' buttons.

When you save sensitive information in Ansible Tower, it is encrypted, and you only have the option to REPLACE or REVERT it. At no point can you view the information again.

Adding an inventory

Now that we have the credentials in place, we need to recreate the content of the inventory file called `production` within Ansible Tower. As a reminder, the file looks like the following:

```
# Register all of the host groups we will be creating in the playbooks
[ec2_instance]
[already_running]

# Put all the groups into into a single group so we can easily apply one config to it
# for overriding things like the ssh user and key location
[aws:children]
ec2_instance
already_running

# Finally, configure some bits to allow us access to the instances before we deploy our
# credentials using Ansible
[aws:vars]
ansible_ssh_user=centos
ansible_ssh_private_key_file=~/ssh/id_rsa
host_key_checking=False
```

To add the inventory, click on INVENTORIES in the top menu and then the +ADD button. You will notice that the +ADD button now brings up a drop-down list; from that list, we want to add an Inventory.

In the form that opens, enter the following:

- NAME: AWS Inventory
- DESCRIPTION: AWS Inventory
- ORGANIZATION: Default
- INSIGHTS CREDENTIALS: Leave blank
- INSIGHTS GROUPS: Leave blank
- VARIABLES: Enter the values listed as follows:

```
ansible_ssh_user: "centos"
ansible_ssh_private_key_file: "~/ssh/id_rsa"
host_key_checking: "False"
```

Once entered, click on SAVE; this will create the Inventory, and we can now add the two groups we need. To do this, click on GROUPS, which can be found in the row on the buttons above the form:

The screenshot shows the Ansible Tower web interface. At the top, there are navigation links for TOWER, PROJECTS, INVENTORIES, TEMPLATES, and JOBS. On the right, there are user account icons for 'admin' and various system settings. Below the header, the URL bar shows '10.20.30.40.nip.io'. The main content area is titled 'INVENTORIES / AWS Inventory'. A sub-section titled 'AWS Inventory' is displayed. It includes tabs for DETAILS (which is selected), PERMISSIONS, GROUPS, HOSTS, SOURCES, and COMPLETED JOBS. The 'NAME' field is set to 'AWS Inventory', the 'DESCRIPTION' field is 'AWS Inventory', and the 'ORGANIZATION' field is 'Default'. Under 'INSIGHTS CREDENTIAL' and 'INSTANCE GROUPS', there are search input fields. The 'VARIABLES' section contains a code editor with the following YAML configuration:

```
1 ansible_ssh_user: "centos"
2 ansible_ssh_private_key_file: "~/.ssh/id_rsa"
3 host_key_checking: "False"
```

At the bottom right of the form are 'CANCEL' and 'SAVE' buttons.

Click on **+ADD GROUP** and then enter the following details:

- **NAME:** ec2_instance
- **DESCRIPTION:** ec2_instance
- **VARIABLES:** Leave blank

Then click on **SAVE**, repeat the process, and add a second group using the following details:

- NAME: already_running
- DESCRIPTION: already_running
- VARIABLES: Leave blank

Again, click on SAVE; you should now have two groups listed:

The screenshot shows the Ansible Tower interface for managing AWS inventories. At the top, there are navigation icons (refresh, back, forward, search), a URL bar (10.20.30.40.nip.io), and user authentication (admin). Below the header, there are links for TOWER, PROJECTS, INVENTORIES, TEMPLATES, and JOBS. On the right, there are additional icons for settings, logs, and help.

The main content area is titled "INVENTORIES / AWS Inventory / GROUPS". It displays a table titled "AWS Inventory" with the following columns: DETAILS, PERMISSIONS, GROUPS (which is selected and highlighted in dark grey), HOSTS, SOURCES, and COMPLETED JOBS. There is also a "RUN COMMANDS" button and a green "+ ADD GROUP" button.

The table lists two groups:

	GROUPS	ACTIONS
<input type="checkbox"/>	already_running	
<input type="checkbox"/>	ec2_instance	

At the bottom right of the table, it says "ITEMS 1-2".

Now that we have our project, inventory, and some credentials for accessing our AWS, we just need to add the templates, one to launch and configure the cluster, and another to terminate it.

Adding the templates

Click on TEMPLATES in the top menu and, in the drop-down menu of the +ADD button, select Job Template. This is the biggest form we have come across so far; however, parts of it will be populated automatically when we start to fill in the details. Let's make a start:

- NAME: AWS - Launch
- DESCRIPTION: Launch and deploy WordPress instances
- JOB TYPE: Leave as Run
- INVENTORY: Click on the icon and select AWS Inventory
- PROJECT: Click on the icon and select AWS Project
- PLAYBOOK: Select `site.yml` from the drop-down list
- CREDENTIAL: Select the CREDENTIAL TYPE of Amazon Web Services, then choose AWS API Credentials; also select the Demo Credential for MACHINE
- FORKS: Leave as Default
- LIMIT: Leave blank
- VERBOSITY: Leave at 0 (Normal)
- INSTANCE GROUPS, JOB TAGS, SKIP TAGS, LABELS: Leave blank
- SHOW CHANGES: Leave OFF
- OPTIONS and EXTRA VARIABLES: Leave as their defaults

Click on SAVE and you can add a second template to remove the cluster. To do this, click on the +ADD button and select Job Template again; this time, use the following information:

- NAME: AWS - Remove
- DESCRIPTION: Remove the WordPress cluster
- JOB TYPE: Leave as Run
- INVENTORY: Click on the icon and select AWS Inventory
- PROJECT: Click on the icon and select AWS Project
- PLAYBOOK: Select `remove.yml` from the drop-down list
- CREDENTIAL: Select the CREDENTIAL TYPE of Amazon Web

Services, then choose AWS API Credentials; also select the Demo Credential for MACHINE

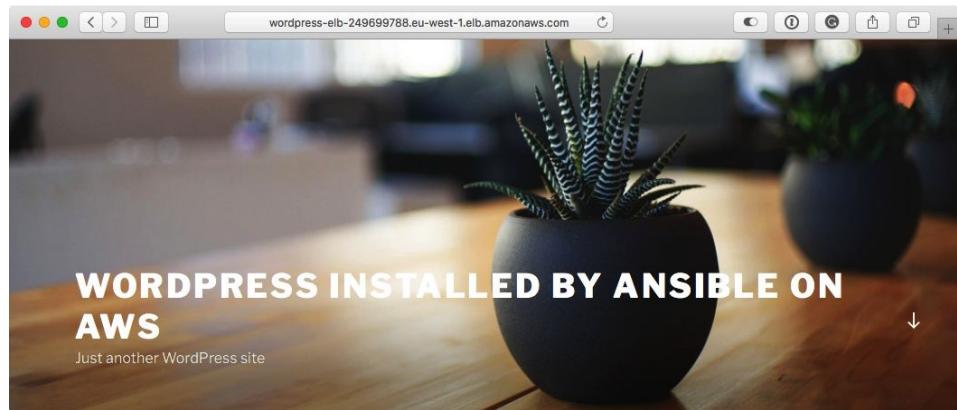
- FORKS: Leave as Default
- LIMIT: Leave blank
- VERBOSITY: Leave at 0 (Normal)
- INSTANCE GROUPS, JOB TAGS, SKIP TAGS, LABELS: Leave blank
- SHOW CHANGES: Leave OFF
- OPTIONS and EXTRA VARIABLES: Leave as their defaults

Running the playbook

Now that we have our playbook ready to run, we can run it by clicking on TEMPLATES in the top menu and then clicking on the run icon, the rocket one, next to AWS -Launch. This will take the same amount of time to run as it did when we executed it from the command line:

The screenshot shows the Tower UI interface. At the top, there's a navigation bar with icons for home, projects, inventories, templates, and jobs. The user is logged in as 'admin'. Below the navigation, a breadcrumb trail indicates the current location: JOBS / 12 - AWS - Launch. On the left, a 'DETAILS' panel displays the job configuration, including status (Successful), start and finish times (3/6/2018 11:32:07 and 3/6/2018 12:11:37), template (AWS - Launch), job type (Run), launched by (admin), inventory (AWS Inventory), project (AWS Project), revision (2c54f2e), playbook (site.yml), machine credential (Demo Credential), extra credentials (AWS API Credentials), forks (0), verbosity (0 (Normal)), and instance group (tower). On the right, the main pane is titled 'AWS - Launch' and shows the play history. It includes a search bar, a key button, and a list of tasks with their IDs (269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279) and log output. The log output includes messages like 'TASK [roles/autoscaling : create / update the auto-scaling group using the launch configuration we just created] ***', 'changed: [localhost]', 'TASK [roles/autoscaling : remove any tmp instances which are running] *****', 'changed: [localhost]', 'PLAY RECAP *****', 'ok=32 changed=25', and 'localhost : ok=47 changed=19 unreachable=0 failed=0'. At the bottom right of the main pane, there's a 'TOP' link. A copyright notice at the bottom right of the page reads 'Copyright © 2017 Red Hat, Inc.'

As you can see from the preceding screenshot, everything built and ran as expected, meaning that we will be able to see our WordPress site when going to the Elastic Load Balancer URL:



Removing the cluster

Now that we have launched the cluster, we can run the second playbook, which removes it. To do this, click on TEMPLATES in the top menu and then click on the run icon, the rocket one, next to AWS - Remove. This will launch the playbook, which removes everything we just launched. Again, it will take a little while to run through all of the tasks.

It is important to point out that in order for the `remove.yml` playbook to successfully execute through Ansible Tower, you have to update one of the tasks in `roles/remove/tasks/main.yml`. If you recall, we had the following lines in there:

```
- name: prompt
  pause:
    prompt: "Make sure the elastic load balancer has been terminated before proceeding"
```

If this task was present, then our playbook execution would stall at this task and not proceed, as Ansible Tower playbook runs are not interactive. The task was replaced with the following:

```
| - name: wait for 2 minutes before continuing
|   pause:
|     minutes: 2
```

This is the only change that was needed for our playbook to run using Ansible Tower; everything else remains as is.

Tower summary

While we have only had time to run a basic playbook, I am sure you can start to see the advantages of using Ansible Tower for enabling all of your users to run playbooks. There are quite a few features you can use. However, there are three different versions of Ansible Tower currently available. The following table provides a quick overview of the features available in each version:

Feature	Self-Support	Standard	Premium
Dashboard: Get an overview of your Ansible Tower status	Yes	Yes	Yes
Real-time job output: View the output of your jobs as they are executed in real time	Yes	Yes	Yes
Job scheduling: Execute jobs on a schedule; also set up repeat runs, for example, every weekday at 9 a.m., run the job to deploy the development instances	Yes	Yes	Yes
Pull from source control: Host your playbooks in source control, such as Git or SVN	Yes	Yes	Yes
Workflows: Chain multiple playbooks together in one job	No	Yes	Yes
Role-based access: Get fine-grained control of your users and what they can access	Yes	Yes	Yes
Integration with third-party authentication: Hook your Tower installation into an Active Directory or an LDAP identity server	No	Yes	Yes

Surveys: Build forms for users to fill out as part of a job run; this allows your users to provide information without them having to write any YAML	No	Yes	Yes
8x5 Support from Red Hat	No	Yes	Yes
24x7 Support from Red Hat	No	No	Yes

The current license costs for Ansible Tower are as follows:

- **Self-support up to 10 nodes:** Free of charge; this is the license we applied to our installation
- **Self-support up to 100 nodes:** \$5,000 per year
- **Self-support up to 250 nodes:** \$10,000 per year
- **Standard up to 100 nodes:** \$10,000 per year
- **Standard over 100 nodes:** Custom pricing, contact Ansible
- **Premium up to 100 nodes:** \$14,000 per year
- **Premium over 100 nodes:** Custom pricing, contact Ansible

These prices do not include the Red Hat-supported Ansible Engine; there are additional costs for that on top of the ones listed here, if you want a supported Ansible engine.

So, Ansible Tower, while great, may not be within everyone's budget and this is where Ansible AWX comes in.

Ansible AWX

Let's get straight into installing Ansible AWX; we will need a Vagrant box, Docker installed on the Vagrant box, and finally a copy of the AWX source.

Preparing the playbook

For our installation, we will be using Ansible to prepare our Vagrant box and install Ansible AWX. To create the structure for the playbook, run the following commands:

```
$ mkdir awx awx/group_vars awx/roles  
$ touch awx/production awx/site.yml awx/group_vars/common.yml awx/Vagrantfile
```

The `Vagrantfile` we are going to be using can be found [here](#):

```
# -*- mode: ruby -*-  
# vi: set ft=ruby :  
  
API_VERSION = "2"  
BOX_NAME = "centos/7"  
BOX_IP = "10.20.30.50"  
DOMAIN = "nip.io"  
PRIVATE_KEY = "~/.ssh/id_rsa"  
PUBLIC_KEY = '~/.ssh/id_rsa.pub'  
  
Vagrant.configure(API_VERSION) do |config|  
  config.vm.box = BOX_NAME  
  config.vm.network "private_network", ip: BOX_IP  
  config.vm.host_name = BOX_IP + '.' + DOMAIN  
  config.ssh.insert_key = false  
  config.ssh.private_key_path = [PRIVATE_KEY, "~/.vagrant.d/insecure_private_key"]  
  config.vm.provision "file", source: PUBLIC_KEY, destination: "~/.ssh/authorized_keys"  
  
  config.vm.provider "virtualbox" do |v|  
    v.memory = "2024"  
    v.cpus = "2"  
  end  
  
  config.vm.provider "vmware_fusion" do |v|  
    v.vmx["memsize"] = "2024"  
    v.vmx["numvcpus"] = "2"  
  end  
end
```

The first role we are going to create is one we have covered already; it is the Docker role from [Chapter 14, Deploying WPScan and OWASP ZAP](#).

The docker role

I am not going to go into detail about the tasks, as these have already been covered. We can bootstrap the role by running the following command:

```
| $ ansible-galaxy init roles/docker
```

Now that we have the files in place, we can update the `roles/docker/defaults/main.yml` file with the following content:

```
docker:
  gpg_key: "https://download.docker.com/linux/centos/gpg"
  repo_url: "https://download.docker.com/linux/centos/docker-ce.repo"
  repo_path: "/etc/yum.repos.d/docker-ce.repo"
  packages:
    - "docker-ce"
    - "device-mapper-persistent-data"
    - "lvm2"
    - "python-setuptools"
    - "libselinux-python"
  pip:
    - "docker"
```

The content of `roles/docker/tasks/main.yml` should be:

```
- name: update all of the installed packages
  yum:
    name: "*"
    state: "latest"
    update_cache: "yes"

- name: add the gpg key for the docker repo
  rpm_key:
    key: "{{ docker.gpg_key }}"
    state: "present"

- name: add docker repo from the remote url
  get_url:
    url: "{{ docker.repo_url }}"
    dest: "{{ docker.repo_path }}"
    mode: "0644"

- name: install the docker packages
  yum:
    name: "{{ item }}"
    state: "installed"
    update_cache: "yes"
  with_items: "{{ docker.packages }}"

- name: install pip
  easy_install:
    name: pip
    state: latest
```

```
- name: install the python packages
  pip:
    name: "{{ item }}"
  with_items: "{{ docker.pip }}"

- name: put selinux into permissive mode
  selinux:
    policy: targeted
    state: permissive

- name: start docker and configure to start on boot
  service:
    name: "docker"
    state: "started"
    enabled: "yes"
```

This should get the Docker part of the AWX installation installed and allow us to move onto the next role.

The awx role

The next and (sort of) final role for our AWX installation can be created by running:

```
| $ ansible-galaxy init roles/awx
```

The default variables in `roles/awx/defaults/main.yml` are similar in format to the ones in the `docker` role:

```
awx:
  repo_url: "https://github.com/ansible/awx.git"
  logo_url: "https://github.com/ansible/awx-logos.git"
  repo_path: "~/awx/"
  packages:
    - "git"
  pip:
    - "ansible"
    - "boto"
    - "boto3"
    - "botocore"
  install_command: 'ansible-playbook -i inventory --extra-vars "awx_official=true"'
install.yml'
```

Starting from the top, we have two different GitHub repo URLs. The first `awx.repo_url` is the main AWX repository and the second `awx.logo_url` is for the official logo pack. Next up, we have the path, `awx.repo_path`, and we want to check out the code too. In this case, it is `~/awx` which, as we are using `become`, will be `/root/awx/`.

To check out the code from GitHub, we need to make sure that we have Git installed. `awx.packages` is the only additional package we need to install using `yum`. Next up, we need to install Ansible itself and a few of the other Python packages we will need using PIP (`awx.pip`).

Finally, we have the command (`awx.install_command`) we need to run in order to install Ansible AWX. As you can see, we are using an Ansible playbook that ships as part of the code we are checking out; the command itself is overriding the option for using official AWX logos by passing `awx_official=true` as extra variables.

Now that we have discussed the variables we need to define, we can add the

tasks to `roles/awx/tasks/main.yml`, starting with the tasks that install the Yum and Pip packages:

```
- name: install the awx packages
  yum:
    name: "{{ item }}"
    state: "installed"
    update_cache: "yes"
    with_items: "{{ awx.packages }}"

- name: install the python packages
  pip:
    name: "{{ item }}"
    with_items: "{{ awx.pip }}"
```

Next, we have the tasks that check out the two AWX repositories from GitHub:

```
- name: check out the awx repo
  git:
    repo: "{{ awx.repo_url }}"
    dest: "{{ awx.repo_path }}"
    clone: "yes"
    update: "yes"

- name: check out the awx logos repo
  git:
    repo: "{{ awx.logo_url }}"
    dest: "{{ awx.repo_path }}"
    clone: "yes"
    update: "yes"
```

As you can see, both repositories are being moved to the same location on our Vagrant box. The final task runs the playbook that downloads, configures, and launches the Ansible AWX Docker containers:

```
- name: install awx
  command: "{{ awx.install_command }}"
  args:
    chdir: "{{ awx.repo_path }}installer"
```

Running the playbook

Now that we have our playbook in place, we can add our host inventory information to the `production` file:

```
box ansible_host=10.20.30.50.nip.io
[awx]
box

[awx:vars]
ansible_connection=ssh
ansible_user=vagrant
ansible_private_key_file=~/ssh/id_rsa
host_key_checking=False
```

Finally, we can add the following to the `site.yml` file, and we should be good to run our installation:

```
---
- hosts: awx
  gather_facts: true
  become: yes
  become_method: sudo

  vars_files:
    - group_vars/common.yml

  roles:
    - roles/docker
    - roles/awx
```

To get Ansible AWX up and running, we need to execute one of the following commands to launch the Vagrant box:

```
$ vagrant up
$ vagrant up --provider=vmware_fusion
```

Then, the following command will run the playbook:

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

It will take a few minutes to run through the playbook; once complete, you should see something like this:

```
1. awx (bash)
TASK [roles/awx : install the python packages] ****
changed: [box] => (item=ansible)
changed: [box] => (item=boto)
changed: [box] => (item=boto3)
ok: [box] => (item=botocore)

TASK [roles/awx : check out the awx repo] ****
changed: [box]

TASK [roles/awx : check out the awx logos repo] ****
changed: [box]

TASK [roles/awx : install awx] ****
changed: [box]

PLAY RECAP ****
box : ok=14    changed=13    unreachable=0    failed=0
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter15/awx on master*
```

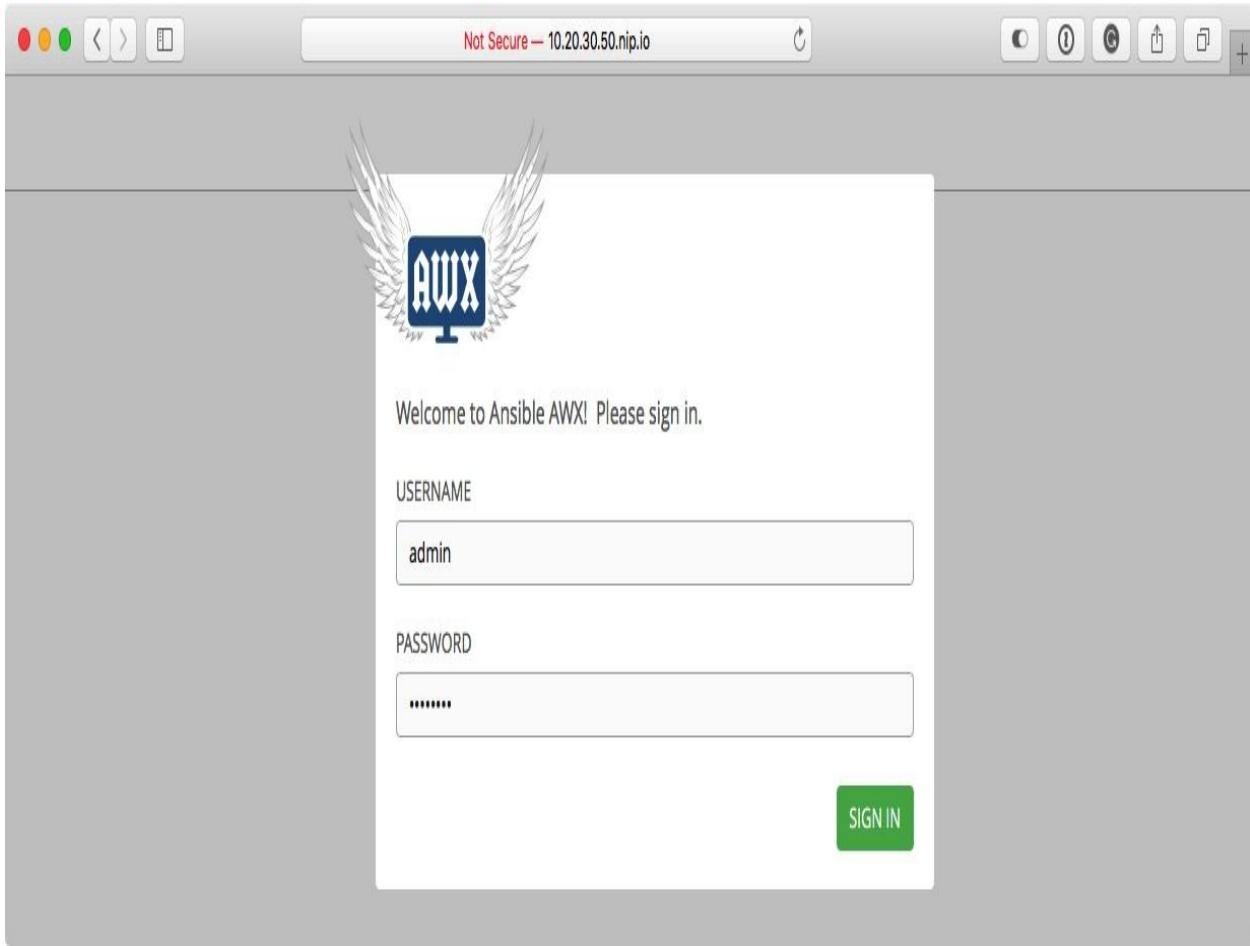
Opening your browser and going to <http://10.20.30.50.nip.io/> should show you the following message:



Keep the page open and, after a few minutes, you should see a login prompt.

Using Ansible AWX

You should be at a login prompt. The USERNAME and PASSWORD are admin/password:



When you first log in, you may notice that the look and feel is similar to Ansible Tower, although, there are a few differences:

The screenshot shows the AWX dashboard interface. On the left, a dark sidebar contains navigation links for DASHBOARD, JOBS, SCHEDULES, PORTAL MODE, RESOURCES, TEMPLATES, CREDENTIALS, PROJECTS, INVENTORIES, INVENTORY SCRIPTS, ORGANIZATIONS, USERS, TEAMS, and ADMINISTRATION. The DASHBOARD link is currently selected. At the top right, there is a user icon labeled "admin" and other navigation icons. The main content area is titled "DASHBOARD". It features a row of six cards with summary statistics: 1 HOSTS, 0 FAILED HOSTS, 1 INVENTORIES, 0 INVENTORY SYNC FAILURES, 1 PROJECTS, and 0 PROJECT SYNC FAILURES. Below this is a section titled "JOB STATUS" with a line chart. The chart has "TIME" on the x-axis (from May 3 to Jun 3) and "SVM" on the y-axis (-1 to 1). A single data point at May 3 is at a value of 1. Above the chart are filters for "PERIOD" (set to "PAST MONTH"), "JOB TYPE" (set to "ALL"), and "VIEW" (set to "ALL"). At the bottom, there are two boxes: "RECENTLY USED JOB TEMPLATES" (text: "No job templates were recently used. You can create a job template here.") and "RECENTLY RUN JOBS" (text: "No jobs were recently run.").

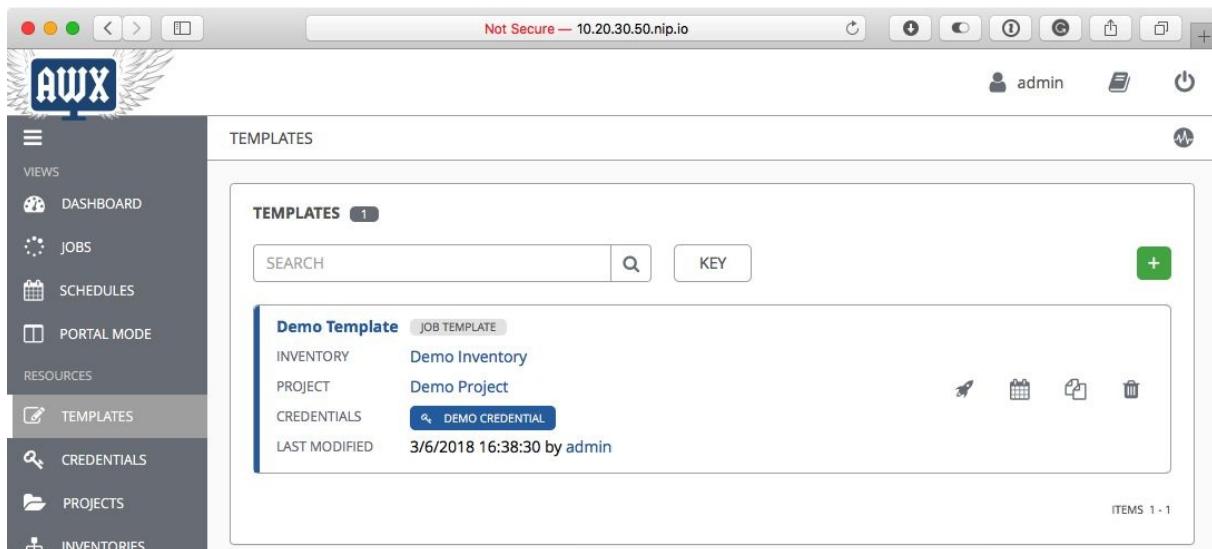
As you can see, the menu has moved from the top to the left-hand side and there are also more options. Clicking on PROJECTS in the left-hand menu will take you to the page where you can Get The Latest SVM Revision of the hello-world example we first ran in Ansible Tower. Click the cloud icon to download it.

Once you have the project synced, click on **TEMPLATES** in the left-hand menu; you should see an empty list. Click on the **+** button and select **Job Template** from the drop-down list.

This will take you to a page that is the same as we saw when adding a template in Ansible Tower. Fill in the following details:

- **NAME:** Demo Template
- **DESCRIPTION:** Run the hello-world example
- **JOB TYPE:** Leave as Run
- **INVENTORY:** Click on the icon and select Demo Inventory
- **PROJECT:** Click on the icon and select Demo Project
- **PLAYBOOK:** Select hello-world.yml from the drop-down list
- **CREDENTIAL:** Click on the icon and select the **Demo Credential** from the list
- **FORKS:** Leave as Default
- **LIMIT:** Leave blank
- **VERBOSITY:** Leave at 0 (Normal)
- **INSTANCE GROUPS, JOB TAGS, SKIP TAGS, LABELS:** Leave blank
- **SHOW CHANGES:** Leave OFF
- **OPTIONS and EXTRA VARIABLES:** Leave as their defaults

Once filled in, click on the **SAVE** button at the bottom of the form. Clicking on **TEMPLATES** in the left-hand side menu will now show the **Demo Template** listed:



The screenshot shows the AWX web interface. The left sidebar has a dark theme with icons for Views, Dashboard, Jobs, Schedules, Portal Mode, Resources, Templates (which is selected and highlighted in blue), Credentials, Projects, and Inventories. The main content area has a light background. At the top, it says "TEMPLATES". Below that is a search bar with a magnifying glass icon and a "KEY" button. A green "+" button is on the right. A table lists one item: "Demo Template" (JOB TEMPLATE). The table rows show: INVENTORY (Demo Inventory), PROJECT (Demo Project), CREDENTIALS (DEMO CREDENTIAL), and LAST MODIFIED (3/6/2018 16:38:30 by admin). To the right of the table are icons for edit, calendar, copy, and delete. At the bottom right of the table area, it says "ITEMS 1 - 1". The top of the browser window shows "Not Secure — 10.20.30.50.nip.io" and the status bar shows "admin".

Clicking on the rocket icon, or **Start a job using this template**, will run the

hello world playbook:

The screenshot shows the Ansible AWX web interface. On the left is a dark sidebar with various navigation links: DASHBOARD, JOBS, SCHEDULES, PORTAL MODE, TEMPLATES, CREDENTIALS, PROJECTS, INVENTORIES, INVENTORY SCRIPTS, ORGANIZATIONS, USERS, TEAMS, CREDENTIAL TYPES, and NOTIFICATIONS. The main content area has a header "JOBS / 2 - Demo Template". On the left within this area is a "DETAILS" panel showing the job status as "Successful", started at "3/6/2018 16:41:19", finished at "3/6/2018 16:41:20", and using the "Demo Template" job template. It also lists the job type as "Run", launched by "admin", and uses the "Demo Inventory" and "Demo Project". The "PLAYBOOK" listed is "hello_world.yml". Below this, under "EXTRA VARIABLES", there is a YAML editor with one entry: "1 ---". To the right is the log output for the "Demo Template" job, titled "Demo Template". The log shows the following steps:

```
PLAY [Hello World Sample] *****
*****
2 TASK [Gathering Facts] *****
*****
3 ok: [localhost]
4
5 ok: [localhost] => {
6   "msg": "Hello World!"
7 }
8
9
10
11
12 PLAY RECAP *****
*****
13 localhost : ok=2    changed=0      unreachable=0 failed=0
14
```

At the bottom right of the main content area, it says "About | Copyright © 2018 Red Hat, Inc."

So there we have it—a very quick overview of Ansible AWX which, as I have already mentioned, is not too dissimilar to Ansible Tower.

AWX summary

Let's get this out of the way now. At the time of writing, Red Hat does not recommend Ansible AWX for production environments. Personally, I have found it to be quite stable, especially for software that is in constant development. Sure, there have been some issues when upgrading but, for the most part, these have been minimal.

With Ansible AWX being the upstream of Ansible Tower, there are features present, such as being able to use third-party authentication services and workflows, which are not present in the self-supported version of Ansible Tower. There are also no limits on the number of hosts you can manage. This makes Ansible AWX a very attractive alternative to Ansible Tower; however, you do need to take into account its development cycle and how upgrades could affect the day-to-day running of your AWX installation.

Summary

In this chapter, we have worked through the installation and usage of two different web frontends that can be used to run your Ansible playbooks. We have also discussed the differences in cost, functionality, and stability between the various versions of the frontends.

I am sure you will agree that using a tool such as Ansible Tower or Ansible AWX will allow your users, colleagues, and end users to consume the playbooks you write in a supported and consistent way.

In the next chapter, we are going to look at the `ansible-galaxy` command and services in more detail.

Questions

1. State and explain the differences between Ansible Tower and Ansible AWX.
2. Using Ansible AWX, configure and run the AWS WordPress playbook as we did with Ansible Tower.

Further reading

For more details on the two pieces of software, see the following URLs:

- **Ansible Tower overview:** <https://www.ansible.com/products/tower/>
- **Ansible Tower full feature list:** <https://www.ansible.com/products/tower/editions/>
- **Ansible AWX announcement:** <https://www.redhat.com/en/about/press-releases/red-hat-advances-enterprise-and-network-automation-new-ansible-offerings/>
- **Ansible AWX FAQ:** <https://www.ansible.com/products/awx-project/faq/>
- **Ansible AWX GitHub repository:** <https://github.com/ansible/awx/>

Ansible Galaxy

We have been using the `ansible-galaxy` command throughout the previous chapters. In this chapter, we are going to look at more of the features provided by the command. Ansible Galaxy is an online repository of community contributed roles; we will discover some of the best roles available, how to use them, and how to create your own role and have it hosted on Ansible Galaxy.

By the end of the chapter, we will have worked through the following:

- An introduction to Ansible Galaxy
- How to use roles from Ansible Galaxy in your own playbooks
- How to write and submit your own roles to Ansible Galaxy

Technical requirements

Again, we will be using a local Vagrant box for this chapter; the playbooks used can be found in the accompanying repository at <https://github.com/PacktPublishing/Learn-Ansible/tree/master/Chapter16>. You will also need access to a GitHub account—a free account will do—and you can sign up for one at <http://github.com/>.

Introduction to Ansible Galaxy

Ansible Galaxy is a number of things: first and foremost, it is a website that can be found at <https://galaxy.ansible.com/>. The website is home to community contributed roles and modules:

The screenshot shows the Ansible Galaxy homepage with a dark background featuring a galaxy image. At the top, there's a navigation bar with links for ABOUT, EXPLORE, SEARCH, BROWSE AUTHORS, and SIGN IN. Below the navigation, a large text area says "Galaxy is your hub for finding, reusing and sharing Ansible content". To the right, there are two sign-in options: "Signin with GitHub" and "Use an existing account not associated with GitHub". At the bottom, there are three main sections: "DOWNLOAD", "SHARE", and "FEATURED". The "DOWNLOAD" section has a "Cloud" icon and text about jump-starting automation projects. The "SHARE" section has a "Share" icon and text about sharing roles with the community. The "FEATURED" section is highlighted in red and lists a featured role by "carlosbuenosvino", an author by "andrewrothstein", and a blog section.

DOWNLOAD

Jump-start your automation project with great content from the Ansible community. Galaxy provides pre-packaged units of work known to Ansible as [roles](#).

Roles can be dropped into Ansible PlayBooks and immediately put to work. You'll find roles for provisioning infrastructure, deploying applications, and all of the tasks you do everyday.

Use [Search](#) to find roles for your project, then download them onto your Ansible host using the [ansible-galaxy](#) command that comes bundled with Ansible.

SHARE

Help other Ansible users by sharing the awesome roles you create.

Maybe you have a role for installing and configuring a popular software package, or a role for deploying software built by your company. Whatever it is, use Galaxy to share it with the community.

Top content authors will be featured on the [Explore](#) page, achieving worldwide fame! Or at least fame on the internet among other developers and sysadmins.

FEATURED

ROLE: [carlosbuenosvino.ansistrano-deploy](#) - Ansible role to deploy scripting applications like PHP, Python, Ruby, etc. in a Capistrano style

< ● ○ ○ >

AUTHOR: [andrewrothstein](#) with 248 roles.

< ● ○ ○ >

BLOG: Read the latest from [The Inside Playbook](#), and keep up with what's happening in the Ansible universe.

So far, we have been writing our own roles that interact with the Ansible Core

modules for use in our playbook. Rather than writing our own roles, we could be using one of the more than 15,000 roles published on Ansible Galaxy—these roles cover a multitude of tasks and support pretty much of all of the operating systems supported by Ansible.

The `ansible-galaxy` command is a way of interacting with the Ansible Galaxy website from the comfort of your own command line, as well as being able to bootstrap roles. Just as we have been using it in previous chapters, we can also use it to download, search and publish our own roles on Ansible Galaxy.

Finally, Red Hat has open sourced the code for Ansible Galaxy, meaning that you can also run your own version of the site should you need to distribute your own roles behind a firewall.

Jenkins playbook

Let's dive straight in and create a playbook that installs Jenkins using just roles downloaded from Ansible Galaxy.



Jenkins, formerly the Hudson project, is an open source continuous integration and continuous delivery server written in Java. It is expandable using plugins and has grown much bigger than its original purpose of compiling Java applications.

To start, we are going to need a few files; let's get these created now by running the following:

```
$ mkdir jenkins  
$ cd jenkins  
$ touch production requirements.yml site.yml Vagrantfile
```

As you can see, we are not creating a `roles` or `group_vars` folder as we have been doing in previous chapters. Instead, we are creating a `requirements.yml` file. This will contain a list of the roles we would like to download from Ansible Galaxy.

In our case, we are going to be using the following two roles:

- **Java:** <https://galaxy.ansible.com/geerlingguy/java/>
- **Jenkins:** <https://galaxy.ansible.com/geerlingguy/jenkins/>

The first of the roles, `geerlingguy.java`, manages the installation of Java on our host and then the second, `geerlingguy.jenkins`, manages the installation and configuration of Jenkins itself. To install the roles, we need to add the following lines to our `requirements.yml` file:

```
- src: "geerlingguy.java"  
- src: "geerlingguy.jenkins"
```

Once added, we can download the roles by running the following command:

```
| $ ansible-galaxy install -r requirements.yml
```

You should see something like the following output:

```
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter16/jenkins on master*
└── ansible-galaxy install -r requirements.yml
  - downloading role 'java', owned by geerlingguy
  - downloading role from https://github.com/geerlingguy/ansible-role-java/archive/1.8.1.tar.gz
  - extracting geerlingguy.java to /Users/russ/.ansible/roles/geerlingguy.java
  - geerlingguy.java (1.8.1) was installed successfully
  - downloading role 'jenkins', owned by geerlingguy
  - downloading role from https://github.com/geerlingguy/ansible-role-jenkins/archive/3.5.0.tar.gz
  - extracting geerlingguy.jenkins to /Users/russ/.ansible/roles/geerlingguy.jenkins
  - geerlingguy.jenkins (3.5.0) was installed successfully
  [WARNING]: - dependency geerlingguy.java from role geerlingguy.jenkins differs from already
  installed version (1.8.1), skipping
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter16/jenkins on master*
```

As you can see from the Terminal output, the two roles have been downloaded from the `roles` folder of GitHub project and placed in the `~/.ansible/roles/` folder.



Using `~` in a path on macOS and Linux is shorthand for current users' home directory.

You can ignore the warning; it is just letting us know that the `geerlingguy.jenkins` role wanted to install an older version of the `geerlingguy.java` role. In our case, this is not going to cause any problems.

Now that we have our two roles downloaded, we can write the `site.yml` file for us to launch Jenkins. This should look like the following:

```
---
- hosts: jenkins
  gather_facts: true
  become: yes
  become_method: sudo

  vars:
    java_packages: "java-1.8.0-openjdk"
    jenkins_hostname: "10.20.30.60.nip.io"
    jenkins_admin_username: "ansible"
    jenkins_admin_password: "Pa55w0rD"

  roles:
    - geerlingguy.java
    - geerlingguy.jenkins
```

Notice that we are just providing the names of the roles. Ansible, by default, will search for roles in the `~/.ansible/roles/` folder if they are not found in a `roles` folder local to your playbook.

We are also passing four variables:

- `java_packages`: This is the name of the `geerlingguy.java` role we want the role to

install; as Jenkins requires Java 8, and we are running a CentOS 7 host, the package name is `java-1.8.0-openjdk`.

The remaining three variables affect what the `geerlingguy.jenkins` role configures:

- `jenkins_hostname`: This is the URL we want to access Jenkins on; as in previous chapters, we are using the `nip.io` service to provide a resolvable hostname for our Vagrant box
- `jenkins_admin_username`: This is the admin username we want to configure for accessing Jenkins
- `jenkins_admin_password`: This is the password for the user

Next up, we have the `production` host's inventory file:

```
box ansible_host=10.20.30.60.nip.io

[jenkins]
box

[jenkins:vars]
ansible_connection=ssh
ansible_user=vagrant
ansible_private_key_file=~/ssh/id_rsa
host_key_checking=False
```

Finally, the content of the `Vagrantfile` is as follows:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

API_VERSION = "2"
BOX_NAME = "centos/7"
BOX_IP = "10.20.30.60"
DOMAIN = "nip.io"
PRIVATE_KEY = "~/ssh/id_rsa"
PUBLIC_KEY = '~/ssh/id_rsa.pub'

Vagrant.configure(API_VERSION) do |config|
  config.vm.box = BOX_NAME
  config.vm.network "private_network", ip: BOX_IP
  config.vm.host_name = BOX_IP + '.' + DOMAIN
  config.ssh.insert_key = false
  config.ssh.private_key_path = [PRIVATE_KEY, "~/.vagrant.d/insecure_private_key"]
  config.vm.provision "file", source: PUBLIC_KEY, destination: "~/ssh/authorized_keys"

  config.vm.provider "virtualbox" do |v|
    v.memory = "2024"
    v.cpus = "2"
  end

  config.vm.provider "vmware_fusion" do |v|
    v.vmx["memsize"] = "2024"
    v.vmx["numvcpus"] = "2"
  end
```

```
| end
```

Now that we have all of the files we need in place and populated with the right code, we can launch our Jenkins server. First, we need to create the Vagrant box:

```
$ vagrant up
$ vagrant up --provider=vmware_fusion
```

Once the Vagrant box is up and running, we can run the playbook using the following command:

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

It will take a few minutes to install and configure both Java and Jenkins; you can see the output of the playbook run here:

```
PLAY [jenkins]
*****
TASK [Gathering Facts]
*****
ok: [box]

TASK [geerlingguy.java : Include OS-specific variables.]
*****
ok: [box]

TASK [geerlingguy.java : Include OS-specific variables for Fedora.]
*****
skipping: [box]

TASK [geerlingguy.java : Include version-specific variables for Debian.]
*****
skipping: [box]

TASK [geerlingguy.java : Define java_packages.]
*****
skipping: [box]

TASK [geerlingguy.java : include_tasks]
*****
included: /Users/russ/.ansible/roles/geerlingguy.java/tasks/setup-RedHat.yml for box

TASK [geerlingguy.java : Ensure Java is installed.]
*****
changed: [box] => (item=java-1.8.0-openjdk)

TASK [geerlingguy.java : include_tasks]
*****
skipping: [box]

TASK [geerlingguy.java : include_tasks]
*****
skipping: [box]

TASK [geerlingguy.java : Set JAVA_HOME if configured.]
```

```
*****
skipping: [box]

TASK [geerlingguy.jenkins : Include OS-Specific variables]
*****
ok: [box]

TASK [geerlingguy.jenkins : Define jenkins_repo_url]
*****
ok: [box]

TASK [geerlingguy.jenkins : Define jenkins_repo_key_url]
*****
ok: [box]

TASK [geerlingguy.jenkins : Define jenkins_pkg_url]
*****
ok: [box]

TASK [geerlingguy.jenkins : include_tasks]
*****
included: /Users/russ/.ansible/roles/geerlingguy.jenkins/tasks/setup-RedHat.yml for box

TASK [geerlingguy.jenkins : Ensure dependencies are installed.]
*****
ok: [box]

TASK [geerlingguy.jenkins : Ensure Jenkins repo is installed.]
*****
changed: [box]

TASK [geerlingguy.jenkins : Add Jenkins repo GPG key.]
*****
changed: [box]

TASK [geerlingguy.jenkins : Download specific Jenkins version.]
*****
skipping: [box]

TASK [geerlingguy.jenkins : Check if we downloaded a specific version of Jenkins.]
*****
skipping: [box]

TASK [geerlingguy.jenkins : Install our specific version of Jenkins.]
*****
skipping: [box]

TASK [geerlingguy.jenkins : Ensure Jenkins is installed.]
*****
changed: [box]

TASK [geerlingguy.jenkins : include_tasks]
*****
skipping: [box]

TASK [geerlingguy.jenkins : include_tasks]
*****
included: /Users/russ/.ansible/roles/geerlingguy.jenkins/tasks/settings.yml for box

TASK [geerlingguy.jenkins : Modify variables in init file]
*****
changed: [box] => (item={u'option': u'JENKINS_ARGS', u'value': u'--prefix='})
changed: [box] => (item={u'option': u'JENKINS_JAVA_OPTIONS', u'value': u'-Djenkins.install.runSetupWizard=false'})
```

```
TASK [geerlingguy.jenkins : Set the Jenkins home directory]
*****
changed: [box]

TASK [geerlingguy.jenkins : Immediately restart Jenkins on init config changes.]
*****
changed: [box]

TASK [geerlingguy.jenkins : Set HTTP port in Jenkins config.]
*****
changed: [box]

TASK [geerlingguy.jenkins : Ensure jenkins_home /var/lib/jenkins exists]
*****
ok: [box]

TASK [geerlingguy.jenkins : Create custom init scripts directory.]
*****
changed: [box]

RUNNING HANDLER [geerlingguy.jenkins : configure default users]
*****
changed: [box]

TASK [geerlingguy.jenkins : Immediately restart Jenkins on http or user changes.]
*****
changed: [box]

TASK [geerlingguy.jenkins : Ensure Jenkins is started and runs on startup.]
*****
ok: [box]

TASK [geerlingguy.jenkins : Wait for Jenkins to start up before proceeding.]
*****
FAILED - RETRYING: Wait for Jenkins to start up before proceeding. (60 retries left).
[WARNING]: Consider using the get_url or uri module rather than running curl. If you
need to use
command because get_url or uri is insufficient you can add warn=False to this command
task or set
command_warnings=False in ansible.cfg to get rid of this message.

ok: [box]

TASK [geerlingguy.jenkins : Get the jenkins-cli jarfile from the Jenkins server.]
*****
changed: [box]

TASK [geerlingguy.jenkins : Remove Jenkins security init scripts after first startup.]
*****
changed: [box]

TASK [geerlingguy.jenkins : include_tasks]
*****
included: /Users/russ/.ansible/roles/geerlingguy.jenkins/tasks/plugins.yml for box

TASK [geerlingguy.jenkins : Get Jenkins admin password from file.]
*****
skipping: [box]

TASK [geerlingguy.jenkins : Set Jenkins admin password fact.]
*****
ok: [box]
```

```

TASK [geerlingguy.jenkins : Get Jenkins admin token from file.]
*****
skipping: [box]

TASK [geerlingguy.jenkins : Set Jenkins admin token fact.]
*****
ok: [box]

TASK [geerlingguy.jenkins : Create update directory]
*****
ok: [box]

TASK [geerlingguy.jenkins : Download current plugin updates from Jenkins update site]
*****
changed: [box]

TASK [geerlingguy.jenkins : Remove first and last line from json file]
*****
ok: [box]

TASK [geerlingguy.jenkins : Install Jenkins plugins using password.]
*****
TASK [geerlingguy.jenkins : Install Jenkins plugins using token.]
*****
```

PLAY RECAP

```
*****
box : ok=32 changed=14 unreachable=0 failed=0
```

Once the playbook has completed, you should be able to access your newly installed Jenkins at <http://10.20.30.60.nip.io:8080/> and use the admin username and password we defined in the `site.yml` file to log in:



As you can see, using a predefined community role to deploy our Jenkins installation was a lot more straightforward than writing our own role. In a few minutes, we were able to write a playbook and deploy the application with no more than a basic understanding of how to install the application. In fact, a quick skim through the readme file on Ansible Galaxy for the two roles was all that was required.

Publishing a role

Now we know how easy it is to download a role, let's look at how we can contribute back to the community by creating a role. In the last few chapters, we have been using Ansible to install Docker. So let's use that as our base and extend the role to support Ubuntu and have it install the Docker CE Edge release rather than the stable one.

Creating the docker role

To start off with, we need the basic files; to get these, run the following command where you normally store your code:

```
| $ ansible-galaxy init ansible-role-docker
```

This will give us the directory and file structure we need for our new role; we can now make a start on creating the role.

Variables

We are going to start with the files in the `vars` folder; we are going to be keeping the `vars/main.yml` file blank and adding two new files starting with `vars/RedHat.yml`: -

```
--  
# vars file for ansible-role-docker  
  
docker:  
  gpg_key: "https://download.docker.com/linux/centos/gpg"  
  repo_url: "https://download.docker.com/linux/centos/docker-ce.repo"  
  repo_path: "/etc/yum.repos.d/docker-ce.repo"  
  edge: "docker-ce-edge"  
  
  packages:  
    - "docker-ce"  
    - "device-mapper-persistent-data"  
    - "lvm2"  
    - "python-setuptools"  
    - "libselinux-python"  
  
  pip:  
    - "docker"
```

The next file to add is `vars/Debian.yml`:

```
--  
# vars file for ansible-role-docker  
  
docker:  
  gpg_key: "https://download.docker.com/linux/ubuntu/gpg"  
  repo: "deb [arch=amd64] https://download.docker.com/linux/{{ ansible_distribution |  
lower }} {{ ansible_distribution_release | lower }} edge"  
  system_packages:  
    - "apt-transport-https"  
    - "ca-certificates"  
    - "curl"  
    - "software-properties-common"  
    - "python3-pip"  
  packages:  
    - "docker-ce"  
  pip:  
    - "docker"
```

These two files contain all of the information we will need to install Docker CE.

Tasks

As we are targeting two different operating systems, our `tasks/main.yml` file needs to look as follows:

```
---
# tasks file for ansible-role-docker

- name: include the operating system specific variables
  include_vars: "{{ ansible_os_family }}.yml"

- name: install the stack on centos
  import_tasks: install-redhat.yml
  when: ansible_os_family == 'RedHat'

- name: install the stack on ubuntu
  import_tasks: install-ubuntu.yml
  when: ansible_os_family == 'Debian'
```

As you can see, this is the same as when we installed our LEMP Stack on the two operating systems in [Chapter 6, Targeting Multiple Distributions](#). The `tasks/install-redhat.yml` file looks pretty much like the tasks we have used to install Docker in the previous chapters:

```
---
# tasks file for ansible-role-docker

- name: add the gpg key for the docker repo
  rpm_key:
    key: "{{ docker.gpg_key }}"
    state: "present"

- name: add docker repo from the remote url
  get_url:
    url: "{{ docker.repo_url }}"
    dest: "{{ docker.repo_path }}"
    mode: "0644"

- name: install the docker packages
  yum:
    name: "{{ item }}"
    state: "installed"
    update_cache: "yes"
    enablerepo: "{{ docker.edge }}"
    with_items: "{{ docker.packages }}"

- name: install pip
  easy_install:
    name: pip
    state: latest

- name: install the python packages
  pip:
```

```

    name: "{{ item }}"
  with_items: "{{ docker.pip }}"

- name: put selinux into permissive mode
  selinux:
    policy: targeted
    state: permissive

- name: start docker and configure to start on boot
  service:
    name: "docker"
    state: "started"
    enabled: "yes"

```

The only difference is that we are enabling the Docker CE Edge repository when installing the packages, and also we are not running a `yum update` when installing Docker. We are not doing this as it is not our role's decision to update a server when someone else is running the role; our role should only install Docker.

The final task file is `tasks/install-ubuntu.yml`. This, as you will have already guessed, contains the tasks to install Docker on Ubuntu hosts:

```

---
# tasks file for ansible-role-docker

- name: install the system packages
  apt:
    name: "{{ item }}"
    state: "present"
    update_cache: "yes"
  with_items: "{{ docker.system_packages }}"

- name: add the apt keys from a key server
  apt_key:
    url: "{{ docker.gpg_key }}"
    state: present

- name: add the apt repo
  apt_repository:
    repo: "{{ docker.repo }}"
    state: present

- name: install the docker package
  apt:
    name: "{{ item }}"
    state: "present"
    update_cache: "yes"
    force: "yes"
  with_items: "{{ docker.packages }}"

- name: install the python packages
  pip:
    name: "{{ item }}"
  with_items: "{{ docker.pip }}"

- name: start docker and configure to start on boot
  service:
    name: "docker"

```

```
|   state: "started"
|   enabled: "yes"
```

That concludes all of the tasks and variables that we need in order to install Docker on the two different operating systems. In previous chapters, that would have been enough for us to add the role to our playbook and run the tasks. However, as we are going to be publishing this role on Ansible Galaxy, we need to add some more information about the role.

Metadata

As you may have seen while browsing Ansible Galaxy, each of the roles uploaded has information on who wrote it, when it is for, a license, which version of Ansible is supported, and so on. This information is all taken from the `meta/main.yml` file. The one we published looks like the following:

```
---
galaxy_info:
  author: "Russ McKendrick"
  description: "Role to install the Docker CE Edge release on either an Enterprise Linux or Ubuntu host"
  license: "license (BSD)"
  min_ansible_version: 2.4
  platforms:
    - name: EL
      versions:
        - 6
        - 7
    - name: Ubuntu
      versions:
        - bionic
        - artful
        - xenial
  galaxy_tags:
    - docker
dependencies: []
```

As you can see, we are providing information in a YAML file that Ansible Galaxy will read when we publish the role. Most of the information in the file is self-explanatory, so I will not go into too much detail here:

- `author`: This is your name or chosen moniker.
- `description`: Add a description of your role; this will appear in searches on the command line and in the web interface, so keep it short and do not add any markup.
- `license`: The license you are releasing your role under; the default is BSD.
- `min_ansible_version`: The version of Ansible your role will work with. Remember, if you are using new functionality, then you must use the version that the functionality was released in. Saying you work with Ansible 1.9 but are using modules from Ansible 2.4 will only frustrate

users.

- `platforms`: This list of supported operating systems and releases is used when displaying information about the role, and it will play a part in a user choosing to use your role. Make sure this is accurate as, again, we do not want to frustrate users.
- `galaxy_tags`: These tags are used by Ansible Galaxy to help identify what your role does.

There is one final part of the role to look at before we publish it: the `README.md` file.

README

The final part of the role we need to complete is the `README.md` file; this contains the information that is displayed on the Ansible Galaxy website. When we initiated our role using `ansible-galaxy`, it created a `README.md` file with a basic structure. The one for our role looks like the following file:

```
Ansible Docker Role
=====
This role installs the current Edge build Docker CE using the official repo, for more
information on Docker CE see the official site at [https://www.docker.com/community-
edition](https://www.docker.com/community-edition).

Requirements
-----
Apart from requiring root access via `become: yes` this role has no special
requirements.

Role Variables
-----
All of the variables can be found in the `vars` folder.

Dependencies
-----
None.

Example Playbook
-----
An example playbook can be found below;

```
- hosts: docker
 gather_facts: true
 become: yes
 become_method: sudo

 roles:
 - russmckendrick.docker
```

License
-----
BSD

Author Information
-----
This role is published by [Russ Kendrick](http://russ.mckendrick.io/).
```

Now that we have all of the files we need in place, we can look at committing our role to GitHub and, from there, publish it on Ansible Galaxy.

Committing the code and publishing

Now that we have our completed role, we need to push it to a public GitHub repository. There are a few reasons why it needs to be published to a public repository, the most important of which is that any potential user will need to download your role. Also, Ansible Galaxy links to the repository, allowing users to review your role before they choose to execute it as part of their playbooks.

On all GitHub pages, when you are logged in, there is a + icon in the top right; clicking this will bring up a menu that contains options for creating a new repository and importing one, as well as gists and organizations. Select New repository from the menu and you will be presented with a screen that looks as follows:

The screenshot shows the GitHub interface for creating a new repository. At the top, there's a navigation bar with links for Pull requests, Issues, Marketplace, and Explore. Below the navigation is a search bar and a user profile icon. The main section is titled "Create a new repository" and contains the following fields:

- Owner:** A dropdown menu showing "russmckendrick" followed by a separator line and the repository name.
- Repository name:** A text input field containing "ansible-role-docker" with a green checkmark icon to its right.

Below these fields, a note says: "Great repository names are short and memorable. Need inspiration? How about solid-potato."

Description (optional): A text area containing "Example Ansible role for installing Docker CE Edge on Enterprise Linux and Ubuntu".

Visibility: A section with two radio button options:

- Public**: Description: "Anyone can see this repository. You choose who can commit."
- Private**: Description: "You choose who can see and commit to this repository."

Initialization: A checkbox labeled "Initialize this repository with a README". Below it, a note says: "This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository."

At the bottom, there are two buttons: "Add .gitignore: None" and "Add a license: None".

A large green "Create repository" button is centered at the bottom of the form.

Name the repository and enter a description; it is important that you name your repository `ansible-role-your-role-name`. The name of the role in Ansible Galaxy will depend on the name you give after `ansible-role`, so, in the previous example, our role will be called `your-role-name` and, for the role we are going to be publishing, it will be called `docker`.

Now that we have our repository, we need to add the files for our role. Go back to the folder that contains your role on the command line and then run the following commands to initialize the Git repository locally. Push it to GitHub, making sure that you replace the repository URL with that of your own repository:

```
$ git init  
$ git add -A .  
$ git commit -m "first commit"  
$ git remote add origin git@github.com:russmckendrick/ansible-role-docker.git  
$ git push -u origin master
```

You should now have your files uploaded, and your repository should look not too dissimilar to the following:

The screenshot shows a GitHub repository page for 'russmckendrick / ansible-role-docker'. At the top, there's a navigation bar with links for Pull requests, Issues, Marketplace, and Explore. Below the navigation is a search bar and a notification icon. The main header displays the repository name 'russmckendrick / ansible-role-docker' and metrics: 5 commits, 1 branch, 0 releases, and 1 contributor. There are buttons for creating a new pull request, uploading files, finding files, and cloning or downloading the repository. A list of files in the repository includes 'defaults', 'handlers', 'meta', 'tasks', 'tests', 'vars', and 'README.md', all checked in 15 hours ago. A 'TIP' icon with a lightbulb suggests reading GitHub documentation for Git setup.

Now that we have our files uploaded and available, we can sign in to Ansible Galaxy using our GitHub credentials and then import our role. Head to the Ansible Galaxy home page at <https://galaxy.ansible.com/> and then click on the Sign in with GitHub link; this will take you to GitHub and ask you to confirm that you are OK with giving Ansible Galaxy permission to access information on your account. Proceed as prompted and you will be returned to Ansible Galaxy.

Clicking on the My Content link in the top menu will take you to a page where you can import content from GitHub; if you do not see your repository listed, click on the refresh icon next to the search box:



Import Your Content from GitHub

Click the toggle next to the repository to reveal a check mark. This will add the repository to Galaxy, making it visible on the Search page and allowing anyone to download it. Removing the check mark will delete the repository from Galaxy. Use settings to enable Travis notifications and control the repository name.

If you don't see all of your repositories, [review and add](#) your authorized organizations.

A screenshot of the "Import Your Content from GitHub" page. It shows a list of repositories under the heading "russmck". One repository, "russmckendrick/ansible-role-docker", is listed with a green checked status, indicating it has been imported successfully. The status is shown as "Succeeded". At the bottom of the page, there are logos for Red Hat and Ansible, along with copyright information: "Copyright © 2018 Red Hat, Inc. | Security Disclosures | Privacy Policy | GALAXY 2.4.0".

When you see your repository listed, click on the on/off switch next to the role and that should do it. Your role is now imported. Clicking your username in the top menu will bring up a drop-down list; from this list, select My Imports. This will give you the logs of your import:

The screenshot shows the Ansible Galaxy interface. At the top, there's a navigation bar with links for ABOUT, EXPLORE, SEARCH, BROWSE AUTHORS, MY CONTENT, and a user profile for RUSSMCK. Below the navigation is a banner with the text "MY IMPORTS".

The main area is titled "IMPORTS" and features a search bar labeled "Search Imports". A green button labeled "+ ADD" is visible. On the left, a list shows a recent import entry for "russmckendrick/ansible-role-docker" with a green status indicator and the note "Finished: 6/9/18 8:04 PM".

The central part of the screen displays the details for the role "russmckendrick/ansible-role-docker". It includes a link to the GitHub repository and a download icon. Below this, a box shows the import progress:

```
master Update - fixing tags
Commit 946728a
Stars 0
Forks 0
Watchers 1

Starting import 249616: role_name=docker repo=russmckendrick/ansible-role-docker
Accessing branch: master
Parsing and validating meta data.
Parsing galaxy_tags
Parsing platforms
Parsing cloud platforms
No cloud platforms found in meta data
Parsing and validating README
Adding repo tags as role versions
Removing old tags
Import completed
Status SUCCESS : warnings=1 errors=0
```



Copyright © 2018 Red Hat, Inc. | Security Disclosures | Privacy Policy | GALAXY 2.4.0

Now your role has been published; you can view your role by clicking on the link at the top, where it says `russmckendrick/ansible-role-docker`. This will take you to the Ansible Galaxy page for your newly added role, for example, <https://galaxy.ansible.com/russmckendrick/docker/>:

The screenshot shows a web browser window with the URL `galaxy.ansible.com` in the address bar. The page header includes the Galaxy logo, navigation links for ABOUT, EXPLORE, SEARCH, BROWSE AUTHORS, MY CONTENT, and a user profile for RUSSMCK. A banner at the top right says "ROLE DETAIL". The main content area displays the details for the "russmckendrick.docker" role.

russmckendrick.docker

Role to install the Docker CE Edge release on either an Enterprise Linux or Ubuntu host



russmckendrick

[Details](#) [README](#)

Downloads 0

[Issue Tracker](#)

[Github Repo](#)

[Download](#)

[Watch 1](#)

[Star 0](#)

Type Ansible

Minimum Ansible Version 2.4

Installation `$ ansible-galaxy install russmckendrick.docker`

Tags docker

Last Commit 39 seconds ago

Last Imported 29 seconds ago

OS Platforms

Platform	Version
EL	6
EL	7
Ubuntu	artful
Ubuntu	bionic
Ubuntu	xenial

As you can see, all of the metadata we added is present in the listing along with links to view the README file, which was imported from GitHub, and also links to GitHub itself.

Testing the role

Now that we have our role, we can test it. To do this, we will need a playbook, inventory, and a requirements file, as well as a CentOS and Ubuntu server. Run the following commands to create the files you need:

```
$ mkdir docker  
$ cd docker  
$ touch production requirements.yml site.yml Vagrantfile
```

The inventory file, `production`, should look as follows:

```
centos ansible_host=10.20.30.10.nip.io  
ubuntu ansible_host=10.20.30.20.nip.io ansible_python_interpreter=/usr/bin/python3  
  
[docker]  
centos  
ubuntu  
  
[docker:vars]  
ansible_connection=ssh  
ansible_user=vagrant  
ansible_private_key_file=~/.ssh/id_rsa  
host_key_checking=False
```

Our `requirements.yml` file contains just our Docker role:

```
| - src: "russmckendrick.docker"
```

Our playbook, the `site.yml` file, should just call our role:

```
---  
- hosts: docker  
  gather_facts: true  
  become: yes  
  become_method: sudo  
  
  roles:  
    - russmckendrick.docker
```

Finally, the `Vagrantfile` should read:

```
# -*- mode: ruby -*-  
# vi: set ft=ruby :  
  
API_VERSION = "2"  
DOMAIN = "nip.io"  
PRIVATE_KEY = "~/.ssh/id_rsa"  
PUBLIC_KEY = '~/.ssh/id_rsa.pub'
```

```

CENTOS_IP = '10.20.30.10'
CENTOS_BOX = 'centos/7'
UBUNTU_IP = '10.20.30.20'
UBUNTU_BOX = 'generic/ubuntu1804'

Vagrant.configure(API_VERSION) do |config|


  config.vm.define "centos" do |centos|
    centos.vm.box = CENTOS_BOX
    centos.vm.network "private_network", ip: CENTOS_IP
    centos.vm.host_name = CENTOS_IP + '.' + DOMAIN
    centos.ssh.insert_key = false
    centos.ssh.private_key_path = [PRIVATE_KEY, "~/.vagrant.d/insecure_private_key"]
    centos.vm.provision "file", source: PUBLIC_KEY, destination:
    "~/.ssh/authorized_keys"

    centos.vm.provider "virtualbox" do |v|
      v.memory = "2024"
      v.cpus = "2"
    end

    centos.vm.provider "vmware_fusion" do |v|
      v.vmx["memsize"] = "2024"
      v.vmx["numvcpus"] = "2"
    end
  end

  config.vm.define "ubuntu" do |ubuntu|
    ubuntu.vm.box = UBUNTU_BOX
    ubuntu.vm.network "private_network", ip: UBUNTU_IP
    ubuntu.vm.host_name = UBUNTU_IP + '.' + DOMAIN
    ubuntu.ssh.insert_key = false
    ubuntu.ssh.private_key_path = [PRIVATE_KEY, "~/.vagrant.d/insecure_private_key"]
    ubuntu.vm.provision "file", source: PUBLIC_KEY, destination:
    "~/.ssh/authorized_keys"

    ubuntu.vm.provider "virtualbox" do |v|
      v.memory = "2024"
      v.cpus = "2"
    end

    ubuntu.vm.provider "vmware_fusion" do |v|
      v.vmx["memsize"] = "2024"
      v.vmx["numvcpus"] = "2"
    end
  end
end

```

Now that we have all of the files in place, we can download our role by running:

```
| $ ansible-galaxy install -r requirements.yml
```

As you can see from the following output, this will download our role to the `~/ansible/roles/` folder:

```
1. docker (bash)
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter16/docker on master*
⚡ ansible-galaxy install -r requirements.yml
- downloading role 'docker', owned by russmckendrick
- downloading role from https://github.com/russmckendrick/ansible-role-docker/archive/master.tar.gz
- extracting russmckendrick.docker to /Users/russ/.ansible/roles/russmckendrick.docker
- russmckendrick.docker (master) was installed successfully
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter16/docker on master*
⚡
```

Next, launch the two Vagrant boxes by running either one of the following commands:

```
| $ vagrant up
| $ vagrant up --provider=vmware_fusion
```

Once the boxes are up and running, we can run the playbook with:

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

As you can see from the following output, everything went as planned and the role installed Docker on both boxes:

```
1. docker (bash)
TASK [russmckendrick.docker : add the apt repo] ****
skipping: [centos]
changed: [ubuntu]

TASK [russmckendrick.docker : install the docker package] ****
skipping: [centos] => (item=[])
changed: [ubuntu] => (item=u'docker-ce')

TASK [russmckendrick.docker : install the python packages] ****
skipping: [centos] => (item=docker)
changed: [ubuntu] => (item=docker)

TASK [russmckendrick.docker : start docker and configure to start on boot] ****
skipping: [centos]
ok: [ubuntu]

PLAY RECAP ****
centos      : ok=9    changed=7    unreachable=0    failed=0
ubuntu      : ok=8    changed=5    unreachable=0    failed=0

russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter16/docker on master*
⚡
```

Ansible Galaxy commands

Before we finish this chapter, let's take a quick look at some of the other functionalities of the `ansible-galaxy` command, starting with logging in.

Logging in

It is possible to log in to Ansible Galaxy from the command line; you can do this by using the following:

```
| $ ansible-galaxy login
```

This will ask for your GitHub username and password; if you have two-factor authentication enabled on your GitHub account, which you really should do, then this method will not work. Instead, you will need to provide a personal access token. You can generate a personal access token at the following URL: <https://github.com/settings/tokens/>. Once you have a token, you can use the following command, replacing the token with your own:

```
| $ ansible-galaxy login --github-token 0aa7c253044609b98425865wbf6z679a94613bae89
```

The following screenshot shows the output for the preceding command:



```
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter16/docker on master*
⚡ ansible-galaxy login --github-token
Successfully logged into Galaxy as russmck
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter16/docker on master*
```



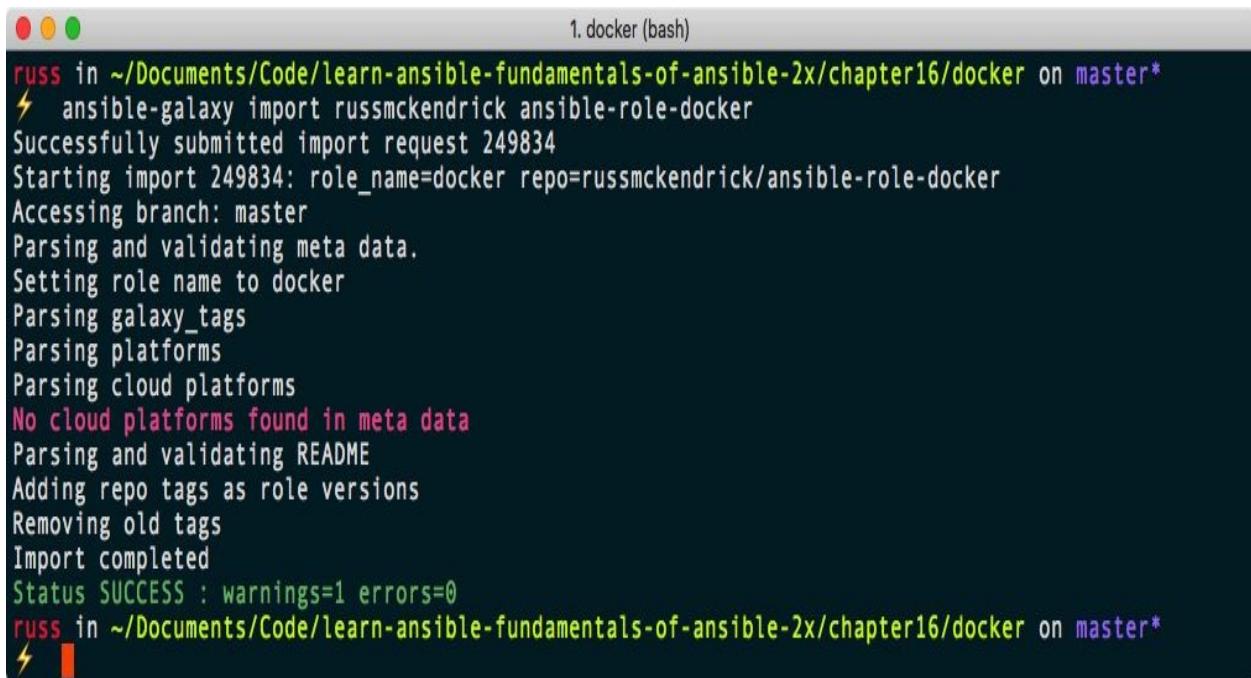
Personal access tokens will give anyone who has one full access to your GitHub account; please store them securely and, if possible, rotate them regularly.

Importing

Once logged in, if you make a change to your role and want to import those changes into Ansible Galaxy, you can run the following command:

```
| $ ansible-galaxy import russmckendrick ansible-role-docker
```

The following screenshot shows the output for the preceding command:



```
1. docker (bash)
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter16/docker on master*
⚡ ansible-galaxy import russmckendrick ansible-role-docker
Successfully submitted import request 249834
Starting import 249834: role_name=docker repo=russmckendrick/ansible-role-docker
Accessing branch: master
Parsing and validating meta data.
Setting role name to docker
Parsing galaxy_tags
Parsing platforms
Parsing cloud platforms
No cloud platforms found in meta data
Parsing and validating README
Adding repo tags as role versions
Removing old tags
Import completed
Status SUCCESS : warnings=1 errors=0
russ in ~/Documents/Code/learn-ansible-fundamentals-of-ansible-2x/chapter16/docker on master*
```

The two bits of information we are passing to the command are the GitHub username, `russmckendrick` in my case, and the name of repository we want to import—so for the Docker one we published in the last section, I am using `ansible-role-docker`.

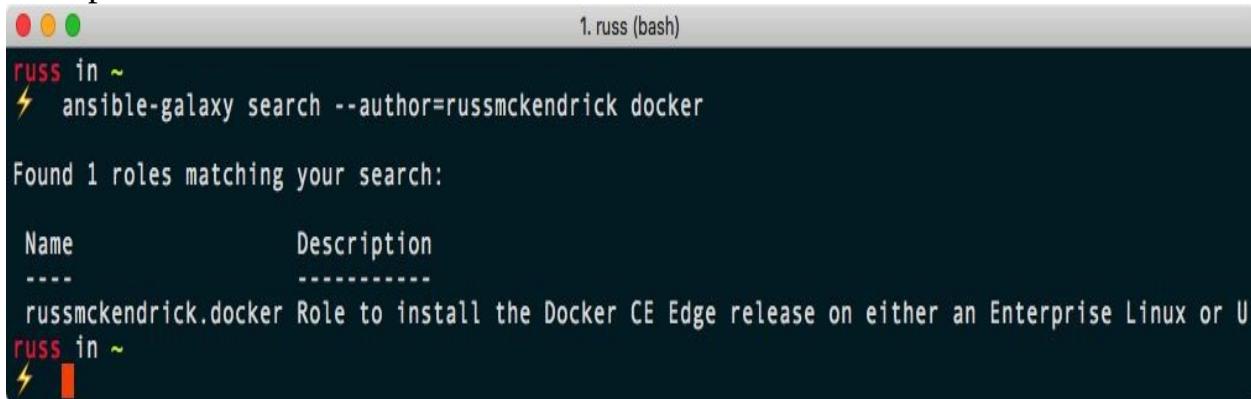
Searching

You can use the `ansible-galaxy` command to search for roles. For example, running the following currently returns 725 roles: **\$ ansible-galaxy search docker**

If you want to search for a role by an author, you can use the following:

```
| $ ansible-galaxy search --author=russmckendrick docker
```

As you can see from the output in the screenshot, this returns just the role we have published:



The screenshot shows a terminal window with a dark background and light-colored text. At the top, there are three small colored icons (red, yellow, green). The title bar says "1. russ (bash)". The terminal prompt is "russ in ~". Below the prompt, the command "ansible-galaxy search --author=russmckendrick docker" is entered. The output shows "Found 1 roles matching your search:" followed by a table with two columns: "Name" and "Description". The single row in the table is "russmckendrick.docker Role to install the Docker CE Edge release on either an Enterprise Linux or U". The bottom of the terminal shows the prompt again: "russ in ~".

Name	Description
russmckendrick.docker	Role to install the Docker CE Edge release on either an Enterprise Linux or U

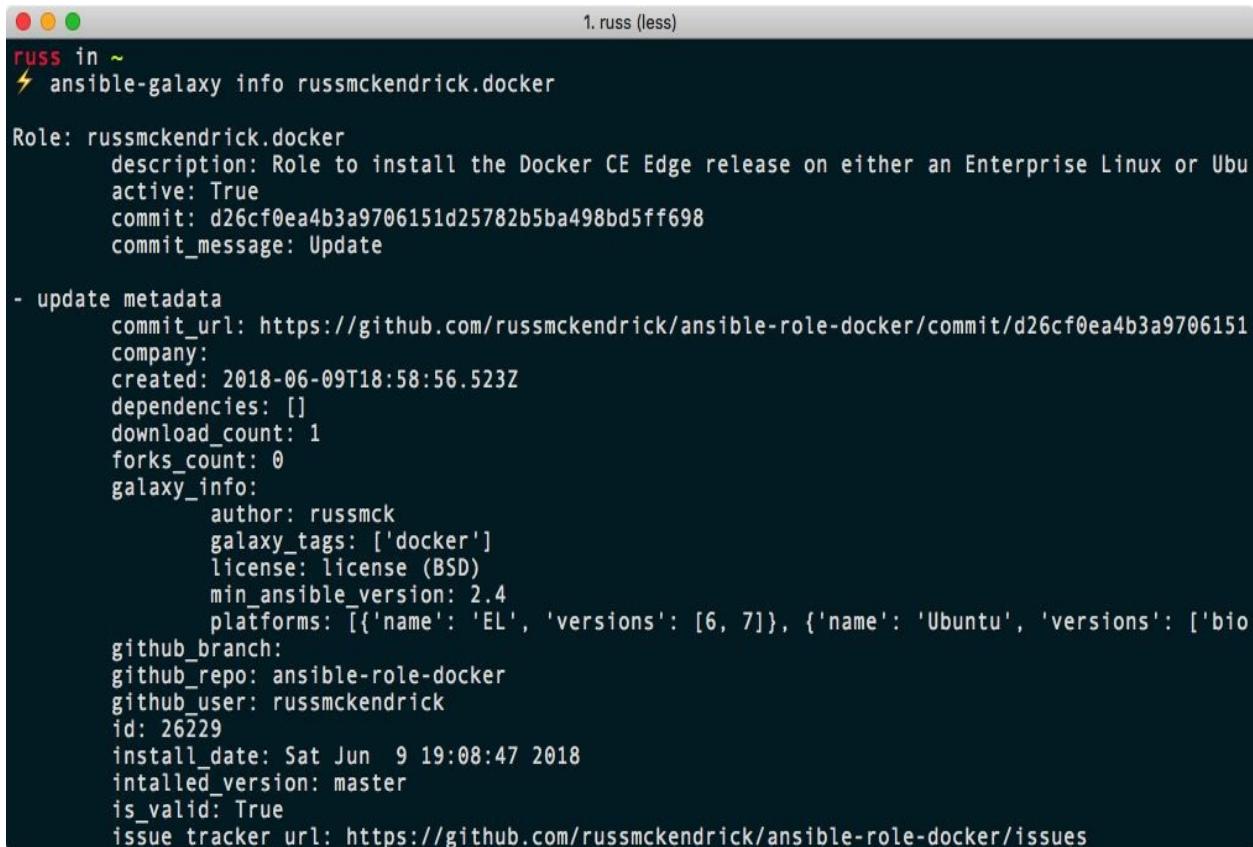
This is useful as you do not have to start switching between your terminal and browser.

Info

The final command we are going to look at is the `info` one; this command will print out information on any role you provide to it. For example, running the following will give you lots of information on the role we published:

```
| $ ansible-galaxy info russmckendrick.docker
```

The following screenshot shows the output for the preceding command:



The screenshot shows a terminal window titled "1. russ (less)". The command run is "ansible-galaxy info russmckendrick.docker". The output displays detailed information about the role "russmckendrick.docker". Key details include:

- Role:** russmckendrick.docker
- description:** Role to install the Docker CE Edge release on either an Enterprise Linux or Ubuntu.
- active:** True
- commit:** d26cf0ea4b3a9706151d25782b5ba498bd5ff698
- commit_message:** Update
- update metadata**
 - commit_url:** <https://github.com/russmckendrick/ansible-role-docker/commit/d26cf0ea4b3a9706151d25782b5ba498bd5ff698>
 - company:**
 - created:** 2018-06-09T18:58:56.523Z
 - dependencies:** []
 - download_count:** 1
 - forks_count:** 0
 - galaxy_info:**
 - author:** russmck
 - galaxy_tags:** ['docker']
 - license:** license (BSD)
 - min_ansible_version:** 2.4
 - platforms:** [{"name": "EL", "versions": [6, 7]}, {"name": "Ubuntu", "versions": ["bio"]}]
 - github_branch:**
 - github_repo:** ansible-role-docker
 - github_user:** russmckendrick
 - id:** 26229
 - install_date:** Sat Jun 9 19:08:47 2018
 - installed_version:** master
 - is_valid:** True
 - issue_tracker_url:** <https://github.com/russmckendrick/ansible-role-docker/issues>

As you can see, all of the information you can get on the website is also available on the command line, meaning you can have a choice when it comes to how you interact with Ansible Galaxy.

Summary

In this chapter, we have taken an in-depth look at Ansible Galaxy, both the website and command-line tool. I am sure that you will agree that Ansible Galaxy offers valuable community services in that it allows Ansible users to share roles for common tasks as well as a way for users to make contributions to the Ansible community by publishing their own roles.

However, just be careful. Remember to check through the code and also read through bug trackers before using roles from Ansible Galaxy in production environments; after all, a lot of these roles need to have escalated privileges in order to be able to successfully execute their tasks.

In the next and final chapter, we are going to be looking at some of the ways Ansible can be integrated into your day-to-day workflows.

Questions

There is just one task for this chapter. Take one of the other roles we have created previously, adapt it for use on more than one operating system, and publish it on Ansible Galaxy.

Further reading

Both of the roles we used at the start of the chapter were published by Jeff Geerling; you can find out more about Jeff and his projects at <https://www.jeffgeerling.com/>.

Next Steps with Ansible

In this chapter, we will discuss how Ansible can be integrated into your day-to-day workflows, from continuous integration tools to monitoring tools and troubleshooting. We will discuss the following topics:

- How to integrate Ansible with services such as Slack
- How you can use Ansible to troubleshoot problems when they occur
- Some real-world examples

Let's dive straight in and look at how we can hook our playbooks into third-party services.

Integrating with third-party services

Although you may be the one running the playbooks, you may keep a log of your playbook run or keep other members of your team or even other departments up to date with the results of the playbook run. Ansible ships with several core modules that allow you to work with third-party services to provide real-time notifications.

Slack

Slack has very quickly become the go-to choice for team-based collaboration services for various departments in IT services. Not only does it support third-party applications via its App Directory, but it also has a powerful API that you can use to bring your tools into the chat rooms provided by Slack.

We are going to look at the example in this section, the full playbook, which can be found in the `chapter17/slack` folder in the GitHub repository. I have taken the playbook from [chapter 9, Building Out a Cloud Network](#), where we created a VPC in AWS, and I have adapted it to use the `slack` Ansible module.

Generating a token

Before we can use the Slack module in our Playbook, we need an access token to request one login to your Slack workspace; you can sign up for a workspace for free at <https://slack.com/> if you don't already have one.

Once you are logged into your workspace, either using the web client or desktop application, select the Manage apps option from the Administration menu, as seen in the following screenshot:

The screenshot shows the Slack application interface. At the top, there's a dark header bar with the title "Learn Ansible" and a dropdown arrow, followed by the user handle "russmckendrick". To the right of the header are icons for phone, info, gear, search ("Search"), and other settings. Below the header, the channel "#random" is selected, indicated by a blue border. The channel header shows 1 member online and 0 messages, with a description: "Non-work banter and water cooler co...". On the left, a sidebar menu is open, showing options like "Set a status", "Profile & account", "Preferences", "Set yourself to away", "Help & feedback", and a section for the workspace "Learn Ansible" (learningansible.slack.com). This section includes "Administration" (with "Manage members", "Manage apps", and "Workspace settings" sub-options), "Invite people", "Customize Slack", "Analytics", "Sign out of Learn Ansible", and "Sign in to another workspace ...". At the bottom of the sidebar, there's a dark purple area. The main content area shows the "#random" channel with one message from "russmckendrick" at 12:47 PM stating "joined #random.". Below the message is a input field with a plus sign and the placeholder "Message #random" and an emoji button.

This will open your browser and take you to the App Directory for your workspace; from here, search for `incoming webhooks` and then click on Add Configuration.

The first part of the configuration is to choose which channel you would like the incoming Webhook to post messages to. I selected the general channel—once selected, you will be taken to a page that gives you a Webhook URL; make sure that you make a note of this URL as we will need it shortly. At the bottom of the page, you have the option to customize your Webhook.

In the Integration Settings at the bottom of the page, I entered the following information:

- Post to Channel: I left it as #general
- Webhook URL: This is prepopulated for you; you also have the choice of regenerating the URL here
- Descriptive Label: I entered Ansible here
- Customize Name: I entered Ansible here as well
- Customize Icon: I left this as it was

Once the preceding details had been filled in, I clicked on the Save Settings button; this left me with a single incoming Webhook:

The screenshot shows a web browser window with the URL `learnansible.slack.com` in the address bar. The page is titled "App Directory" and features the Slack logo. A search bar says "Search App Directory". On the right, there are buttons for "Browse", "Manage", and "Build", along with a "Learn Ansible" button. The main content area is titled "Incoming WebHooks" and includes a large icon of a red and black interlocking circular logo. Below the icon are two buttons: "Add Configuration" (green) and "App Homepage" (white). To the right, under the heading "Configurations", is a list entry: "Posts to #general as Ansible" by "russmckendrick" on Jun 23, 2018, with a small profile picture next to the name.

As mentioned already, I also made a note of the Webhook URL; for me, it was:

`https://hooks.slack.com/services/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`

Now that we have everything, we need to be configured on the Slack side to be able to start to send messages to our users using Ansible.

The Ansible playbook

I am only going to cover the updating of the tasks for a single role here, and it is the role that creates the VPC. The first change I made was to add a few lines to the `group_vars/common.yml` file:

```
---
environment_name: "VPC-Slack"
ec2_region: "eu-west-1"

slack:
  token: "TBCRVDGMA/BBCPTPNH1/tyudQIccviG7gh4JnfeoPScc"
  username: "Ansible"
  icon:
    "https://upload.wikimedia.org/wikipedia/commons/thumb/0/05/Robot_icon.svg/200px-Robot_icon.svg.png"
```

As you can see, I added the following three nested variables:

- `token`: This is taken from the Webhook URL; as you can see, I entered everything after `https://hooks.slack.com/services/`
- `username`: We can override the username through which the updates will be posted, as I just kept it as Ansible
- `icon`: This is the avatar that will be displayed as part of our post

If you remember the VPC role from before, you'll recall that it contained a single task that used the `ec2_vpc_net` module to create a VPC. Now, we want to introduce Slack notifications, and to be able to provide our users with feedback. So, first of all, let's send a notification to say that we are checking whether the VPC exists:

```
- name: Send notification message via Slack all options
slack:
  token: "{{ slack.token }}"
  msg: "Checking for VPC called '{{ environment_name }}'"
  username: "{{ slack.username }}"
  icon_url: "{{ slack.icon }}"
  link_names: 0
  parse: 'full'
```

As you can see from the preceding task, we are sending a message that, in our case, will read `Checking for VPC called 'VPC-Slack'`, along with `token`, `username`, and `icon`. The next task in the role is the one from the original role:

```

- name: ensure that the VPC is present
  ec2_vpc_net:
    region: "{{ ec2_region }}"
    name: "{{ environment_name }}"
    state: present
    cidr_block: "{{ vpc_cidr_block }}"
    resource_tags: { "Name" : "{{ environment_name }}", "Environment" : "{{ environment_name }}" }
  register: vpc_info

```

Now, one of two things would have happened: a VPC called `vpc-slack` would have been created, or Ansible would have gathered information on an already-existing VPC called `vpc-slack`. When we send the message to our users, it should change depending upon what Ansible did. The following task sends a message informing our users that a new VPC has been created:

```

- name: Send notification message via Slack all options
  slack:
    token: "{{ slack.token }}"
    msg: "VPC called '{{ environment_name }}' created with an ID of '{{ vpc_info.vpc.id }}'"
    username: "{{ slack.username }}"
    icon_url: "{{ slack.icon }}"
    link_names: 0
    parse: 'full'
  when: vpc_info.changed

```

Note that I only run this task when the `vpc_info` variable I registered is marked as changed. Also, I pass the ID of the VPC as part of the message. If `vpc_info` didn't register any changes, then the preceding task will be skipped; the following task will run instead:

```

- name: Send notification message via Slack all options
  slack:
    token: "{{ slack.token }}"
    msg: "Found a VPC called '{{ environment_name }}' which has an ID of '{{ vpc_info.vpc.id }}'"
    username: "{{ slack.username }}"
    icon_url: "{{ slack.icon }}"
    link_names: 0
    parse: 'full'
  when: vpc_info.changed == false and vpc_info.failed == false

```

Note how I am changing the wording and also how it is only being called when there is no change. I went through the rest of the roles, adding tasks to send notifications to Slack using the same logic as in the preceding code; as mentioned, you can find all the additions in the `chapter17/slack` folder in the repository.

Running the playbook

When running the playbook, use the following commands:

```
| $ export AWS_ACCESS_KEY=AKIAI5KECPOTNTTVM3EDA  
| $ export AWS_SECRET_KEY=Y4B7FFiSw10Am3VIFc071gnc/TAtK5+RpxzIGTr  
| $ ansible-playbook -i production site.yml
```

I received the following notifications from Slack:

The screenshot shows the Slack interface for the '#general' channel. The left sidebar lists workspace channels, direct messages, and apps. The main area displays the channel's purpose and recent activity. A message from 'Ansible' details the creation of a VPC, an internet gateway, and an elastic load balancer.

You created this channel today. This is the very beginning of the **#general** channel.

Purpose: This channel is for workspace-wide communication and announcements. All members are in this channel. ([edit](#))

+ Add an app [Invite others to this channel](#)

Today

russmckendrick 12:47 PM
joined #general.

russmckendrick 3:39 PM
added an integration to this channel: Ansible

Ansible APP 4:18 PM
Checking for VPC called 'VPC-Slack'
VPC called 'VPC-Slack' created with an ID of 'vpc-dbbcffbd'
Ensuring that the subnets are present in 'VPC-Slack'
Checking there is an internet gateway configured in 'VPC-Slack'
Adding an internet gateway in 'VPC-Slack' which has an ID of 'igw-48fab92f'
Ensuring that the route is present for the internet gateway in 'VPC-Slack'
Ensuring that the security groups are present in 'VPC-Slack'
Ensuring that the target group is present in 'VPC-Slack'
Checking for an Elastic Load Balancer in 'VPC-Slack'

System details
Created an Elastic Load Balancer in 'VPC-Slack'
ELB URL
VPC-Slack-elb-854065100.eu-west-1.elb.amazonaws.com

+ Message #general @

As you can see, a lot of the messages are talking about services within the VPC being created. Rerunning the playbook immediately after returns the following results:

The screenshot shows the Slack desktop application interface. On the left is the sidebar with a dark theme, displaying:

- A red status indicator.
- A profile picture icon.
- The workspace name: **Learn Ansible**.
- The user's name: **russmckendrick**.
- A bell icon with a notification count of 1.
- A list of channels:
 - All Threads
 - Channels
 - # general (highlighted in green)
 - # random
 - Direct Messages
 - slackbot
 - russmckendrick (you)
 - + Invite People
 - Apps
- A plus sign icon for creating new channels.

The main pane shows the **#general** channel. At the top of the channel view are icons for phone, info, gear, search (with the word "Search"), and a menu. Below the header, there is a "Today" section with the following messages:

- Ensuring that the subnets are present in 'VPC-Slack'
- Checking there is an internet gateway configured in 'VPC-Slack'
- Adding an internet gateway in 'VPC-Slack' which has an ID of 'igw-48fab92f'
- Ensuring that the route is present for the internet gateway in 'VPC-Slack'
- Ensuring that the security groups are present in 'VPC-Slack'
- Ensuring that the target group is present in 'VPC-Slack'
- Checking for an Elastic Load Balancer in 'VPC-Slack'

Below these messages is a section titled **System details** with the following information:

- Created an Elastic Load Balancer in 'VPC-Slack'
- ELB URL**
VPC-Slack-elb-854065100.eu-west-1.elb.amazonaws.com

At the bottom of the channel view, there is a "new messages" button. The bottom of the screen features a message input field with a plus sign icon, the text "Message #general", and a recipient icon.

This time, the messages are talking about finding existing services and returning the IDs. Slack is just one service. Let's now take a brief look at a few more of the services you can interact with from your Ansible playbooks.

Other services

Slack is not the only service that Ansible can interact with; here are some more that you may want to use in your playbooks.

Campfire

Campfire is a chat service built into Basecamp; you can use this module to send updates to your project stakeholders directly from Ansible, for example:

```
- name: Send a message to Campfire
```

```
campfire:
```

```
subscription: "my_subscription"
```

```
token: "my_subscription"
```

```
room: "Demo"
```

```
notify: "loggins"
```

```
msg: "The task has completed and all is well"
```

Cisco Webex Teams (Cisco Spark)

Cisco Webex Teams, or Cisco Spark as it was formally known, is the collaboration service from Cisco that provides your teams with virtual meeting spaces, messages, and video calling. Also, it has a rich API that Ansible can be configured to interact with:

```
- name: Send a message to Cisco Spark
  cisco_spark:
    recipient_type: "roomId"
    recipient_id: "{{ spark.room_id }}"
    message_type: "markdown"
    personal_token: "{{ spark.token }}"
    message: "The task has **completed** and all is well"
```

CA Flowdock

CA Flowdock is a messaging service, which has been designed from the ground up to integrate with developer-focused services, such as GitHub, Bitbucket, Jira, Jenkins, and Ansible:

```
- name: Send a message to a Flowdock inbox
  flowdock:
    type: "inbox"
    token: "{{ flowdock.token }}"
    from_address: "{{ flowdock.email }}"
    source: "{{ flowdock.source }}"
    msg: "The task has completed and all is well"
    subject: "Task Success"
```

Hipchat

Hipchat is the group-messaging service provided by Atlassian; it has tight integration with the rest of the Atlassian family of products:

- name: Send a message to a Hipchat room

hipchat:

```
api: "https://api.hipchat.com/v2/"  
token: "{{ hipchat.token }}"  
room: "{{ hipchat.room }}"  
msg: "The task has completed and all is well"
```

Mail

This service shouldn't need any introduction; Ansible can be configured to send emails using all sorts of configuration. The following example shows an email being sent through an external SMTP server:

```
- name: Send an email using external mail servers
  mail:
    host: "{{ mail.smtp_host }}"
    port: "{{ mail.smtp_port }}"
    username: "{{ mail.smtp_username }}"
    password: "{{ mail.smtp_password }}"
    to: "Russ Kendrick <russ@mckendrick.io>"
    subject: "Task Success"
    body: "The task has completed and all is well"
    delegate_to: localhost
```

Mattermost

Mattermost is an open source alternative to proprietary services like the ones we have covered elsewhere in the list (for example, Slack, Cisco Webex Teams, and Hipchat):

```
- name: Send a message to a Mattermost channel
mattermost:
  url: "{{ mattermost.url }}"
  api_key: "{{ mattermost.api_key }}"
  text: "The task has completed and all is well"
  channel: "{{ mattermost.channel }}"
  username: "{{ mattermost.username }}"
  icon_url: "{{ mattermost.icon_url }}"
```

Say

Most modern computers come with some level of voice synthesis built in; using this module, you can have Ansible verbally inform you of the status of your playbook run:

```
- name: Say a message on your Ansible host
```

```
say:
```

```
msg: "The task has completed and all is well"
```

```
voice: "Daniel"
```

```
delegate_to: localhost
```

ServiceNow

ServiceNow is the enterprise-grade IT service management Software as a service product provided by ServiceNow, Inc. Using the `snow_record` module, your playbook can open incidents within your ServiceNow installation:

```
- name: Create an incident in ServiceNow
  snow_record:
    username: "{{ snow.username }}"
    password: "{{ snow.password }}"
    instance: "{{ snow.instance }}"
    state: "present"
    data:
      short_description: "The task has completed and all is well"
      severity: "3"
      priority: "3"
  register: snow_incident
```

Syslog

If you ship the log files from your hosts, then you may want to send the results of the playbook run to your hosts syslog so that it is shipped to your central logging service:

```
- name: Send a message to the hosts syslog
  syslogger:
    msg: "The task has completed and all is well"
    priority: "info"
    facility: "daemon"
    log_pid: "true"
```

Twilio

Use your Twilio account to send an SMS message directly from your Ansible playbook, as follows:

```
- name: Send an SMS message using Twilio
  twilio:
```

```
  msg: "The task has completed and all is well"
  account_sid: "{{ twilio.account }}"
  auth_token: "{{ twilio.auth }}"
  from_number: "{{ twilio.from_number }}"
  to_number: "+44 7911 123456"
  delegate_to: localhost
```

Summary of third-party services

One of the takeaways I hope you get from this book is that automation is great—it is not only a real-time saver, but using tools like the ones we covered in the previous chapter, Ansible Tower and Ansible AWX, can enable people who are not sys-admins or developers to execute their playbooks from a friendly web interface.

The modules we have covered in this section allow you take your automation to the next level by not only allowing you to record the results, but to also automatically do some housekeeping during your playbook run and have it notify your users itself.

Let's say, for example, you need to deploy a new configuration to your server. Your service desk raises a change for you to action the work within your ServiceNow installation. Your playbook could be written in such a way that before the change is actioned, it uses the `fetch` module to copy the configuration file to your Ansible Controller. The playbook could then use the `snow_record` module attach a copy of the existing configuration file to the change request, proceed to make the changes, and then automatically update the change request with the results.

You can find details on the services mentioned in this part of the chapter at the following URLs:

- **Slack:** <https://slack.com/>
- **Campfire:** <https://basecamp.com/>
- **Cisco Webex Teams (Cisco Spark):** <https://www.webex.com/products/teams/>
- **CA Flowdock:** <https://www.flowdock.com/>
- **Mattermost:** <https://mattermost.com/>
- **ServiceNow:** <https://www.servicenow.com/>
- **Twilio:** <https://twilio.com/>

The Ansible playbook debugger

Ansible has a debugger built in. Let's take a look at how you can build this into your playbook by creating a simple playbook with an error. As we have just mentioned it, we are going to write a playbook that uses the `say` module. The playbook itself looks as follows:

```
---
- hosts: localhost
  gather_facts: false
  debugger: "on_failed"

  vars:
    message: "The task has completed and all is well"
    voice: "Daniel"

  tasks:
    - name: Say a message on your Ansible host
      say:
        msg: "{{ message }}"
        voice: "{{ voice }}"
```

There are two things to point out: the first being the mistake. As you can see, we are defining a variable named `message`, but when we come to use it the task, I have made a typo and entered `massage` instead. Luckily, as I am developing the playbook, I have instructed Ansible to drop to the interactive debugger whenever a task fails.

Debugging the task

Let's run the playbook and see what happens:

```
| $ ansible-playbook playbook.yml
```

The first problem is that we are not passing a host inventory file, so there will be warnings that only the localhost is available; this is fine, as we want to run the say module only on our Ansible Controller anyway:

```
[WARNING]: Unable to parse /etc/ansible/hosts as an inventory source
[WARNING]: No inventory was parsed, only implicit localhost is available
[WARNING]: provided hosts list is empty, only localhost is available. Note that the
implicit
localhost does not match 'all'
```

Next, Ansible runs the play itself; this should result in a fatal error:

```
PLAY [localhost]
*****
TASK [Say a message on your Ansible host]
*****
fatal: [localhost]: FAILED! => {"msg": "The task includes an option with an undefined variable. The error was: 'massage' is undefined\n\nThe error appears to have been in\n'/Users/russ/Documents/Code/learn-ansible-fundamentals-of-ansible-\n2x/chapter17/say/playbook.yml': line 12, column 7, but may\nbe elsewhere in the file\ndepending on the exact syntax problem.\n\nThe offending line appears to be:\n\n  tasks:\n    - name: Say a message on your Ansible host\n      ^ here\n"}
```

Typically, the playbook run will stop, and you will be returned to your shell; however, because we have instructed Ansible to drop into the interactive debugger, we now see the following prompt:

```
| [localhost] TASK: Say a message on your Ansible host (debug)>
```

From here, we can start to look into the problem a little more; for example, we can review the error by typing the following command:

```
| p result._result
```

Once you hit the *Enter* key, the results of the failed task will be returned:

```
[localhost] TASK: Say a message on your Ansible host (debug)> p result._result
{'failed': True,
 'msg': u"The task includes an option with an undefined variable. The error was:"}
```

```
'message' is undefined\n\nThe error appears to have been in\n'/Users/russ/Documents/Code/learn-ansible-fundamentals-of-ansible-\n2x/chapter17/say/playbook.yml': line 12, column 7, but may\nbe elsewhere in the file\ndepending on the exact syntax problem.\n\nThe offending line appears to be:\n\n\n  tasks:\n    - name: Say a message on your Ansible host\n      ^ here\n}\n[localhost] TASK: Say a message on your Ansible host (debug)>
```

Let's take a closer look at the variables used in the task by typing the following:

```
| p task.args
```

This will return the two arguments we are using in the task:

```
[localhost] TASK: Say a message on your Ansible host (debug)> p task.args\n{u'msg': u'{{ message }}', u'veoice': u'{{ voice }}'}\n[localhost] TASK: Say a message on your Ansible host (debug)>
```

Now, let's take a look at the variables that are available to the task using the following:

```
| p task_vars
```

You may have noted that we instructed Ansible to execute the setup module as part of the playbook run; that is to keep this list of available variables to a minimum:

```
[localhost] TASK: Say a message on your Ansible host (debug)> p task_vars\n{'ansible_check_mode': False,\n 'ansible_connection': 'local',\n 'ansible_current_hosts': [u'localhost'],\n 'ansible_diff_mode': False,\n 'ansible_facts': {},\n 'ansible_failed_hosts': [],\n 'ansible_forks': 5,\n 'ansible_inventory_sources': [u'/etc/ansible/hosts'],\n 'ansible_play_batch': [],\n 'ansible_play_hosts': [u'localhost'],\n 'ansible_play_hosts_all': [u'localhost'],\n 'ansible_playbook_python': '/usr/bin/python',\n 'ansible_python_interpreter': '/usr/bin/python',\n 'ansible_run_tags': [u'all'],\n 'ansible_skip_tags': [],\n 'ansible_version': {'full': '2.5.5',\n                    'major': 2,\n                    'minor': 5,\n                    'revision': 5,\n                    'string': '2.5.5'},\n 'environment': [],\n 'group_names': [],\n 'groups': {'all': [], 'ungrouped': []},\n 'hostvars': {},\n 'inventory_hostname': u'localhost',\n 'inventory_hostname_short': u'localhost',\n u'message': u'The task has completed and all is well',\n 'omit': '__omit_place_holder__0529a2749315462e1ae1a0d261987dedea3bfdad',
```

```
'play_hosts': [],  
'playbook_dir': u'/Users/russ/Documents/Code/learn-ansible-fundamentals-of-ansible-  
2x/chapter17/say',  
u'voice': u'Daniel'}  
[localhost] TASK: Say a message on your Ansible host (debug)>
```

As you can see, there is a lot of information there about the environment our playbook is being executed on. In the list of variables, you will notice that two of them are prefixed with a `u`: they are `voice` and `message`. We can find out more about these by running:

```
| p task_vars['message']  
| p task_vars['voice']
```

This will display the contents of the variable:

```
[localhost] TASK: Say a message on your Ansible host (debug)> p task_vars['message']  
u'The task has completed and all is well'  
[localhost] TASK: Say a message on your Ansible host (debug)> p task_vars['voice']  
u'Daniel'  
[localhost] TASK: Say a message on your Ansible host (debug)>
```

We know that we are passing a misspelt variable to the `msg` argument, so we make some changes on the fly and continue the playbook run. To do this, we are going to run the following command:

```
| task.args['msg'] = '{{ message }}'
```

This will update the argument to use the correct variable meaning, so that we can rerun the task by running the following:

```
| redo
```

This will immediately rerun the task with the correct argument and, with any luck, you should hear *The task has completed, and all is well*:

```
[localhost] TASK: Say a message on your Ansible host (debug)> task.args['msg'] = '{{  
message }}'  
[localhost] TASK: Say a message on your Ansible host (debug)> redo  
changed: [localhost]  
  
PLAY RECAP  
*****  
localhost : ok=1 changed=1 unreachable=0 failed=0
```

As you can see from the preceding output, because we only have a single task, the playbook completed. If we had more, then it would carry on from where it left off. You can now update your playbook with the correct spelling and proceed

with the rest of your day.

Also, if we wanted to, we could have typed either `continue` or `quit` to proceed or stop respectively.

Summary of the Ansible debugger

The Ansible debugger is an extremely useful option to enable when you are working on creating large playbooks—for example, imagine that you have a playbook that takes about 20 minutes to run, but there is an error somewhere toward the end, say, 15 minutes after you first run the playbook.

Having Ansible drop into the interactive debugger shell not only means you can see exactly what is and isn't defined, but it also means that you don't have to blindly make changes to your playbook and then wait another 15 minutes to see whether those changes resolved whatever was causing the fatal error.

Real-world examples

Before we finish the chapter, and also the book, I thought I would give a few examples of how I am using Ansible and interacting with Ansible: the first is interacting with Ansible using chat.

The chat example

A few months ago, I needed to set up a demo to show automation working—however, I needed to be able to show the demo on my laptop or phone, which meant that I couldn't assume I had access to the command line.

The demo I came up with ended up using Slack and a few other tools that we haven't covered in this book, namely Hubot and Jenkins; before I go into any details, let's quickly have a look at the output of the demo running:

The screenshot shows the Slack application interface. On the left is the sidebar with various workspace icons and sections like Direct Messages, Apps, and Invite People. The main area is a channel named '#demo' where a bot named 'bot' is interacting with the user. The bot is performing actions such as launching a Linux instance, checking system details, and reporting job status.

#demo

Russ McK Kendrick

Jump to... All Threads Channels # demo

Direct Messages slackbot Russ McK Kendrick (you)

+ Invite People Apps bot

Star 3 | 0 Demo stuff in here

Search Today

Russ McK Kendrick 5:37 PM @bot give me a linux server

bot APP 5:37 PM awslaunch is building. I'll let you know when it's done.

job <https://.../job/awslaunch/68/> added with id 752972.

Jenkins APP 5:38 PM Checking number of running instances

I am allowed to launch a maximum of 2 instances and I found 0, proceeding with launching your Linux instance

Generating a random instance name

Launching a Linux instance called 'Kind Keldysh'

Checking the VPC configuration

Looking for the right AMI to use

Using AMI ami-924aa8f5, which is Amazon Linux 2 LTS Candidate AMI 2017.12.0.20180509 x86_64 HVM GP2

Checking the public keys are up-to-date

Launching 'Kind Keldysh'

Instance 'Kind Keldysh' launched, waiting for SSH to become available

System details

Linux instance 'Kind Keldysh' is now ready

Hostname

ec2-18-130-152-190.eu-west-2.compute.amazonaws.com

Operating System	Key
Linux	russ

bot APP 5:40 PM @RussMcK Kendrick, job <https://.../job/awslaunch/68/> finished with status: SUCCESS.

+ Message #demo @ 😊

As you can see from the preceding output, I asked the following in a Slack

channel:

@bot give me a linux server

This then triggered an Ansible playbook run, which launched an instance in AWS and returned information on the instance once the playbook had confirmed that the server was available on the network. I also configured it to remove all running instances by asking the following:

@bot terminate all servers

As you can see, this runs another playbook, and this time, returns an animated GIF once the instance has been removed:

The screenshot shows the Slack desktop application interface. On the left is the sidebar with various channels and users listed. The main area shows a channel named '#demo' with several messages. One message from 'bot' at 5:42 PM says '@bot terminate all servers'. Another message from 'bot' at 5:42 PM says 'awsremove is building. I'll let you know when it's done.' A message from 'Jenkins APP' at 5:43 PM says 'Checking number of running instances'. A message from 'bot' at 5:43 PM says 'I found you have 1 instances, terminating them now'. Below this, a message from 'bot' at 5:44 PM says 'Instances terminated (2 MB)'. An image of a person in a plaid shirt is attached to this message. At the bottom, there is a message input field with '+ Message #demo' and a send button.

#demo

☆ | 3 | 0 | Demo stuff in here

Russ McK Kendrick 5:42 PM
@bot terminate all servers

bot APP 5:42 PM
awsremove is building. I'll let you know when it's done.

Today

new messages

Jenkins APP 5:43 PM
Checking number of running instances

I found you have 1 instances, terminating them now

Instances terminated (2 MB)

bot APP 5:44 PM
@RussMcK Kendrick, job https://[REDACTED]/job/awsremove/22/ finished with status:
SUCCESS.

+ Message #demo

So, what did I use for this? As already mentioned, for a start, I used Hubot. Hubot is an open source extendable chatbot developed by GitHub. It was

configured using the `hubot-slack` plugin in my Slack channel, and it listened out for any commands it was given.

I used the `hubot-alias` plugin to define an alias that translated *@bot give me a linux server* into `build awsLaunch OS=linux`; this used the `hubot-yardmaster` plugin to trigger to a build in my Jenkins installation.

Jenkins is an open source automation server, used mostly for continuous integration and also continuous delivery—it too has a plugin architecture. Using the Jenkins Ansible plugin and also the Jenkins Git plugin, I was able to pull the playbook and roles used to launch the AWS instance to my Jenkins server and then have Jenkins run the playbook for me—the playbook itself wasn't too dissimilar from the playbook we worked through in [chapter 9, Building Out a Cloud Network](#), and [chapter 10, Highly Available Cloud Deployments](#).

The playbook had a little logic built into it that limited the number of instances that could be launched, randomized the name of the instance that it was launching, and also displayed a random GIF from a list of several options—all of this information, along with details of the instance and the AMI, were all passed to the user via the Ansible Slack module to give the impression that the playbook was actually doing more than it was.

In the two preceding examples, the bot user is Hubot, and Jenkins is actually the feedback from the playbook run.

Automated deployment

Another example—I recently worked with several developers who needed a way to automatically deploy their code to both their development and staging servers. Using a combination of Docker, GitHub, Jenkins, and Ansible AWX, I was able to provide the developers with a workflow that was triggered every time they pushed code to either the development or staging branches of their repository on GitHub.

To achieve this, I deployed the code on their own Jenkins server, using Ansible to deploy Jenkins in a container, and also deployed AWX using Docker on the same server. Then, using the **Jenkins GitHub** plugin, I connected the Jenkins projects to GitHub to create the Webhooks needed to trigger the build. Then using the **Jenkins Ansible Tower** plugin, I had Jenkins trigger a playbook run in AWX.

I did this because at the moment, AWX does not hook in that easily with GitHub Webhooks, whereas **Jenkins** and the **Jenkins GitHub** plugin have a great level of compatibility—I imagine with the rate at which AWX is being developed, this slight niggle will be ironed out at some point soon.

As AWX allows you to grant role-based access to playbooks, I gave the development manager and operations engineers access to run the production playbook, and the developers have read-only access so that they can review the results of the playbook run.

This means that the deployment to production was also able to be automated, be it that, someone with the correct permissions had to trigger the playbook run manually.

The level of control which AWX allows us to have of who could trigger deployments, fitted it in with our existing deployment strategy which stated that the developers should not have access to production systems to deploy code which they had written.

Summary

Now we have come to the end of not only the chapter but also our book. I have been trying to think of a way I can sum up Ansible, and I have managed to find it in a tweet (<https://twitter.com/laser1llama/status/976135074117808129>) from Ansible creator Michael DeHaan, who, in response to a technical recruiter, said the following: "Anyone using Ansible for a few months is as good as anyone using Ansible for three years. It's a simple tool on purpose."

That perfectly sums up my experience of Ansible and hopefully yours. Once you know the basics, it is very easy to quickly move on and start to build more and more complex playbooks, which can not only assist with deploying basic code and applications, but also with deploying complex cloud and even physical architectures.

Being able to not only reuse your own roles but have access to a large collection of a community-contributed roles via Ansible Galaxy means you have many examples or quick starting points for your next project. So, you can roll your sleeves up and get stuck in a lot sooner than maybe you would with other tools. Also, if there is something Ansible cannot do, then odds are there is a tool it can be integrated with to provide the missing functionality.

Going back to what we discussed back in [Chapter 1, An Introduction to Ansible](#), being able to define your infrastructure and deployment in code in a repeatable and shareable way that encourages others to contribute to your playbooks should really be the ultimate aim of why you would start to introduce Ansible into your day-to-day workflows. I hope that, through this book, you have started to think of day-to-day tasks where Ansible could help you and save you time.

Further reading

More information on the tools mentioned in this chapter can be found at the following URLs:

- **Hubot:** <https://hubot.github.com>
- **Hubot Slack:** <https://github.com/slackapi/hubot-slack>
- **Hubot Alias:** <https://github.com/dtaniwaki/hubot-alias>
- **Hubot Yardmaster:** <https://github.com/hacklanta/hubot-yardmaster>
- **Jenkins Git:** <https://plugins.jenkins.io/git>
- **Jenkins Ansible:** <https://plugins.jenkins.io/ansible>
- **Jenkins GitHub:** <https://plugins.jenkins.io/github>
- **Jenkins Ansible Tower:** <https://plugins.jenkins.io/ansible-tower>

Assessments

Chapter 2, Installing and Running Ansible

1. What is the command to install Ansible using pip?

```
sudo -H pip install ansible
```

2. True or false: You can choose exactly which version of Ansible to install or roll back to when using Homebrew.

False

3. True or false: The Windows Subsystem for Linux runs in a virtual machine.

False

4. Name three hypervisors that are supported by Vagrant.

VirtualBox, VMware, and Hyper-V

5. State and explain what a host inventory is.

A host inventory is a list of hosts alongwith options for accessing them which Ansible will target

6. True or false: Indentation in YAML files is extremely important to their execution and isn't just cosmetic.

True

Chapter 3, The Ansible Commands

1. Of the commands that provide information about your host inventory that we have covered in this chapter, which ships with Ansible by default?

The `ansible-inventory` command

2. True or false: Variable files that have strings encrypted with Ansible Vault will work with versions of Ansible lower than 2.4.

False

3. What command would you run to get an example of how you should call the `yum` module as part of your task?

You would use the `ansible-doc` command

4. Explain why you would want to run single modules against hosts within your inventory.

You would use a single module if you want to use Ansible to run an ad hoc command against several hosts in a controlled way.

Chapter 4, Deploying a LAMP Stack

1. Which Ansible module would you use to download and unarchive a zip file?

The module is called `unarchive`

2. True or false: The variables found in the `roles/rolename/default/` folder override all other references of the same variable.

False

3. Explain how you would add a second user to our playbook?

By adding a second line to the users variable, for example: `{ name: "user2", group: "lamp", state: "present", key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}" }`

4. True or false: You can only call a single handler from a task.

False

Chapter 5, Deploying WordPress

1. Which fact gathered during the `setup` module execution can we use to tell our playbook how many processors our target host has?

The fact is `ansible_processor_count`

2. True or false: Using `backref` in the `lineinfile` module makes sure that no changes are applied if the regular expression is not matched.

True

3. Explain why we would want to build logic into the playbook to check whether WordPress is already installed.

So that we can skip the task that downloads and installs WordPress the next time the playbook is run.

4. Which module do we use to define variables as part of a playbook run?

The `set_fact` module

5. Which argument do we pass to the `shell` module to have the command we want to run executed in a directory of our choosing?

The argument is `chdir`

6. True or false: Setting MariaDB to bind to `127.0.0.1` will allow us to access it externally.

False

Chapter 6, Targeting Multiple Distributions

1. True or false: We need to double-check every task in our playbook, so it will work on both operating systems.

True

2. Which configuration option allows us to define the path to Python that, Ansible will use?

The option is `ansible_python_interpreter`

3. Explain why we need to make changes to the tasks that are configured and interact with the PHP-FPM service.

The path to the configuration files is different and also PHP-FPM runs under a different group by default on Ubuntu

4. True or false: The package names for each of the operating systems correspond exactly.

False

Chapter 7, The Core Network Modules

1. True or False: You have to use `with_items` with a `for` loop within a template.

False

2. Which character is used to split your variable over multiple lines?

You would use the `|` character

3. True or False: When using the VyOS module, we do not need to pass details of our device in the host inventory file.

True

Chapter 8, Moving to the Cloud

1. What is the name of the Python module we need to install to support the `digital_ocean` module?

The module is called `dopy`

2. True or false: You should always encrypt sensitive values such as the DigitalOcean personal access token.

True

3. Which filter are we using to find the ID of the SSH key we need to launch our Droplet with?

The filter will be `[?name=='Ansible']`

4. State and explain why we used the `unique_name` option in the `digital_ocean` task.

To ensure we do not launch multiple droplets with the same name with each playbook run.

5. What is the correct syntax for accessing variables from another Ansible host?

Using `hostvars`, for example using `{{ hostvars['localhost'].droplet_ip }}`, which has been registered on the Ansible Controller.

6. True or false: The `add_server` module is used to add our Droplet to the host group.

False

Chapter 9, Building Out a Cloud Network

1. Which two environment variables are used by AWS modules to read your access ID and secret?

They are `AWS_ACCESS_KEY` and `AWS_SECRET_KEY`

2. True or false: Every time you run the playbook, you will get a new VPC.

False

3. State and explain why we are not bothering to register the results of creating subnets.

So that we can group together a list of subnet IDs by the role we have assigned them later in the playbook run

4. What is the difference between using `cidr_ip` and `group_id` when defining a rule in a security group?

`cidr_ip` creates a rule that locks the supplied port down to a certain IP address where as `group_id` locks the post down to all hosts that are in the `group_id` you supply

5. True or false: The order in which security groups are added when using rules that have `group_id` defined doesn't matter.

False

Chapter 10, Highly Available Cloud Deployments

1. What is the name of the variable that is registered using the `gather_facts` option, which contains the date and time our playbook was executed?

It is the `ansible_date_time` fact

2. True or false: Ansible automatically figures out which task it needs to execute, meaning we don't have to define any logic ourselves.

False

3. Explain why we have to use the `local_action` module.

Because we do not want to interact with the AWS API from the host we are targeting with Ansible; instead, we want all AWS API interaction to take place from our Ansible Controller

4. Which command do we prepend to our `ansible-playbook` command to record how long our command took to execute?

The `time` command

5. True or false: When using autoscaling, you have to manually launch EC2 instances.

False

Chapter 11, Building Out a VMware Deployment

1. Which Python module do you need to install on your Ansible controller to be able to interact with vSphere?

The module is called PyVmomi

2. True or false: `vmware_dns_config` only allows you to set DNS resolvers on your ESXi hosts.

False

3. Name two of the modules we have covered that can be used to launch virtual machines; there are three, but one is deprecated.

The `vca_vapp` and `vmware_guest` modules; it is the `vsphere_guest` module which has been deprecated

4. Which of the modules we have looked at would you use to ensure that a virtual machine is fully available before progressing to a task that interacts with the VM via VMware?

The `vmware_guest_tools_wait` module

5. True or false: It is possible to schedule changing a power state using Ansible.

True

Chapter 12, Ansible Windows Modules

1. Which of the following two modules can be used on both a Windows and Linux host: setup, or file?

The `setup` module

2. True or false: You can use SSH to access your Windows target.

False

3. Explain the type of interface WinRM uses.

WinRM uses a SOAP interface rather than an interactive shell

4. Which Python module do you need to install to be able to interact with WinRM on macOS and Linux?

The `pywinrm` module

5. True or false: You can have a separate task to install Chocolatey before you use the `win_chocolatey` module.

False

Chapter 13, Hardening Your Servers Using Ansible and OpenSCAP

1. What effect does adding `>` to a multiline variable have?

The variable will be rendered as a single line when Ansible inserts it into the playbook run

2. True or false: OpenSCAP is certified by NIST.

True

3. Why are we telling Ansible to continue if the `scan` command is marked as failed?

Because the task will always fail if it doesn't get a 100% score

4. Explain why we are using tags for certain roles.

So that we can run certain parts of the playbook when we use the `--tags` flag

5. True or false: We use the `copy` command to copy HTML reports from the remote host to the Ansible controller.

False

Chapter 14, Deploying WPScan and OWASP ZAP

1. Why are we using Docker rather than installing WPScan and OWASP ZAP directly on our Vagrant box?

To simplify the deployment process; it is easier to deploy two containers than it is to install the support software stack for both tools

2. True or false: `pip` is installed on our Vagrant box by default.

False

3. What is the name of the Python module we need to install for Ansible Docker modules to function?

The `docker` module

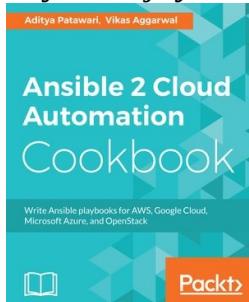
Chapter 15, Introducing Ansible Tower and Ansible AWX

1. State and explain the differences between Ansible Tower and Ansible AWX.

Ansible Tower is a commercially-supported, enterprise-grade piece of software supplied by Red Hat. Ansible AWX is the open source upstream for future versions of Ansible Tower; it is updated often and is supplied as-is.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

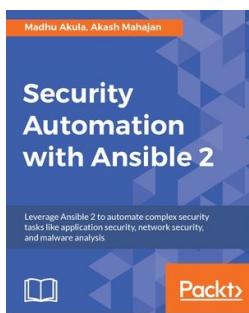


Ansible 2 Cloud Automation Cookbook

Aditya Patawari, Vikas Aggarwal

ISBN: 978-1-78829-582-6

- Use Ansible Vault to protect secrets
- Understand how Ansible modules interact with cloud providers to manage resources
- Build cloud-based resources for your application
- Create resources beyond simple virtual machines
- Write tasks that can be reused to create resources multiple times
- Work with self-hosted clouds such as OpenStack and Docker
- Deploy a multi-tier application on various cloud providers



Security Automation with Ansible 2

Madhu Akula, Akash Mahajan

ISBN: 978-1-78839-451-2

- Use Ansible playbooks, roles, modules, and templating to build generic, testable playbooks
- Manage Linux and Windows hosts remotely in a repeatable and predictable manner
- See how to perform security patch management, and security hardening with scheduling and automation
- Set up AWS Lambda for a serverless automated defense
- Run continuous security scans against your hosts and automatically fix and harden the gaps
- Extend Ansible to write your custom modules and use them as part of your already existing security automation programs
- Perform automation security audit checks for applications using Ansible
- Manage secrets in Ansible using Ansible Vault

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!