

# Mastering Ansible

**Fourth Edition**

---

Automate configuration management and overcome deployment challenges with Ansible

James Freeman | Jesse Keating



# **Mastering Ansible**

## **Fourth Edition**

Automate configuration management and overcome deployment challenges with Ansible

**James Freeman**

**Jesse Keating**

**Packt**

BIRMINGHAM—MUMBAI

# Mastering Ansible Fourth Edition

Copyright © 2021 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 authors, 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.

**Group Product Manager:** Rahul Nair

**Publishing Product Manager:** Meeta Rajani

**Senior Editor:** Sangeeta Purkayastha

**Content Development Editor:** Nihar Kapadia

**Technical Editor:** Shruthi Shetty

**Copy Editor:** Safis Editing

**Project Coordinator:** Shagun Saini

**Proofreader:** Safis Editing

**Indexer:** Manju Arasan

**Production Designer:** Jyoti Chauhan

First published: November 2015

Second edition: March 2017

Third edition: March 2019

Fourth edition: November 2021

Production reference: 1271021

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80181-878-0

[www.packtpub.com](http://www.packtpub.com)

*To Corinne Woolley for helping me to see who I really am, for her care,  
continued support, and presence in my world. To Ray and Mary, my  
beloved grandparents, always in my heart. To Ken – for teaching me  
to take life in my stride.*

*– James Freeman*

# Contributors

## About the author

**James Freeman** is an accomplished IT consultant with over 20 years' experience in the technology industry. He has more than 8 years of first-hand experience solving real-world enterprise problems in production environments using Ansible, frequently introducing Ansible as a new technology to businesses and CTOs. In addition, he has authored and facilitated bespoke Ansible workshops and training sessions and has presented at both international conferences and meetups on Ansible.

*So many people have made this possible and I would like to thank each and every one of them for their love and support, especially Neeshia Jasmara.*

**Jesse Keating** is an accomplished Ansible user, contributor, and presenter. He has been an active member of the Linux and open source community for over 15 years. He has firsthand experience involving a variety of IT activities, software development, and large-scale system administration. He has presented at numerous conferences and meetups, and has written many articles on a variety of topics.

## About the reviewer

**Mario Vázquez** is a software engineer passionate about container technologies, automation, and hybrid cloud. He has been working with Ansible since the early days and has always had Ansible in his toolbelt. These days, Mario is helping partners and customers to move their workloads to Kubernetes across multiple infrastructure providers.



# Table of Contents

## Preface

---

## Section 1: Ansible Overview and Fundamentals

### 1

#### The System Architecture and Design of Ansible

---

Technical requirements	4	Play and task names	30
Ansible versions and configurations	4	Module transport and execution	33
Inventory parsing and data sources	6	The module reference	33
Static inventories	6	Module arguments	34
Inventory ordering	8	Module blacklisting	36
Inventory variable data	9	Module transport and execution	37
Dynamic inventories	12	Variable types and location	38
Runtime inventory additions	17	Variable types	38
Inventory limiting	18	Magic variables	40
Playbook parsing	21	Accessing external data	42
The order of operations	22	Variable precedence	42
Relative path assumptions	24	Precedence order	43
Play behavior directives	27	Variable group priority ordering	44
Execution strategies	28	Merging hashes	48
The host selection for plays and tasks	29	Summary	49
		Questions	49

## 2

### Migrating from Earlier Ansible Versions

---

Technical requirements	54	The anatomy of an Ansible collection	62
Changes in Ansible 4.3	54	Installing additional modules with ansible-galaxy	69
Ansible Content Collections	55		
Upgrading from earlier Ansible installations	57	How to port legacy playbooks to Ansible 4.3 (a primer)	73
Uninstalling Ansible 3.0 or older	57	Summary	77
Installing Ansible from scratch	58	Questions	78
What are Ansible Collections?	61		

## 3

### Protecting Your Secrets with Ansible

---

Technical requirements	82	Executing ansible-playbook with encrypted files	96
Encrypting data at rest	82		
Vault IDs and passwords	83	Mixing encrypted data with plain YAML	98
Things Vault can encrypt	85		
Creating and editing encrypted files	86	Protecting secrets while operating	102
Encrypting existing files	90	Secrets transmitted to remote hosts	103
Editing encrypted files	92	Secrets logged to remote or local files	103
Password rotation on encrypted files	93		
Decrypting encrypted files	95	Summary	105
		Questions	106

## 4

### Ansible and Windows – Not Just for Linux

---

Technical requirements	110	Installing Linux under WSL	112
Running Ansible from Windows	110	Setting up Windows hosts for Ansible control using WinRM	115
Checking your build	111		
Enabling WSL	112	System requirements for automation with Ansible using WinRM	115

Enabling the WinRM listener	116	Setting up Windows hosts for Ansible control using OpenSSH	128
Connecting Ansible to Windows using WinRM	119	Automating Windows tasks with Ansible	131
<b>Handling Windows authentication and encryption when using WinRM</b>	<b>121</b>	Picking the right module	132
Authentication mechanisms	122	Installing software	134
A note on accounts	125	Extending beyond modules	135
Certificate validation over WinRM	126	<b>Summary</b>	135
		<b>Questions</b>	136

## 5

### **Infrastructure Management for Enterprises with AWX**

---

Technical requirements	140	Organizations	164
Getting AWX up and running	140	Scheduling	164
<b>Integrating AWX with your first playbook</b>	<b>149</b>	Auditing	166
Defining a project	151	Surveys	167
Defining an inventory	152	Workflow templates	169
Defining credentials	156	Notifications	170
Defining a template	158	Using the API	171
<b>Going beyond the basics</b>	<b>163</b>	<b>Summary</b>	173
Role-based access control (RBAC)	163	<b>Questions</b>	173

## **Section 2:**

### **Writing and Troubleshooting Ansible Playbooks**

## 6

### **Unlocking the Power of Jinja2 Templates**

---

Technical requirements	180	Loops	185
Control structures	180	Macros	191
Conditionals	180	<b>Data manipulation</b>	201

Syntax	201	Comparisons	217
Useful built-in filters	203	Logic	218
Useful Ansible provided custom filters	205	Tests	218
Omitting undefined arguments	214	<b>Summary</b>	220
Python object methods	215	<b>Questions</b>	220
<b>Comparing values</b>	<b>217</b>		

## 7

### Controlling Task Conditions

---

Technical requirements	224	<b>Error recovery</b>	238
<b>Defining a failure</b>	<b>224</b>	Using the rescue section	239
Ignoring errors	224	Using the always section	241
Defining an error condition	226	Handling unreliable environments	245
<b>Defining a change</b>	<b>232</b>	<b>Iterative tasks with loops</b>	248
Special handling of the command family	234	<b>Summary</b>	253
Suppressing a change	237	<b>Questions</b>	254

## 8

### Composing Reusable Ansible Content with Roles

---

Technical requirements	258	<b>Roles (structures, defaults, and dependencies)</b>	287
<b>Task, handler, variable, and playbook inclusion concepts</b>	<b>258</b>	Role structure	287
Including tasks	259	Role dependencies	291
Task inclusions with loops	271	Role application	294
Including handlers	274	Role sharing	300
Including variables	277	<b>Summary</b>	306
Including playbooks	284	<b>Questions</b>	306

## 9

### Troubleshooting Ansible

---

Technical requirements	310	Verbosity	310
Playbook logging and verbosity	310	Logging	311

---

<b>Variable introspection</b>	<b>311</b>	Playbook debugging	<b>320</b>
Variable subelements	315	Debugging local code	322
<b>Debugging code execution</b>	<b>319</b>	<b>Summary</b>	<b>339</b>
		<b>Questions</b>	<b>339</b>

## 10

### Extending Ansible

---

<b>Technical requirements</b>	<b>344</b>	Action plugins	<b>368</b>
<b>Developing modules</b>	<b>344</b>	Distributing plugins	<b>368</b>
The basic module construct	345	<b>Developing dynamic inventory plugins</b>	<b>369</b>
Custom modules	345	Listing hosts	370
Example – simple module	346	Listing host variables	370
<b>Developing plugins</b>	<b>360</b>	Simple inventory plugin	<b>371</b>
Connection-type plugins	360	<b>Contributing to the Ansible project</b>	<b>379</b>
Shell plugins	361	Contribution submissions	379
Lookup plugins	361	<b>Summary</b>	<b>387</b>
Vars plugins	361	<b>Questions</b>	<b>388</b>
Fact-caching plugins	361		
Filter plugins	362		
Callback plugins	363		

## Section 3: Orchestration with Ansible

## 11

### Minimizing Downtime with Rolling Deployments

---

<b>Technical requirements</b>	<b>394</b>	<b>Minimizing disruptions</b>	<b>410</b>
<b>In-place upgrades</b>	<b>394</b>	Delaying a disruption	<b>410</b>
<b>Expanding and contracting</b>	<b>398</b>	Running destructive tasks only once	<b>415</b>
<b>Failing fast</b>	<b>401</b>	<b>Serializing single tasks</b>	<b>417</b>
The any_errors_fatal option	401	<b>Summary</b>	<b>420</b>
The max_fail_percentage option	404	<b>Questions</b>	<b>420</b>
Forcing handlers	407		

## 12

### Infrastructure Provisioning

---

Technical requirements	424	Building images	451
Managing an on-premise cloud infrastructure	424	Building containers without a Dockerfile	456
Creating servers	425	Docker inventory	461
Using OpenStack inventory sources	435	<b>Building containers with Ansible</b>	465
Managing a public cloud infrastructure	442	Summary	471
Interacting with Docker containers	451	Questions	472

## 13

### Network Automation

---

Technical requirements	476	Working with the cli_command module	484
Ansible for network management	477	Configuring Arista EOS switches with Ansible	487
Cross-platform support	477	<b>Configuring Cumulus Networks switches with Ansible</b>	492
Configuration portability	478	Defining our inventory	492
Backup, restore, and version control	478	Practical examples	494
Automated change requests	479		
<b>Handling multiple device types</b>	480	<b>Best practices</b>	499
Researching your modules	481	Summary	505
Configuring your modules	482	Questions	505
Writing your playbooks	483		

---

### Other Books You May Enjoy

---

### Index

---

# Preface

Welcome to *Mastering Ansible*, your fully updated guide to the most valuable advanced features and functionalities provided by Ansible—the automation and orchestration tool. This book will provide you with the knowledge and skills required to truly understand how Ansible functions at a fundamental level, including all the latest features and changes since the release of version 3.0. In turn, this will allow you to master the advanced capabilities needed to tackle the complex automation challenges of today and the future. You will gain knowledge of Ansible workflows, explore use cases for advanced features, troubleshoot unexpected behavior, extend Ansible through customization, and learn about many of the new and important developments in Ansible, especially around infrastructure and network provisioning.

## Who this book is for

This book is for Ansible developers and operators who have an understanding of the core elements and applications but are now looking to enhance their skills in applying automation using Ansible.

## What this book covers

*Chapter 1, The System Architecture and Design of Ansible*, looks at the ins and outs of how Ansible goes about performing tasks on behalf of an engineer, how it is designed, and how to work with inventory and variables.

*Chapter 2, Migrating from Earlier Ansible Versions*, explains the architectural changes you will experience when you migrate from Ansible 2.x to any version from 3.x onward, how to work with Ansible collections, and also how to build your own—essential reading for anyone familiar with earlier Ansible versions.

*Chapter 3, Protecting Your Secrets with Ansible*, explores the tools available to encrypt data at rest and prevent secrets from being revealed at runtime.

*Chapter 4, Ansible and Windows – Not Just for Linux*, explores the integration of Ansible with Windows hosts to enable automation in cross-platform environments.

*Chapter 5, Infrastructure Management for Enterprises with AWX*, provides an overview of the powerful, open source graphical management framework for Ansible known as AWX, and how this might be employed in an enterprise environment.

*Chapter 6, Unlocking the Power of Jinja2 Templates*, states the varied uses of the Jinja2 templating engine within Ansible and discusses ways to make the most of its capabilities.

*Chapter 7, Controlling Task Conditions*, explains how to change the default behavior of Ansible to customize task error and change conditions.

*Chapter 8, Composing Reusable Ansible Content with Roles*, explains how to move beyond executing loosely organized tasks on hosts, and instead build clean, reusable, and self-contained code structures known as roles to achieve the same end result.

*Chapter 9, Troubleshooting Ansible*, takes you through the various methods that can be employed to examine, introspect, modify, and debug the operations of Ansible.

*Chapter 10, Extending Ansible*, covers the various ways in which new capabilities can be added to Ansible via modules, plugins, and inventory sources.

*Chapter 11, Minimizing Downtime with Rolling Deployments*, explains the common deployment and upgrade strategies to showcase the relevant Ansible features.

*Chapter 12, Infrastructure Provisioning*, examines cloud infrastructure providers and container systems for creating an infrastructure to manage.

*Chapter 13, Network Automation*, describes the advancements in the automation of network device configuration using Ansible.

## To get the most out of this book

To follow the examples provided in this book, you will need access to a computer platform capable of running Ansible. Currently, Ansible can be run on any machine with Python 2.7 or Python 3 (versions 3.5 and higher) installed (Windows is supported for the control machine, but only through a Linux distribution running in the **Windows Subsystem for Linux (WSL)** layer available on newer versions—see *Chapter 4, Ansible and Windows – Not Just for Linux*, for details). Operating systems supported include (but are not limited to) Red Hat, Debian, Ubuntu, CentOS, macOS, and FreeBSD.

This book uses the Ansible 4.x.x series release. Ansible installation instructions can be found at [https://docs.ansible.com/ansible/latest/installation\\_guide/intro\\_installation.html](https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html).

Some examples use Docker version 20.10.8. Docker installation instructions can be found at <https://docs.docker.com/get-docker/>.

A handful of examples in this book make use of accounts on both **Amazon Web Services (AWS)** and Microsoft Azure. More information about these services may be found at <https://aws.amazon.com/> and <https://azure.microsoft.com>, respectively. We also delve into the management of OpenStack with Ansible, and the examples in this book were tested against a single *all-in-one* instance of DevStack as per the instructions found here: <https://docs.openstack.org/devstack/latest/>.

Finally, *Chapter 13, Network Automation*, makes use of Arista vEOS 4.26.2F and Cumulus VX version 4.4.0 in the example code—please see here for more information: <https://www.arista.com/en/support/software-download> and <https://www.nvidia.com/en-gb/networking/ethernet-switching/cumulus-vx/>. If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

## Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition>. If there's an update to the code, it will be updated in the 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!

## Code in Action

The Code in Action videos for this book can be viewed at <https://bit.ly/3vvkzbP>.

## Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: [https://static.packt-cdn.com/downloads/9781801818780\\_ColorImages.pdf](https://static.packt-cdn.com/downloads/9781801818780_ColorImages.pdf).

## Conventions used

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

**Code in text:** Indicates code words in the text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "This book will assume that there are no settings in the `ansible.cfg` file that would affect the default operation of Ansible"

A block of code is set as follows:

```
---
```

```
plugin: amazon.aws.aws_ec2
boto_profile: default
```

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

```
ansible-playbook -i mastery-hosts --vault-id
test@./password.sh showme.yaml -v
```

**Bold:** Indicates a new term, an important word, or words that you see on screen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "You simply need to navigate to your profile preferences page and click the **Show API Key** button."

**Tips or important notes**

Appear like this.

## Get in touch

Feedback from our readers is always welcome.

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

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packtpub.com/support/errata](http://www.packtpub.com/support/errata) and fill in the form.

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

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

## Share Your Thoughts

Once you've read *Mastering Ansible Fourth Edition*, we'd love to hear your thoughts! Please click [here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.



# Section 1: Ansible Overview and Fundamentals

In this section, we will explore the fundamentals of how Ansible works and establish a sound basis on which to develop playbooks and workflows. We will also examine and explain the changes you will discover if you are familiar with the older Ansible 2.x releases.

The following chapters are included in this section:

- *Chapter 1, The System Architecture and Design of Ansible*
- *Chapter 2, Migrating from Earlier Ansible Versions*
- *Chapter 3, Protecting Your Secrets with Ansible*
- *Chapter 4, Ansible and Windows – Not Just for Linux*
- *Chapter 5, Infrastructure Management for Enterprises with AWX*



# 1

# The System Architecture and Design of Ansible

This chapter provides a detailed exploration of the architecture and design of **Ansible** and how it goes about performing tasks on your behalf. We will cover the basic concepts of inventory parsing and how data is discovered. Then, we will proceed onto playbook parsing. We will take a walk through module preparation, transportation, and execution. Finally, we will detail variable types and find out where variables are located, their scope of use, and how precedence is determined when variables are defined in more than one location. All these things will be covered in order to lay the foundation for mastering Ansible!

In this chapter, we will cover the following topics:

- Ansible versions and configurations
- Inventory parsing and data sources
- Playbook parsing
- Execution strategies
- Module transport and execution

- Ansible collections
- Variable types and locations
- Magic variables
- Accessing external data
- Variable precedence (and interchanging this with variable priority ordering)

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running **Ansible 4.3** or later. Almost any flavor of Linux should do. For those who are interested in specifics, all the code presented in this chapter was tested on **Ubuntu Server 20.04 LTS**, unless stated otherwise, and on Ansible 4.3. The example code that accompanies this chapter can be downloaded from GitHub at <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter01>.

Check out the following video to view the Code in Action: <https://bit.ly/3E37xpn>.

## Ansible versions and configurations

It is assumed that you have Ansible installed on your system. There are many documents out there that cover installing Ansible in a way that is appropriate to the operating system and version that you might be using. However, it is important to note that Ansible versions that are newer than 2.9.x feature some major changes from all of the earlier versions. For everyone reading this book who has had exposure to Ansible 2.9.x and earlier, *Chapter 2, Migrating from Earlier Ansible Versions*, explains the changes in detail, along with how to address them.

This book will assume the use of Ansible version 4.0.0 (or later), coupled with ansible-core 2.11.1 (or newer), both of which are required and are the latest and greatest releases at the time of writing. To discover the version in use on a system where Ansible is already installed, make use of the `--version` argument, that is, either `ansible` or `ansible-playbook`, as follows:

```
ansible-playbook --version
```

This command should give you an output that's similar to *Figure 1.1*; note that the screenshot was taken on Ansible 4.3, so you might see an updated version number corresponding to the version of your `ansible-core` package (for instance, for Ansible 4.3.0, this would be `ansible-core 2.11.1`, which is the version number that all of the commands will return):

```
jfreeman@mastery1:~$ ansible-playbook --version
ansible-playbook [core 2.11.3]
  config file = None
  configured module search path = ['/home/jfreeman/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
    ansible python module location = /usr/local/lib/python3.8/dist-packages/ansible
    ansible collection location = /home/jfreeman/.ansible/collections:/usr/share/ansible/collections
    executable location = /usr/local/bin/ansible-playbook
    python version = 3.8.10 (default, Jun 2 2021, 10:49:15) [GCC 9.4.0]
    jinja version = 3.0.1
    libyaml = True
jfreeman@mastery1:~$
```

Figure 1.1 – An example output showing the installed version of Ansible on a Linux system

#### Important note

Note that `ansible` is the executable for doing ad hoc one-task executions, and `ansible-playbook` is the executable that will process playbooks to orchestrate multiple tasks. We will cover the concepts of ad hoc tasks and playbooks later in the book.

The configuration for Ansible can exist in a few different locations, where the first file found will be used. The search involves the following:

- `ANSIBLE_CFG`: This environment variable is used, provided that it is set.
- `ansible.cfg`: This is located in the current working directory.
- `~/.ansible.cfg`: This is located in the user's home directory.
- `/etc/ansible/ansible.cfg`: The default central Ansible configuration file for the system.

Some installation methods could include placing a `config` file in one of these locations. Look around to check whether such a file exists and view what settings are in the file to get an idea of how the Ansible operation might be affected. This book assumes that there are no settings in the `ansible.cfg` file that can affect the default operation of Ansible.

## Inventory parsing and data sources

In Ansible, nothing happens without an inventory. Even ad hoc actions performed on the localhost require an inventory – although that inventory might just consist of the localhost. The inventory is the most basic building block of Ansible architecture. When executing `ansible` or `ansible-playbook`, an inventory must be referenced. Inventories are files or directories that exist on the same system that runs `ansible` or `ansible-playbook`. The location of the inventory can be defined at runtime with the `--inventory-file (-i)` argument or by defining the path in an Ansible config file.

Inventories can be static or dynamic, or even a combination of both, and Ansible is not limited to a single inventory. The standard practice is to split inventories across logical boundaries, such as staging and production, allowing an engineer to run a set of plays against their staging environment for validation, and then follow with the exact plays run against the production inventory set.

Variable data, such as specific details about how to connect to a particular host in your inventory, can be included, along with an inventory in a variety of ways, and we'll explore the options available to you.

## Static inventories

The static inventory is the most basic of all the inventory options. Typically, a static inventory will consist of a single file in `ini` format. Other formats are supported, including `YAML`, but you will find that `ini` is commonly used when most people start out with Ansible. Here is an example of a static inventory file describing a single host, `mastery.example.name`:

```
mastery.example.name
```

That is all there is to it. Simply list the names of the systems in your inventory. Of course, this does not take full advantage of all that an inventory has to offer. If every name were listed like this, all plays would have to reference specific hostnames, or the special built-in `all` group (which, as the name suggests, contains all hosts inside the inventory). This can be quite tedious when developing a playbook that operates across different environments within your infrastructure. At the very least, hosts should be arranged into groups.

A design pattern that works well is arranging your systems into groups based on expected functionality. At first, this might seem difficult if you have an environment where single systems can play many different roles, but that is perfectly fine. Systems in an inventory can exist in more than one group, and groups can even consist of other groups! Additionally, when listing groups and hosts, it is possible to list hosts without a group. These would have to be listed first before any other group is defined. Let's build on our previous example and expand our inventory with a few more hosts and groupings, as follows:

```
[web]
mastery.example.name

[dns]
backend.example.name

[database]
backend.example.name

[frontend:children]
web

[backend:children]
dns
database
```

Here, we have created a set of three groups with one system in each, and then two more groups, which logically group all three together. Yes, that's right: you can have groups of groups. The syntax used here is `[groupname:children]`, which indicates to Ansible's inventory parser that this group, going by the name of `groupname`, is nothing more than a grouping of other groups.

The `children`, in this case, are the names of the other groups. This inventory now allows writing plays against specific hosts, low-level role-specific groups, or high-level logical groupings, or any combination thereof.

By utilizing generic group names, such as `dns` and `database`, Ansible plays can reference these generic groups rather than the explicit hosts within. An engineer can create one inventory file that fills in these groups with hosts from a preproduction staging environment, and another inventory file with the production versions of these groupings. The content of the playbook does not need to change when executing on either a staging or production environment because it refers to the generic group names that exist in both inventories. Simply refer to the correct inventory to execute it in the desired environment.

## Inventory ordering

A new play-level keyword, `order`, was added to Ansible in version 2.4. Prior to this, Ansible processed the hosts in the order specified in the inventory file, and it continues to do so by default, even in newer versions. However, the following values can be set for the `order` keyword for a given play, resulting in the processing order of hosts, which is described as follows:

- `inventory`: This is the default option. It simply means that Ansible proceeds as it always has, processing the hosts in the order that is specified in the `inventory` file.
- `reverse_inventory`: This results in the hosts being processed in the reverse order that is specified in the `inventory` file.
- `sorted`: The hosts are processed in alphabetical order by name.
- `reverse_sorted`: The hosts are processed in reverse alphabetical order.
- `shuffle`: The hosts are processed in a random order, with the order being randomized on each run.

In Ansible, the alphabetical sorting used is alternatively known as lexicographical. Put simply, this means that values are sorted as strings, with the strings being processed from left to right. Therefore, let's say that we have three hosts: `mastery1`, `mastery11`, and `mastery2`. In this list, `mastery1` comes first as the character, as position 8 is a 1. Then comes `mastery11`, as the character at position 8 is still a 1, but now there is an additional character at position 9. Finally comes `mastery2`, as character 8 is a 2, and 2 comes after 1. This is important as, numerically, we know that 11 is greater than 2. However, in this list, `mastery11` comes before `mastery2`. You can easily work around this by adding leading zeros to any numbers on your hostnames; for example, `mastery01`, `mastery02`, and `mastery11` will be processed in the order they have been listed in this sentence, resolving the lexicographical issue described.

## Inventory variable data

Inventories provide more than just system names and groupings. Data regarding the systems can be passed along as well. This data could include the following:

- Host-specific data to use in templates
- Group-specific data to use in task arguments or conditionals
- Behavioral parameters to tune how Ansible interacts with a system

Variables are a powerful construct within Ansible and can be used in a variety of ways, not just those described here. Nearly every single thing done in Ansible can include a variable reference. While Ansible can discover data about a system during the setup phase, not all of the data can be discovered. Defining data with the inventory expands this. Note that variable data can come from many different sources, and one source could override another. We will cover the order of variable precedence later in this chapter.

Let's improve upon our existing example inventory and add to it some variable data. We will add some host-specific data and group-specific data:

```
[web]
mastery.example.name ansible_host=192.168.10.25

[dns]
backend.example.name

[database]
backend.example.name

[frontend:children]
web

[backend:children]
dns
database

[web:vars]
http_port=88
proxy_timeout=5
```

```
[backend:vars]
ansible_port=314

[all:vars]
ansible_ssh_user=otto
```

In this example, we defined `ansible_host` for `mastery.example.name` to be the IP address of `192.168.10.25`. The `ansible_host` variable is a **behavioral inventory variable**, which is intended to alter the way Ansible behaves when operating with this host. In this case, the variable instructs Ansible to connect to the system using the IP address provided, rather than performing a DNS lookup on the name using `mastery.example.name`. There are a number of other behavioral inventory variables that are listed at the end of this section, along with their intended use.

Our new inventory data also provides group-level variables for the web and backend groups. The web group defines `http_port`, which could be used in an **NGINX** configuration file, and `proxy_timeout`, which might be used to determine **HAProxy** behavior. The backend group makes use of another behavioral inventory parameter to instruct Ansible to connect to the hosts in this group using port 314 for SSH, rather than the default of 22.

Finally, a construct is introduced that provides variable data across all the hosts in the inventory by utilizing a built-in `all` group. Variables defined within this group will apply to every host in the inventory. In this particular example, we instruct Ansible to log in as the `otto` user when connecting to the systems. This is also a behavioral change, as the Ansible default behavior is to log in as a user with the same name as the user executing `ansible` or `ansible-playbook` on the control host.

Here is a list of behavior inventory variables and the behaviors they intend to modify:

- `ansible_host`: This is the DNS name or the Docker container name that Ansible will initiate a connection to.
- `ansible_port`: This specifies the port number that Ansible will use to connect to the inventory host if it is not the default value of 22.
- `ansible_user`: This specifies the username that Ansible will use to connect with the inventory host, regardless of the connection type.
- `ansible_password`: This is used to provide Ansible with the password for authentication to the inventory host in conjunction with `ansible_user`. Use this for testing purposes only – you should always use a vault to store sensitive data such as passwords (please refer to *Chapter 3, Protecting Your Secrets with Ansible*).

- `ansible_ssh_private_key_file`: This is used to specify which SSH private key file will be used to connect to the inventory host if you are not using the default one or `ssh-agent`.
- `ansible_ssh_common_args`: This defines SSH arguments to append to the default arguments for `ssh`, `sftp`, and `scp`.
- `ansible_sftp_extra_args`: This is used to specify additional arguments that will be passed to the `sftp` binary when called by Ansible.
- `ansible_scp_extra_args`: This is used to specify additional arguments that will be passed to the `scp` binary when called by Ansible.
- `ansible_ssh_extra_args`: This is used to specify additional arguments that will be passed to the `ssh` binary when called by Ansible.
- `ansible_ssh_pipelining`: This setting uses a Boolean to define whether SSH pipelining should be used for this host.
- `ansible_ssh_executable`: This setting overrides the path to the SSH executable for this host.
- `ansible_become`: This defines whether privilege escalation (`sudo` or something else) should be used with this host.
- `ansible_become_method`: This is the method to use for privilege escalation and can be one of `sudo`, `su`, `pbrun`, `pfexec`, `doas`, `dzdo`, or `ksu`.
- `ansible_become_user`: This is the user to switch to through privilege escalation, typically root on Linux and Unix systems.
- `ansible_become_password`: This is the password to use for privilege escalation. Only use this for testing purposes; you should always use a vault to store sensitive data such as passwords (please refer to *Chapter 3, Protecting Your Secrets with Ansible*).
- `ansible_become_exe`: This is used to set the executable that was used for the chosen escalation method if you are not using the default one defined by the system.
- `ansible_become_flags`: This is used to set the flags passed to the chosen escalation executable if required.
- `ansible_connection`: This is the connection type of the host. Candidates are `local`, `smart`, `ssh`, `paramiko`, `docker`, or `winrm` (we will look at this in more detail later in the book). The default setting is `smart` in any modern Ansible distribution (this detects whether the `ControlPersist` SSH feature is supported and, if so, uses `ssh` as the connection type; otherwise, it falls back to `paramiko`).

- `ansible_docker_extra_args`: This is used to specify the extra argument that will be passed to a remote Docker daemon on a given inventory host.
- `ansible_shell_type`: This is used to determine the shell type on the inventory host(s) in question. It defaults to the `sh`-style syntax but can be set to `csh` or `fish` to work with systems that use these shells.
- `ansible_shell_executable`: This is used to determine the shell type on the inventory host(s) in question. It defaults to the `sh`-style syntax but can be set to `csh` or `fish` to work with systems that use these shells.
- `ansible_python_interpreter`: This is used to manually set the path to Python on a given host in the inventory. For example, some distributions of Linux have more than one Python version installed, and it is important to ensure that the correct one is set. For example, a host might have both `/usr/bin/python2.7` and `/usr/bin/python3`, and this is used to define which one will be used.
- `ansible_*_interpreter`: This is used for any other interpreted language that Ansible might depend upon (for example, Perl or Ruby). This replaces the interpreter binary with the one that is specified.

## Dynamic inventories

A static inventory is great and can be enough for many situations. However, there are times when a statically written set of hosts is just too unwieldy to manage. Consider situations where inventory data already exists in a different system, such as **LDAP**, a cloud computing provider, or an in-house **configuration management database (CMDB)** (inventory, asset tracking, and data warehousing) system. It would be a waste of time and energy to duplicate that data and, in the modern world of on-demand infrastructure, that data would quickly grow stale or become disastrously incorrect.

Another example of when a dynamic inventory source might be desired is when your site grows beyond a single set of playbooks. Multiple playbook repositories can fall into the trap of holding multiple copies of the same inventory data, or complicated processes have to be created to reference a single copy of the data. An external inventory can easily be leveraged to access the common inventory data that is stored outside of the playbook repository to simplify the setup. Thankfully, Ansible is not limited to static inventory files.

A dynamic inventory source (or plugin) is an executable that Ansible will call at runtime to discover real-time inventory data. This executable can reach out to external data sources and return data, or it can just parse local data that already exists but might not be in the `.ini`/`.yaml` Ansible inventory format. While it is possible, and easy, to develop your own dynamic inventory source, which we will cover in a later chapter, Ansible provides an ever-growing number of example inventory plugins. This includes, but is not limited to, the following:

- OpenStack Nova
- Rackspace Public Cloud
- DigitalOcean
- Linode
- Amazon EC2
- Google Compute Engine
- Microsoft Azure
- Docker
- Vagrant

Many of these plugins require some level of configuration, such as user credentials for EC2 or an authentication endpoint for **OpenStack Nova**. Since it is not possible to configure additional arguments for Ansible to pass along to the inventory script, the configuration for the script must either be managed via an `.ini` config file that is read from a known location or environment variables that are read from the shell environment used to execute `ansible` or `ansible-playbook`. Also, note that, sometimes, external libraries are required for these inventory scripts to function.

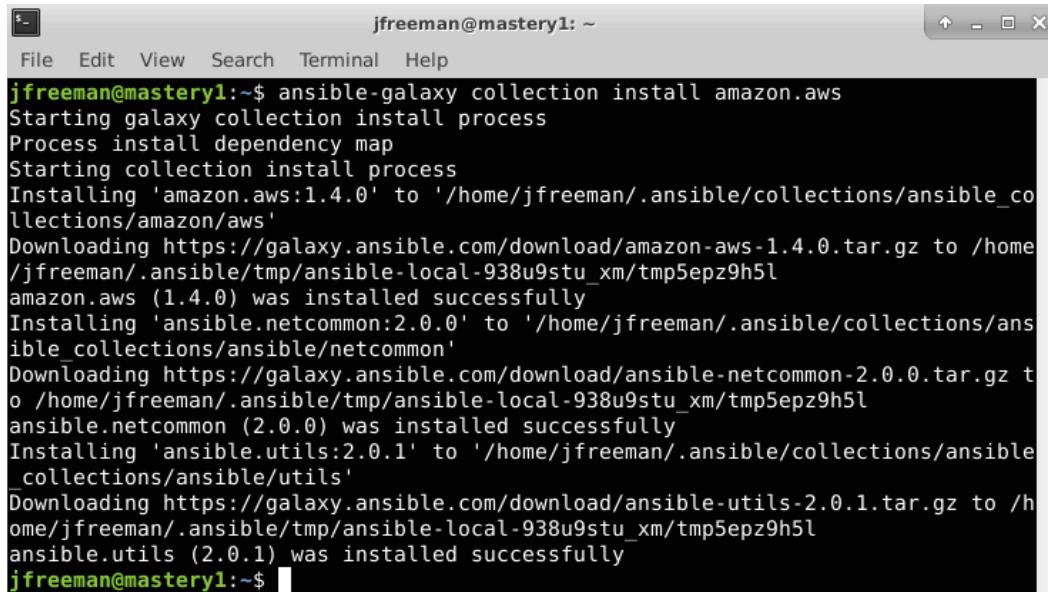
When `ansible` or `ansible-playbook` is directed at an executable file for an inventory source, Ansible will execute that script with a single argument, `--list`. This is so that Ansible can get a listing of the entire inventory in order to build up its internal objects to represent the data. Once that data is built up, Ansible will then execute the script with a different argument for every host in the data to discover variable data. The argument used in this execution is `--host <hostname>`, which will return any variable data that is specific to that host.

The number of inventory plugins is too numerous for us to go through each of them in detail in this book. However, similar processes are needed to set up and use just about all of them. So, to demonstrate the process, we will work through the use of the EC2 dynamic inventory.

Many of the dynamic inventory plugins are installed as part of the `community.general` collection, which is installed, by default, when you install Ansible 4.0.0. Nonetheless, the first part of working with any dynamic inventory plugin is finding out which collection the plugin is part of and, if required, installing that collection. The EC2 dynamic inventory plugin is installed as part of the `amazon.aws` collection. So, your first step will be to install this collection – you can do this with the following command:

```
ansible-galaxy collection install amazon.aws
```

If all goes well, you should see a similar output on your Terminal to that in *Figure 1.2*:



A screenshot of a terminal window titled "jfreeman@mastery1: ~". The window shows the command `ansible-galaxy collection install amazon.aws` being run and its output. The output details the process of installing the collection, including dependency maps and individual component installations for `amazon.aws`, `ansible.netcommon`, and `ansible.utils`.

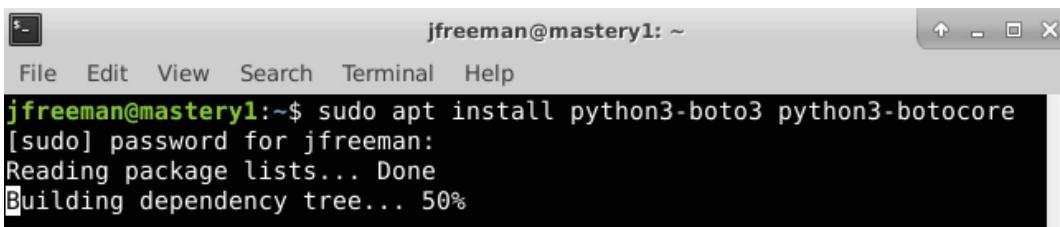
```
jfreeman@mastery1:~$ ansible-galaxy collection install amazon.aws
Starting galaxy collection install process
Process install dependency map
Starting collection install process
Installing 'amazon.aws:1.4.0' to '/home/jfreeman/.ansible/collections/ansible_collections/amazon/aws'
Downloading https://galaxy.ansible.com/download/amazon-aws-1.4.0.tar.gz to /home/jfreeman/.ansible/tmp/ansible-local-938u9stu_xm/tmp5epz9h5l
amazon.aws (1.4.0) was installed successfully
Installing 'ansible.netcommon:2.0.0' to '/home/jfreeman/.ansible/collections/ansible_collections/ansible/netcommon'
Downloading https://galaxy.ansible.com/download/ansible-netcommon-2.0.0.tar.gz to /home/jfreeman/.ansible/tmp/ansible-local-938u9stu_xm/tmp5epz9h5l
ansible.netcommon (2.0.0) was installed successfully
Installing 'ansible.utils:2.0.1' to '/home/jfreeman/.ansible/collections/ansible_collections/ansible/utils'
Downloading https://galaxy.ansible.com/download/ansible-utils-2.0.1.tar.gz to /home/jfreeman/.ansible/tmp/ansible-local-938u9stu_xm/tmp5epz9h5l
ansible.utils (2.0.1) was installed successfully
jfreeman@mastery1:~$
```

Figure 1.2 – The installation of the `amazon.aws` collection using `ansible-galaxy`

Whenever you install a new plugin or collection, it is always advisable to read the accompanying documentation as some of the dynamic inventory plugins require additional libraries or tools to function correctly. For example, if you refer to the documentation for the `aws_ec2` plugin at [https://docs.ansible.com/ansible/latest/collections/amazon/aws/aws\\_ec2\\_inventory.html](https://docs.ansible.com/ansible/latest/collections/amazon/aws/aws_ec2_inventory.html), you will see that both the `boto3` and `botocore` libraries are required for this plugin to operate. Installing this will depend on your operating system and Python environment. However, on Ubuntu Server 20.04 (and other Debian variants), it can be done with the following command:

```
sudo apt install python3-boto3 python3-botocore
```

Here's the output for the preceding command:

A screenshot of a terminal window titled "jfreeman@mastery1: ~". The window has a standard Linux-style title bar with icons for maximize, minimize, and close. Below the title bar is a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The main terminal area shows the command "sudo apt install python3-boto3 python3-botocore" being run. A password prompt "[sudo] password for jfreeman:" appears, followed by the output of the command: "Reading package lists... Done" and "Building dependency tree... 50%".

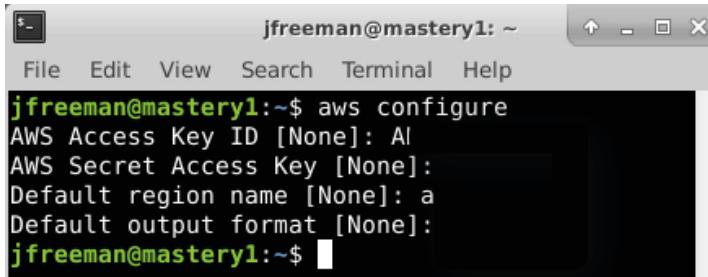
```
jfreeman@mastery1: ~
File Edit View Search Terminal Help
jfreeman@mastery1:~$ sudo apt install python3-boto3 python3-botocore
[sudo] password for jfreeman:
Reading package lists... Done
Building dependency tree... 50%
```

Figure 1.3 – Installing the Python dependencies for the EC2 dynamic inventory script

Now, looking at the documentation for the plugin (often, you can also find helpful hints by looking within the code and any accompanying configuration files), you will note that we need to provide our AWS credentials to this script in some manner. There are several possible ways in which to do this – one example is to use the `awscli` tool (if you have it installed) to define the configuration, and then reference this configuration profile from your inventory. For example, I configured my default AWS CLI profile using the following command:

```
aws configure
```

The output will appear similar to the following screenshot (the secure details have been redacted for obvious reasons!):



```
jfreeman@mastery1:~$ aws configure
AWS Access Key ID [None]: AI
AWS Secret Access Key [None]:
Default region name [None]: a
Default output format [None]:
jfreeman@mastery1:~$
```

Figure 1.4 – Configuring AWS credentials using the AWS CLI utility

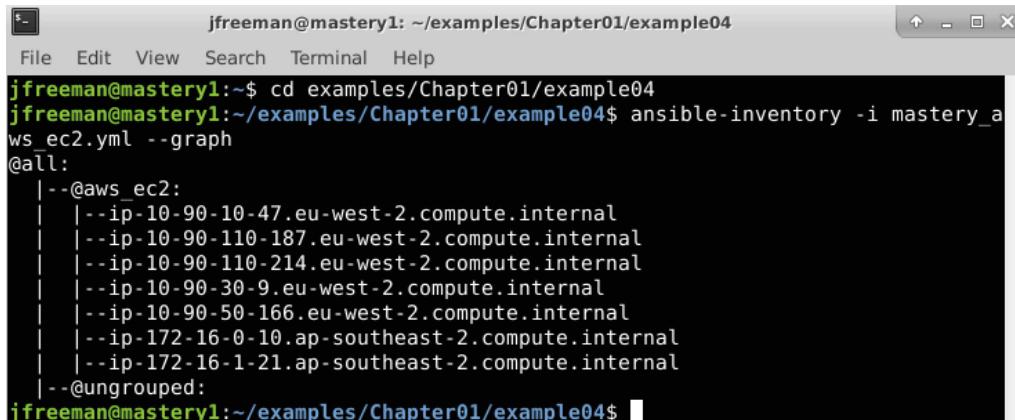
With this done, we can now create our inventory definition, telling Ansible which plugin to use, and passing the appropriate parameters to it. In our example here, we simply need to tell the plugin to use the default profile we created earlier. Create a file called `mastery_aws_ec2.yml`, which contains the following content:

```
---
plugin: amazon.aws.aws_ec2
boto_profile: default
```

Finally, we will test our new inventory plugin configuration by passing it to the `ansible-inventory` command with the `--graph` parameter:

```
ansible-inventory -i mastery_aws_ec2.yml --graph
```

Assuming you have some instances running in AWS EC2, you will see a similar output to the following:



```
jfreeman@mastery1:~/examples/Chapter01/example04
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter01/example04$ cd examples/Chapter01/example04
jfreeman@mastery1:~/examples/Chapter01/example04$ ansible-inventory -i mastery_aws_ec2.yml --graph
@all:
  |--@aws_ec2:
  |   |--ip-10-90-10-47.eu-west-2.compute.internal
  |   |--ip-10-90-110-187.eu-west-2.compute.internal
  |   |--ip-10-90-110-214.eu-west-2.compute.internal
  |   |--ip-10-90-30-9.eu-west-2.compute.internal
  |   |--ip-10-90-50-166.eu-west-2.compute.internal
  |   |--ip-172-16-0-10.ap-southeast-2.compute.internal
  |   |--ip-172-16-1-21.ap-southeast-2.compute.internal
  |--@ungrouped:
jfreeman@mastery1:~/examples/Chapter01/example04$
```

Figure 1.5 – An example output from the dynamic inventory plugin

Voila! We have a listing of our current AWS inventory, along with a glimpse into the automatic grouping performed by the plugin. If you want to delve further into the capabilities of the plugin and view, for example, all the inventory variables assigned to each host (which contain useful information, including instance type and sizing), try passing the `--list` parameter to `ansible-inventory` instead of `--graph`.

With the AWS inventory in place, you could use this right away to run a single task or the entire playbook against this dynamic inventory. For example, to use the `ansible.builtin.ping` module to check Ansible authentication and connectivity to all the hosts in the inventory, you could run the following command:

```
ansible -i mastery_aws_ec2.yml all -m ansible.builtin.ping
```

Of course, this is just one example. However, if you follow this process for other dynamic inventory providers, you should get them to work with ease.

In *Chapter 10, Extending Ansible*, we will develop our own custom inventory plugin to demonstrate how they operate.

## Runtime inventory additions

Just like static inventory files, it is important to remember that Ansible will parse this data once, and only once, per the `ansible` or `ansible-playbook` execution. This is a fairly common stumbling point for users of cloud dynamic sources, where, frequently, a playbook will create a new cloud resource and then attempt to use it as if it were part of the inventory. This will fail, as the resource was not part of the inventory when the playbook launched. All is not lost, though! A special module is provided that allows a playbook to temporarily add an inventory to the in-memory inventory object, that is, the `ansible.builtin.add_host` module.

This module takes two options: `name` and `groups`. The `name` option should be obvious; it defines the hostname that Ansible will use when connecting to this particular system. The `groups` option is a comma-separated list of groups that you can add to this new system. Any other option passed to this module will become the host variable data for this host. For example, if we want to add a new system, name it `newmastery.example.name`, add it to the `web` group, and instruct Ansible to connect to it by way of IP address `192.168.10.30`. This will create a task that resembles the following:

```
- name: add new node into runtime inventory
  ansible.builtin.add_host:
    name: newmastery.example.name
```

```
groups: web
ansible_host: 192.168.10.30
```

This new host will be available to use – either by way of the name provided or by way of the `web` group – for the rest of the `ansible-playbook` execution. However, once the execution has been completed, this host will not be available unless it has been added to the inventory source itself. Of course, if this were a new cloud resource that had been created, the next `ansible` or `ansible-playbook` execution that sourced a dynamic inventory from that cloud would pick up the new member.

## Inventory limiting

As mentioned earlier, every execution of `ansible` or `ansible-playbook` will parse the entire inventory it has been provided with. This is even true when a limit has been applied. Put simply, a limit is applied at runtime by making use of the `--limit` runtime argument to `ansible` or `ansible-playbook`. This argument accepts a pattern, which is essentially a mask to apply to the inventory. The entire inventory is parsed, and at each play, the limit mask that is supplied restricts the play to only run against the pattern that has been specified.

Let's take our previous inventory example and demonstrate the behavior of Ansible with and without a limit. If you recall, we have a special group, `all`, that we can use to reference all of the hosts within an inventory. Let's assume that our inventory is written out in the current working directory, in a file named `mastery-hosts`, and we will construct a playbook to demonstrate the host on which Ansible is operating. Let's write this playbook out as `mastery.yaml`:

```
---
- name: limit example play
  hosts: all
  gather_facts: false

  tasks:
    - name: tell us which host we are on
      ansible.builtin.debug:
        var: inventory_hostname
```

The `ansible.builtin.debug` module is used to print out text or values of variables. We'll use this module a lot in this book to simulate the actual work being done on a host.

Now, let's execute this simple playbook without supplying a limit. For simplicity's sake, we will instruct Ansible to utilize a local connection method, which will execute locally rather than attempt to SSH to these nonexistent hosts. Run the following command:

```
ansible-playbook -i mastery-hosts -c local mastery.yaml
```

The output should appear similar to *Figure 1.6*:

```
jfreeman@mastery1: ~/examples/Chapter01/example06
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter01/example06$ ansible-playbook -i mastery-hosts -c local mastery.yaml

PLAY [limit example play] ****
TASK [tell us which host we are on] ****
ok: [backend.example.name] => {
    "inventory_hostname": "backend.example.name"
}
ok: [mastery.example.name] => {
    "inventory_hostname": "mastery.example.name"
}

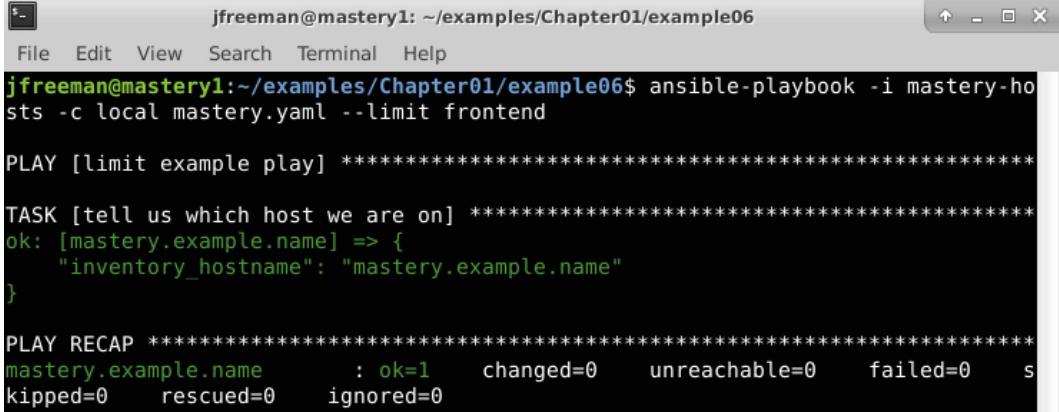
PLAY RECAP ****
backend.example.name      : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
mastery.example.name     : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
```

Figure 1.6 – Running the simple playbook on an inventory without a limit applied

As you can see, both the `backend.example.name` and `mastery.example.name` hosts were operated on. Now, let's see what happens if we supply a limit, that is, to limit our run to the frontend systems only, by running the following command:

```
ansible-playbook -i mastery-hosts -c local mastery.yaml --limit
frontend
```

This time around, the output should appear similar to *Figure 1.7*:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter01/example06". The window shows the command "ansible-playbook -i mastery-hosts -c local mastery.yaml --limit frontend" being run. The output shows a single host "mastery.example.name" being processed, with status "ok=1" and other metrics like changed=0, unreachable=0, failed=0, etc. The terminal interface includes standard file menu options (File, Edit, View, Search, Terminal, Help) and a close button.

```
jfreeman@mastery1:~/examples/Chapter01/example06$ ansible-playbook -i mastery-hosts -c local mastery.yaml --limit frontend

PLAY [limit example play] ****
TASK [tell us which host we are on] ****
ok: [mastery.example.name] => {
    "inventory_hostname": "mastery.example.name"
}

PLAY RECAP ****
mastery.example.name      : ok=1      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 1.7 – Running the simple playbook on an inventory with a limit applied

Here, we can see that only `mastery.example.name` was operated on this time. While there are no visual clues that the entire inventory was parsed, if we dive into the Ansible code and examine the `inventory` object, we will indeed find all the hosts within. Additionally, we will see how the limit is applied every time the object is queried for items.

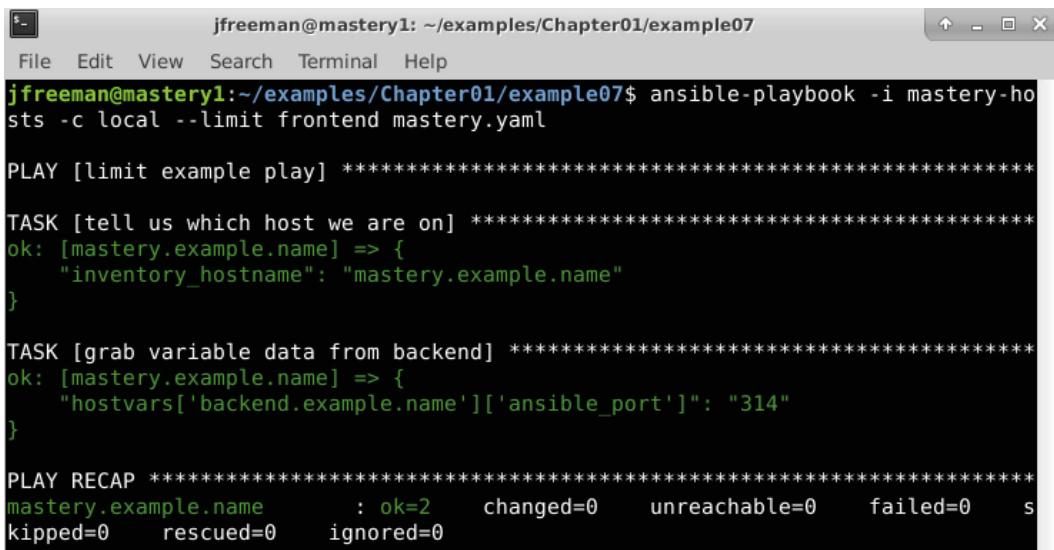
It is important to remember that regardless of the host's pattern used in a play, or the limit that is supplied at runtime, Ansible will still parse the entire inventory that is set during each run. In fact, we can prove this by attempting to access the host variable data for a system that would otherwise be masked by our limit. Let's expand our playbook slightly and attempt to access the `ansible_port` variable from `backend.example.name`:

```
---
- name: limit example play
  hosts: all
  gather_facts: false

  tasks:
    - name: tell us which host we are on
      ansible.builtin.debug:
        var: inventory_hostname

    - name: grab variable data from backend
      ansible.builtin.debug:
        var: hostvars['backend.example.name']['ansible_port']
```

We will still apply our limit by running the playbook with the same command we used in the previous run, which will restrict our operations to just `mastery.example.name`:



```
jfreeman@mastery1: ~/examples/Chapter01/example07$ ansible-playbook -i mastery-hosts -c local --limit frontend mastery.yaml

PLAY [limit example play] ****
TASK [tell us which host we are on] ****
ok: [mastery.example.name] => {
    "inventory_hostname": "mastery.example.name"
}

TASK [grab variable data from backend] ****
ok: [mastery.example.name] => {
    "hostvars['backend.example.name']['ansible_port']: "314"
}

PLAY RECAP ****
mastery.example.name      : ok=2    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
```

Figure 1.8 – Demonstrating that the entire inventory is parsed even with a limit applied

We have successfully accessed the host variable data (by way of group variables) for a system that was otherwise limited out. This is a key skill to understand, as it allows for more advanced scenarios, such as directing a task at a host that is otherwise limited out. Additionally, delegation can be used to manipulate a load balancer; this will put a system into maintenance mode while it is being upgraded without you having to include the load balancer system in your limit mask.

## Playbook parsing

The whole purpose of an inventory source is to have systems to manipulate. The manipulation comes from playbooks (or, in the case of Ansible ad hoc execution, simple single-task plays). You should already have a basic understanding of playbook construction, so we won't spend a lot of time covering that; however, we will delve into some specifics of how a playbook is parsed. Specifically, we will cover the following:

- The order of operations
- Relative path assumptions
- Play behavior keys

- The host selection for plays and tasks
- Play and task names

## The order of operations

Ansible is designed to be as easy as possible for humans to understand. The developers strive to strike the best balance of human comprehension and machine efficiency. To that end, nearly everything in Ansible can be assumed to be executed in a top-to-bottom order; that is, the operation listed at the top of a file will be accomplished before the operation listed at the bottom of a file. Having said that, there are a few caveats and even a few ways to influence the order of operations.

A playbook only has two main operations it can accomplish. It can either run a play, or it can include another playbook from somewhere on the filesystem. The order in which these are accomplished is simply the order in which they appear in the playbook file, from top to bottom. It is important to note that while the operations are executed in order, the entire playbook and any included playbooks are completely parsed before any executions. This means that any included playbook file has to exist at the time of the playbook parsing – they cannot be generated in an earlier operation. This is specific to playbook inclusions but not necessarily to task inclusions that might appear within a play, which will be covered in a later chapter.

Within a play, there are a few more operations. While a playbook is strictly ordered from top to bottom, a play has a more nuanced order of operations. Here is a list of the possible operations and the order in which they will occur:

- Variable loading
- Fact gathering
- The `pre_tasks` execution
- Handlers notified from the `pre_tasks` execution
- The roles execution
- The tasks execution
- Handlers notified from the roles or tasks execution
- The `post_tasks` execution
- Handlers notified from the `post_tasks` execution

The following is an example play with most of these operations shown:

```
---
```

```
- hosts: localhost
  gather_facts: false
```

```

vars:
  - a_var: derp
```

```

pre_tasks:
  - name: pretask
    debug:
      msg: "a pre task"
    changed_when: true
    notify: say hi
```

```

roles:
  - role: simple
    derp: newval
```

```

tasks:
  - name: task
    debug:
      msg: "a task"
    changed_when: true
    notify: say hi
```

```

post_tasks:
  - name: posttask
    debug:
      msg: "a post task"
    changed_when: true
    notify: say hi
```

```

handlers:
  - name: say hi
```

```
debug:  
  msg: hi
```

Regardless of the order in which these blocks are listed in a play, the order detailed in the previous code block is the order in which they will be processed. Handlers (that is, the tasks that can be triggered by other tasks that result in a change) are a special case. There is a utility module, `ansible.builtin.meta`, that can be used to trigger handler processing at a specific point:

```
- ansible.builtin.meta: flush_handlers
```

This will instruct Ansible to process any pending handlers at that point before continuing with the next task or next block of actions within a play. Understanding the order and being able to influence the order with `flush_handlers` is another key skill to have when there is a need to orchestrate complicated actions; for instance, where things such as service restarts are very sensitive to order. Consider the initial rollout of a service.

The play will have tasks that modify `config` files and indicate that the service should be restarted when these files change. The play will also indicate that the service should be running. The first time this play happens, the `config` file will change, and the service will change from not running to running. Then, the handlers will trigger, which will cause the service to restart immediately. This can be disruptive to any consumers of the service. It is better to flush the handlers before a final task to ensure the service is running. This way, the restart will happen before the initial start, so the service will start up once and stay up.

## Relative path assumptions

When Ansible parses a playbook, there are certain assumptions that can be made about the relative paths of items referenced by the statements in a playbook. In most cases, paths for things such as variable files to include, task files to include, playbook files to include, files to copy, templates to render, and scripts to execute are all relative to the directory where the file that is referencing them resides. Let's explore this with an example playbook and directory listing to demonstrate where the files are:

- The directory structure is as follows:

```
.  
├── a_vars_file.yaml  
├── mastery-hosts  
├── relative.yaml  
└── tasks
```

```
|── a.yaml  
└── b.yaml
```

- The content of `a_vars_file.yaml` is as follows:

```
---  
something: "better than nothing"
```

- The content of `relative.yaml` is as follows:

```
---  
- name: relative path play  
  hosts: localhost  
  gather_facts: false  
  
  vars_files:  
    - a_vars_file.yaml  
  
  tasks:  
    - name: who am I  
      ansible.builtin.debug:  
        msg: "I am mastery task"  
  
    - name: var from file  
      ansible.builtin.debug:  
        var: something  
  
    - ansible.builtin.include: tasks/a.yaml
```

- The content of `tasks/a.yaml` is as follows:

```
---  
- name: where am I  
  ansible.builtin.debug:  
    msg: "I am task a"  
  
- ansible.builtin.include: b.yaml
```

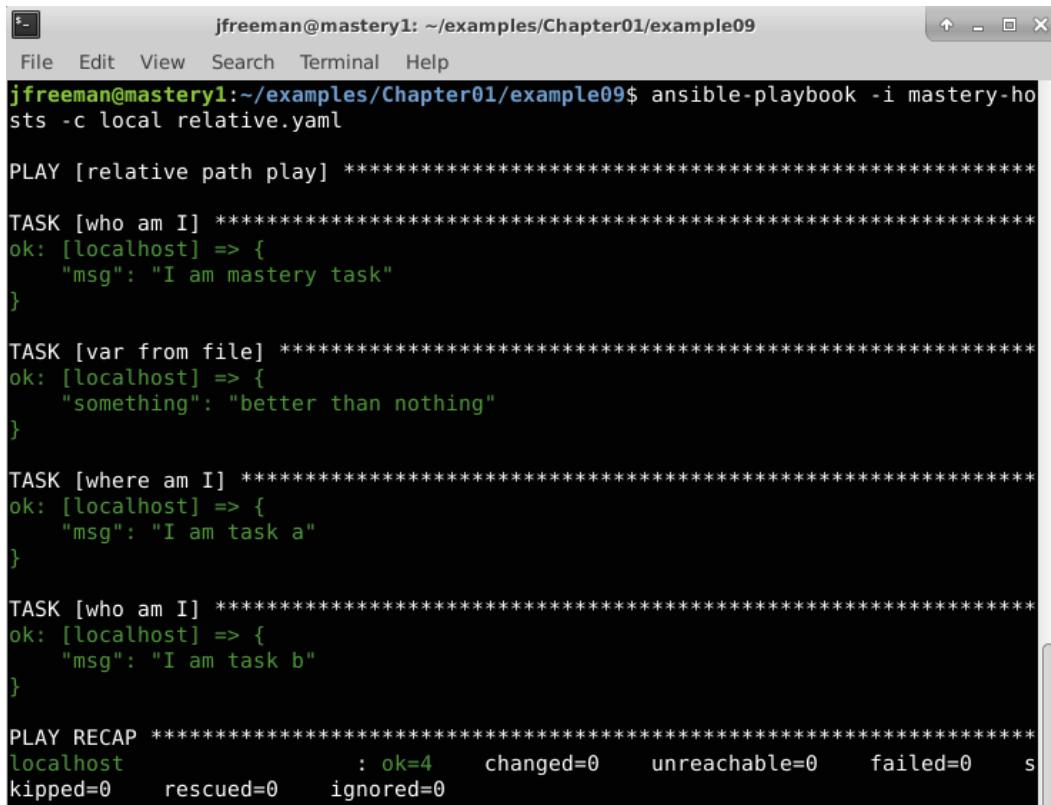
- The content of `tasks/b.yaml` is as follows:

```
---  
- name: who am I  
  ansible.builtin.debug:  
    msg: "I am task b"
```

The execution of the playbook is performed with the following command:

```
ansible-playbook -i mastery-hosts -c local relative.yaml
```

The output should be similar to *Figure 1.9*:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter01/example09". The window shows the command "ansible-playbook -i mastery-hosts -c local relative.yaml" being run. The output displays the execution of the playbook, showing four tasks: "who am I", "var from file", "where am I", and "who am I" again. Each task is successful ("ok: [localhost] => {") and outputs a message. The final output is a PLAY RECAP showing the status of each host.

```
jfreeman@mastery1:~/examples/Chapter01/example09$ ansible-playbook -i mastery-hosts -c local relative.yaml  
  
PLAY [relative path play] *****  
  
TASK [who am I] *****  
ok: [localhost] => {  
    "msg": "I am mastery task"  
}  
  
TASK [var from file] *****  
ok: [localhost] => {  
    "something": "better than nothing"  
}  
  
TASK [where am I] *****  
ok: [localhost] => {  
    "msg": "I am task a"  
}  
  
TASK [who am I] *****  
ok: [localhost] => {  
    "msg": "I am task b"  
}  
  
PLAY RECAP *****  
localhost          : ok=4      changed=0      unreachable=0      failed=0      s  
kipped=0      rescued=0      ignored=0
```

Figure 1.9 – The expected output from running a playbook utilizing relative paths

Here, we can clearly see the relative references to the paths and how they are relative to the file referencing them. When using roles, there are some additional relative path assumptions; however, we'll cover that, in detail, in a later chapter.

## Play behavior directives

When Ansible parses a play, there are a few directives it looks for in order to define various behaviors for a play. These directives are written at the same level as the `hosts`: directive. Here is a list of descriptions for some of the more frequently used keys that can be defined in this section of the playbook:

- `any_errors_fatal`: This Boolean directive is used to instruct Ansible to treat any failure as a fatal error to prevent any further tasks from being attempted. This changes the default, where Ansible will continue until all the tasks have been completed or all the hosts have failed.
- `connection`: This string directive defines which connection system to use for a given play. A common choice to make here is `local`, which instructs Ansible to do all the operations locally but with the context of the system from the inventory.
- `collections`: This is a list of the collection namespaces used within the play to search for modules, plugins, and roles, and it can be used to prevent the need to enter **Fully Qualified Collection Names (FQCNs)** – we will learn more about this in *Chapter 2, Migrating from Earlier Ansible Versions*. Note that this value does not get inherited by role tasks, so you must set it separately in each role in the `meta/main.yml` file.
- `gather_facts`: This Boolean directive controls whether or not Ansible will perform the fact-gathering phase of the operation, where a special task will run on a host to uncover various facts about the system. Skipping fact gathering – when you are sure that you do not require any of the discovered data – can be a significant time-saver in a large environment.
- `Max_fail_percentage`: This number directive is similar to `any_errors_fatal`, but it is more fine-grained. It allows you to define what percentage of your hosts can fail before the whole operation is halted.
- `no_log`: This is a Boolean to control whether or not Ansible will log (to the screen and/or a configured `log` file) the command given or the results received from a task. This is important if your task or return deals with secrets. This key can also be applied to a task directly.
- `port`: This is a number directive to define what SSH port (or any other remote connection plugin) you should use to connect unless this is already configured in the inventory data.
- `remote_user`: This is a string directive that defines which user to log in with on the remote system. The default setting is to connect as the same user that `ansible-playbook` was started with.

- `serial`: This directive takes a number and controls how many systems Ansible will execute a task on before moving to the next task in a play. This is a drastic change from the normal order of operations, where a task is executed across every system in a play before moving to the next. This is very useful in rolling update scenarios, which we will discuss in later chapters.
- `become`: This is a Boolean directive that is used to configure whether privilege escalation (`sudo` or something else) should be used on the remote host to execute tasks. This key can also be defined at a task level. Related directives include `become_user`, `become_method`, and `become_flags`. These can be used to configure how the escalation will occur.
- `strategy`: This directive sets the execution strategy to be used for the play.

Many of these keys will be used in the example playbooks throughout this book.

For a full list of available play directives, please refer to the online documentation at [https://docs.ansible.com/ansible/latest/reference\\_appendices/playbooks\\_keywords.html#play](https://docs.ansible.com/ansible/latest/reference_appendices/playbooks_keywords.html#play).

## Execution strategies

With the release of Ansible 2.0, a new way to control play execution behavior was introduced: `strategy`. A strategy defines how Ansible coordinates each task across the set of hosts. Each strategy is a plugin, and three strategies come with Ansible: linear, debug, and free. The linear strategy, which is the default strategy, is how Ansible has always behaved. As a play is executed, all the hosts for a given play execute the first task.

Once they are all complete, Ansible moves to the next task. The serial directive can create batches of hosts to operate in this way, but the base strategy remains the same. All the targets for a given batch must complete a task before the next task is executed. The debug strategy makes use of the same linear mode of execution described earlier, except that here, tasks are run in an interactive debugging session rather than running to completion without any user intervention. This is especially valuable during the testing and development of complex and/or long-running automation code where you need to analyze the behavior of the Ansible code as it runs, rather than simply running it and hoping for the best!

The free strategy breaks from this traditional linear behavior. When using the free strategy, as soon as a host completes a task, Ansible will execute the next task for that host, without waiting for any other hosts to finish.

This will happen for every host in the set and for every task in the play. Each host will complete the tasks as fast as they can, thus minimizing the execution time of each specific host. While most playbooks will use the default linear strategy, there are situations where the free strategy would be advantageous; for example, when upgrading a service across a large set of hosts. If the play requires numerous tasks to perform the upgrade, which starts with shutting down the service, then it would be more important for each host to suffer as little downtime as possible.

Allowing each host to independently move through the play as fast as it can will ensure that each host is only down for as long as necessary. Without using the free strategy, the entire set will be down for as long as the slowest host in the set takes to complete the tasks.

As the free strategy does not coordinate task completion across hosts, it is not possible to depend on the data that is generated during a task on one host to be available for use in a later task on a different host. There is no guarantee that the first host will have completed the task that generates the data.

Execution strategies are implemented as a plugin and, as such, custom strategies can be developed to extend Ansible behavior by anyone who wishes to contribute to the project.

## The host selection for plays and tasks

The first thing that most plays define (after a name, of course) is a host pattern for the play. This is the pattern used to select hosts out of the inventory object to run the tasks on. Generally, this is straightforward; a host pattern contains one or more blocks indicating a host, group, wildcard pattern, or **regular expression (regex)** to use for the selection. Blocks are separated by a colon, wildcards are just an asterisk, and regex patterns start with a tilde:

```
hostname:groupname:*.example:~(web|db)\.example\.com
```

Advanced usage can include group index selections or even ranges within a group:

```
webservers[0]:webservers[2:4]
```

Each block is treated as an inclusion block; that is, all the hosts found in the first pattern are added to all the hosts found in the next pattern, and so on. However, this can be manipulated with control characters to change their behavior. The use of an ampersand defines an inclusion-based selection (all the hosts that exist in both patterns).

The use of an exclamation point defines an exclusion-based selection (all the hosts that exist in the previous patterns but are NOT in the exclusion pattern):

- `webservers : &dbservers`: Hosts must exist in both the `webservers` and `dbservers` groups.
- `webservers : !dbservers`: Hosts must exist in the `webservers` group but not the `dbservers` group.

Once Ansible parses the patterns, it will then apply restrictions if there are any.

Restrictions come in the form of limits or failed hosts. This result is stored for the duration of the play, and it is accessible via the `play_hosts` variable. As each task is executed, this data is consulted, and an additional restriction could be placed upon it to handle serial operations. As failures are encountered, be it a failure to connect or a failure to execute a task, the failed host is placed in a restriction list so that the host will be bypassed in the next task.

If, at any time, a host selection routine gets restricted down to zero hosts, the play execution will stop with an error. A caveat here is that if the play is configured to have a `max_fail_percentage` or `any_errors_fatal` parameter, then the playbook execution stops immediately after the task where this condition is met.

## Play and task names

While not strictly necessary, it is a good practice to label your plays and tasks with names. These names will show up in the command-line output of `ansible-playbook` and will show up in the log file if the output of `ansible-playbook` is directed to log to a file. Task names also come in handy when you want to direct `ansible-playbook` to start at a specific task and to reference handlers.

There are two main points to consider when naming plays and tasks:

- The names of the plays and tasks should be unique.
- Beware of the kinds of variables that can be used in play and task names.

In general, naming plays and tasks uniquely is a best practice that will help to quickly identify where a problematic task could be residing in your hierarchy of playbooks, roles, task files, handlers, and more. When you first write a small monolithic playbook, they might not seem that important. However, as your use of and confidence in Ansible grows, you will quickly be glad that you named your tasks! Uniqueness is more important when notifying a handler or when starting at a specific task. When task names have duplicates, the behavior of Ansible could be non-deterministic, or at least non-obvious.

With uniqueness as a goal, many playbook authors will look to variables to satisfy this constraint. This strategy might work well, but authors need to be careful regarding the source of the variable data they are referencing. Variable data can come from a variety of locations (which we will cover later in this chapter), and the values assigned to variables can be defined a variety of times. For the sake of play and task names, it is important to remember that only variables for which the values can be determined at the playbook parse time will parse and render correctly. If the data of a referenced variable is discovered via a task or other operation, the variable string will be displayed as unparsed in the output. Let's take a look at an example playbook that utilizes variables for play and task names:

```
---
```

```
- name: play with a {{ var_name }}
```

```
  hosts: localhost
```

```
  gather_facts: false
```

```
  vars:
```

```
    - var_name: not-mastery
```

```
  tasks:
```

```
    - name: set a variable
```

```
      ansible.builtin.set_fact:
```

```
        task_var_name: "defined variable"
```

```
    - name: task with a {{ task_var_name }}
```

```
      ansible.builtin.debug:
```

```
        msg: "I am mastery task"
```

```
- name: second play with a {{ task_var_name }}
```

```
  hosts: localhost
```

```
  gather_facts: false
```

```
  tasks:
```

```
    - name: task with a {{ runtime_var_name }}
```

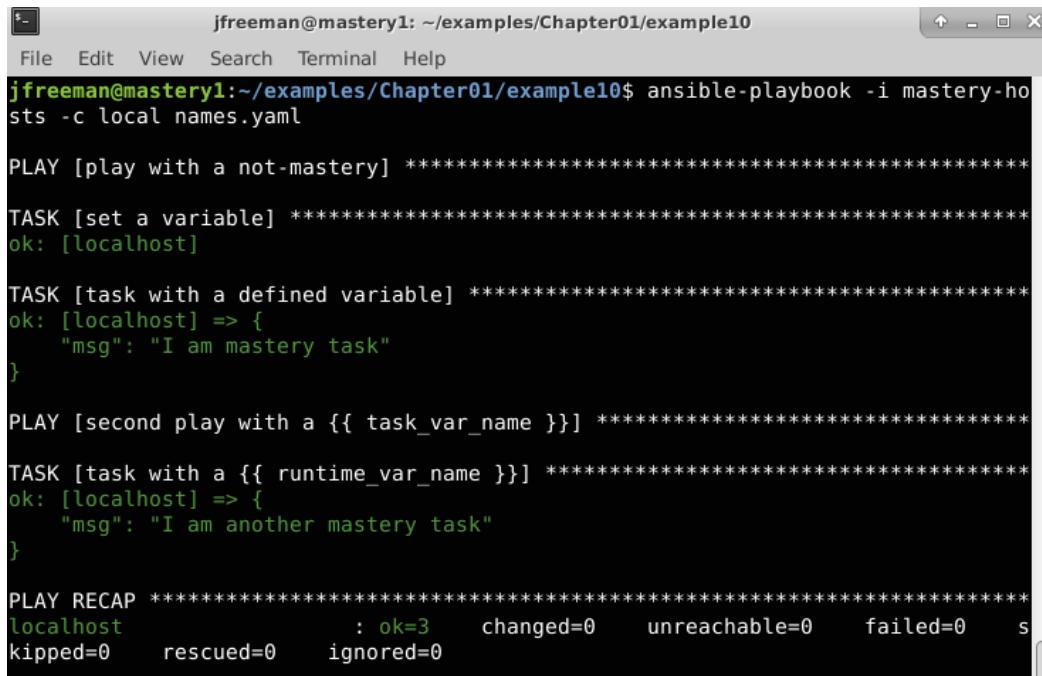
```
      ansible.builtin.debug:
```

```
        msg: "I am another mastery task"
```

At first glance, you might expect at least `var_name` and `task_var_name` to render correctly. We can clearly see `task_var_name` being defined before its use. However, armed with our knowledge that playbooks are parsed in their entirety before execution, we know better. Run the example playbook with the following command:

```
ansible-playbook -i mastery-hosts -c local names.yaml
```

The output should look something like *Figure 1.10*:

A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter01/example10". The terminal shows the execution of an Ansible playbook named "names.yaml". The output is as follows:

```
jfreeman@mastery1:~/examples/Chapter01/example10$ ansible-playbook -i mastery-hosts -c local names.yaml

PLAY [play with a not-mastery] ****
TASK [set a variable] ****
ok: [localhost]

TASK [task with a defined variable] ****
ok: [localhost] => {
    "msg": "I am mastery task"
}

PLAY [second play with a {{ task_var_name }}] ****
TASK [task with a {{ runtime_var_name }}] ****
ok: [localhost] => {
    "msg": "I am another mastery task"
}

PLAY RECAP ****
localhost                  : ok=3      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 1.10 – A playbook run showing the effect of using variables in task names when they are not defined prior to execution

As you can see in *Figure 1.10*, the only variable name that is properly rendered is `var_name`, as it was defined as a static play variable.

# Module transport and execution

Once a playbook is parsed and the hosts are determined, Ansible is ready to execute a task. Tasks are made up of a name (this optional, but nonetheless important, as mentioned previously), a module reference, module arguments, and task control directives. In Ansible 2.9 and earlier, modules were identified by a single unique name. However, in versions of Ansible such as 2.10 and later, the advent of collections (which we will discuss in more detail in the next chapter) meant that Ansible module names could now be non-unique. As a result, those of you with prior Ansible experience might have noticed that, in this book, we are using `ansible.builtin.debug` instead of `debug`, which we would have used in Ansible 2.9 and earlier. In some cases, you can still get away with using the short-form module names (such as `debug`); however, remember that the presence of a collection with its own module called `debug` might cause unexpected results. And, as such, the advice from Ansible in their official documentation is to start making friends with the long-form module names as soon as possible – these are officially called FQCNs. We will use them throughout this book and will explain all of this in more detail in the next chapter. In addition to this, a later chapter will cover task control directives in detail, so we will only concern ourselves with the module reference and arguments.

## The module reference

Every task has a module reference. This tells Ansible which bit of work to carry out. Ansible has been designed to easily allow for custom modules to live alongside a playbook. These custom modules can be a whole new functionality, or they can replace modules shipped with Ansible itself. When Ansible parses a task and discovers the name of the module to use for a task, it looks in a series of locations in order to find the module requested. Where it looks also depends on where the task lives, for example, whether inside a role or not.

If a task is inside a role, Ansible will first look for the module within a directory tree named `library` within the role the task resides in. If the module is not found there, Ansible looks for a directory named `library` at the same level as the main playbook (the one referenced by the `ansible-playbook` execution). If the module is not found there, Ansible will finally look in the configured library path, which defaults to `/usr/share/ansible/`. This library path can be configured in an Ansible `config` file or by way of the `ANSIBLE_LIBRARY` environment variable.

In addition to the preceding paths (which have been established as valid module locations in Ansible almost since its inception), the advent of Ansible 2.10 and newer versions bring with them *Collections*. Collections are now one of the key ways in which modules can be organized and shared with others. For instance, in the earlier example where we looked at the Amazon EC2 dynamic inventory plugin, we installed a collection called `amazon.aws`. In that example, we only made use of the dynamic inventory plugin; however, installing the collection actually installed a whole set of modules for us to use to automate tasks on Amazon EC2. The collection would have been installed in `~/ ansible/collections/ansible_collections/amazon/aws` if you ran the command provided in this book. If you look in there, you will find the modules in the `plugins/modules` subdirectory. Further collections that you install will be located in similar directories, which have been named after the collection that they were installed from.

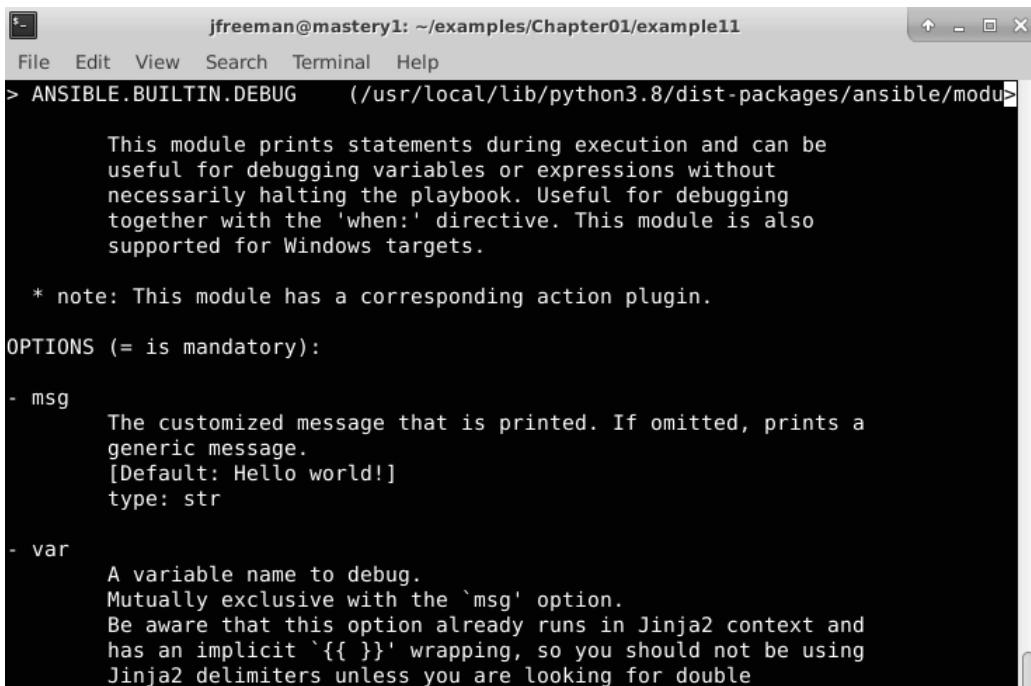
This design, which enables modules to be bundled with collections, roles, and playbooks, allows for the addition of functionality or the reparation of problems quickly and easily.

## Module arguments

Arguments to a module are not always required; the help output of a module will indicate which arguments are required and which are not. Module documentation can be accessed with the `ansible-doc` command, as follows (here, we will use the `debug` module, which we have already used as an example):

```
ansible-doc ansible.builtin.debug
```

*Figure 1.11* shows the kind of output you can expect from this command:



```
jfreeman@mastery1: ~/examples/Chapter01/example11
File Edit View Search Terminal Help
> ANSIBLE.BUILTIN.DEBUG      (/usr/local/lib/python3.8/dist-packages/ansible/modu>
This module prints statements during execution and can be
useful for debugging variables or expressions without
necessarily halting the playbook. Useful for debugging
together with the 'when:' directive. This module is also
supported for Windows targets.

* note: This module has a corresponding action plugin.

OPTIONS (= is mandatory):

- msg
  The customized message that is printed. If omitted, prints a
  generic message.
  [Default: Hello world!]
  type: str

- var
  A variable name to debug.
  Mutually exclusive with the 'msg' option.
  Be aware that this option already runs in Jinja2 context and
  has an implicit `{{ }}` wrapping, so you should not be using
  Jinja2 delimiters unless you are looking for double
```

Figure 1.11 – An example of the output from the `ansible-doc` command run on the `debug` module  
If you scroll through the output, you will find a wealth of useful information including example code, the outputs from the module, and the arguments (that is, options), as shown in *Figure 1.11*

Arguments can be templated with **Jinja2**, which will be parsed at module execution time, allowing for data discovered in a previous task to be used in later tasks; this is a very powerful design element.

Arguments can either be supplied in a `key=value` format or in a complex format that is more native to YAML. Here are two examples of arguments being passed to a module showcasing these two formats:

```
- name: add a keypair to nova
  openstack.cloudkeypair: cloud={{ cloud_name }} name=admin-key
  wait=yes

- name: add a keypair to nova
  openstack.cloud.keypair:
```

```
cloud: "{{ cloud_name }}"  
name: admin-key  
wait: yes
```

In this example, both formats will lead to the same result; however, the complex format is required if you wish to pass complex arguments into a module. Some modules expect a list object or a hash of data to be passed in; the complex format allows for this. While both formats are acceptable for many tasks, the complex format is the format used for the majority of examples in this book as, despite its name, it is actually easier for humans to read.

## Module blacklisting

Starting with Ansible 2.5, it is now possible for system administrators to blacklist Ansible modules that they do not wish to be available to playbook developers. This might be for security reasons, to maintain conformity or even to avoid the use of deprecated modules.

The location for the module blacklist is defined by the `plugin_filters_cfg` parameter found in the `defaults` section of the Ansible configuration file. By default, it is disabled, and the suggested default value is set to `/etc/ansible/plugin_filters.yml`.

The format for this file is, at present, very simple. It contains a version header to allow for the file format to be updated in the future and a list of modules to be filtered out. For example, if you were preparing for a transition to Ansible 4.0 and were currently on Ansible 2.7, you would note that the `sf_account_manager` module is to be completely removed in Ansible 4.0. As a result, you might wish to prevent users from making use of this by blacklisting it to prevent anyone from creating code that would break when Ansible 4.0 is rolled out (please refer to [https://docs.ansible.com/ansible-devel/porting\\_guides/porting\\_guide\\_2.7.html](https://docs.ansible.com/ansible-devel/porting_guides/porting_guide_2.7.html)). Therefore, to prevent anyone from using this internally, the `plugin_filters.yml` file should look like this:

```
---  
filter_version: '1.0'  
module_blacklist:  
  # Deprecated - to be removed in 4.0  
  - sf_account_manager
```

Although useful in helping to ensure high-quality Ansible code is maintained, this functionality is, at the time of writing, limited to modules. It cannot be extended to anything else, such as roles.

## Module transport and execution

Once a module is found, Ansible has to execute it in some way. How the module is transported and executed depends on a few factors; however, the common process is to locate the module file on the local filesystem and read it into memory, and then add the arguments passed to the module. Then, the boilerplate module code from the Ansible core is added to the file object in memory. This collection is compressed, Base64-encoded, and then wrapped in a script. What happens next really depends on the connection method and runtime options (such as leaving the module code on the remote system for review).

The default connection method is `smart`, which most often resolves to the `ssh` connection method. With a default configuration, Ansible will open an SSH connection to the remote host, create a temporary directory, and close the connection. Ansible will then open another SSH connection in order to write out the wrapped ZIP file from memory (the result of local module files, task module arguments, and Ansible boilerplate code) into a file within the temporary directory that we just created and close the connection.

Finally, Ansible will open a third connection in order to execute the script and delete the temporary directory and all of its contents. The module results are captured from `stdout` in JSON format, which Ansible will parse and handle appropriately. If a task has an `async` control, Ansible will close the third connection before the module is complete and SSH back into the host to check the status of the task after a prescribed period until the module is complete or a prescribed timeout has been reached.

## Task performance

The previous discussion regarding how Ansible connects to hosts results in three connections to the host for every task. In a small environment with a small number of tasks, this might not be a concern; however, as the task set grows and the environment size grows, the time required to create and tear down SSH connections increases. Thankfully, there are a couple of ways to mitigate this.

The first is an SSH feature, `ControlPersist`, which provides a mechanism that creates persistent sockets when first connecting to a remote host that can be reused in subsequent connections to bypass some of the handshaking required when creating a connection. This can drastically reduce the amount of time Ansible spends on opening new connections. Ansible automatically utilizes this feature if the host platform that runs Ansible supports it. To check whether your platform supports this feature, refer to the SSH man page for `ControlPersist`.

The second performance enhancement that can be utilized is an Ansible feature called pipelining. Pipelining is available to SSH-based connection methods and is configured in the Ansible configuration file within the `ssh_connection` section:

```
[ssh_connection]
  pipelining=true
```

This setting changes how modules are transported. Instead of opening an SSH connection to create a directory, another to write out the composed module, and a third to execute and clean up, Ansible will instead open an SSH connection on the remote host. Then, over that live connection, Ansible will pipe in the zipped composed module code and script for execution. This reduces the connections from three to one, which can really add up. By default, pipelining is disabled to maintain compatibility with the many Linux distributions that have `requiretty` enabled in their `sudoers` configuration file.

Utilizing the combination of these two performance tweaks can keep your playbooks nice and fast even as you scale your environment. However, bear in mind that Ansible will only address as many hosts at once as the number of forks Ansible is configured to run. Forks are the number of processes Ansible will split off as a worker to communicate with remote hosts. The default is five forks, which will address up to five hosts at once. You can raise this number to address more hosts as your environment size grows by adjusting the `forks=` parameter in an Ansible configuration file or by using the `--forks` (`-f`) argument with `ansible` or `ansible-playbook`.

## Variable types and location

Variables are a key component of the Ansible design. Variables allow for dynamic play content and reusable plays across different sets of an inventory. Anything beyond the most basic of Ansible use will utilize variables. Understanding the different variable types and where they can be located, as well as learning how to access external data or prompt users to populate variable data, is one of the keys to mastering Ansible.

### Variable types

Before diving into the precedence of variables, first, we must understand the various types and subtypes of variables available to Ansible, their locations, and where they are valid for use.

The first major variable type is **inventory variables**. These are the variables that Ansible gets by way of the inventory. These can be defined as variables that are specific to `host_vars`, to individual hosts, or applicable to entire groups as `group_vars`. These variables can be written directly into the inventory file, delivered by the dynamic inventory plugin, or loaded from the `host_vars/<host>` or `group_vars/<group>` directories.

These types of variables can be used to define Ansible behavior when dealing with these hosts or site-specific data related to the applications that these hosts run. Whether a variable comes from `host_vars` or `group_vars`, it will be assigned to a host's `hostvars`, and it can be accessed from the playbooks and template files. Accessing a host's own variables can be done by simply referencing the name, such as `{ { foobar } }`, and accessing another host's variables can be accomplished by accessing `hostvars`; for example, to access the `foobar` variable for `examplehost`, you can use `{ { hostvars['examplehost']['foobar'] } }`. These variables have global scope.

The second major variable type is **role variables**. These are variables that are specific to a role and are utilized by the role tasks. However, it should be noted that once a role has been added to a playbook, its variables are generally accessible throughout the rest of the playbook, including from within other roles. In most simple playbooks, this won't matter, as the roles are typically run one at a time. But it is worth remembering this as the playbook structure becomes more complex; otherwise, unexpected behavior could result from variables being set within a different role!

These variables are often supplied as a **role default**, that is, they are meant to provide a default value for the variable but can easily be overridden when applying the role. When roles are referenced, it is possible to supply variable data at the same time, either by overriding role defaults or creating wholly new data. We'll cover roles in more depth in a later chapter. These variables apply to all hosts on which the role is executed and can be accessed directly, much like a host's own `hostvars`.

The third major variable type is **play variables**. These variables are defined in the control keys of a play, either directly by the `vars` key or sourced from external files via the `vars_files` key. Additionally, the play can interactively prompt the user for variable data using `vars_prompt`. These variables are to be used within the scope of the play and in any tasks or included tasks of the play. The variables apply to all hosts within the play and can be referenced as if they are `hostvars`.

The fourth variable type is **task variables**. Task variables are made from data that has been discovered while executing tasks or in the fact-gathering phase of a play. These variables are host-specific and are added to the host's `hostvars`, and they can be used as such, which also means they have a global scope after the point in which they were discovered or defined. Variables of this type can be discovered via `gather_facts` and `fact modules` (that is, modules that do not alter state but instead return data), populated from task return data via the `register` task key, or defined directly by a task making use of the `set_fact` or `add_host` modules. Data can also be interactively obtained from the operator using the `prompt` argument to the `pause` module and registering the result:

```
- name: get the operators name
  ansible.builtin.pause:
    prompt: "Please enter your name"
  register: opname
```

The **extra variables**, or the `extra-vars` type, are variables supplied on the command line when executing `ansible-playbook` via `--extra-vars`. Variable data can be supplied as a list of `key=value` pairs, a quoted piece of JSON data, or a reference to a YAML-formatted file with variable data defined within:

```
--extra-vars "foo=bar owner=fred"
--extra-vars '{"services": ["nova-api", "nova-conductor"]}'
--extra-vars @/path/to/data.yaml
```

Extra variables are considered global variables. They apply to every host and have scope throughout the entire playbook.

## Magic variables

In addition to the previously listed variable types, Ansible offers a set of variables that deserve their own special mention – **magic variables**. These are variables that are always set when a playbook is run without them having to be explicitly created. Their names are always reserved and should not be used for other variables.

Magic variables are used to provide information about the current playbook run to the playbooks themselves and are extremely useful as Ansible environments become larger and more complex. For example, if one of your plays needs information about which groups the current host is in, the `group_names` magic variable returns a list of them. Similarly, if you need to configure the hostname for a service using Ansible, the `inventory_hostname` magic variable will return the current hostname as it is defined in the inventory. A simple example of this is as follows:

```
---  
- name: demonstrate magic variables  
  hosts: all  
  gather_facts: false  
  
  tasks:  
    - name: tell us which host we are on  
      ansible.builtin.debug:  
        var: inventory_hostname  
  
    - name: tell us which groups we are in  
      ansible.builtin.debug:  
        var: group_names
```

As with everything in the Ansible project, magic variables are well documented, and you can find a full list of them and what they contain in the official Ansible documentation at [https://docs.ansible.com/ansible/latest/reference\\_appendices/special\\_variables.html](https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html). A practical example of the use of magic variables is this: imagine, for example, setting up the hostnames on a new set of Linux servers from a blank template. The `inventory_hostname` magic variable provides us with the hostname we need directly from the inventory, without the need for another source of data (or, for example, a connection to the **CMDB**). Similarly, accessing `groups_names` allows us to define which plays should be run on a given host within a single playbook – perhaps, for example, installing **NGINX** if the host is in the `webservers` group. In this way, Ansible code can be made more versatile and efficient; hence, these variables deserve a special mention.

## Accessing external data

Data for role variables, play variables, and task variables can also come from external sources. Ansible provides a mechanism in which to access and evaluate data from the **control machine** (that is, the machine running `ansible-playbook`). The mechanism is called a **lookup plugin**, and a number of them come with Ansible. These plugins can be used to look up or access data by reading files, generate and locally store passwords on the Ansible host for later reuse, evaluate environment variables, pipe data in from executables or CSV files, access data in the Redis or etcd systems, render data from template files, query dnstxt records, and more. The syntax is as follows:

```
lookup('<plugin_name>', 'plugin_argument')
```

For example, to use the `mastery` value from etcd in an `ansible.builtin.debug` task, execute the following command:

```
- name: show data from etcd
  ansible.builtin.debug:
    msg: "{{ lookup('etcd', 'mastery') }}"
```

Lookups are evaluated when the task referencing them is executed, which allows for dynamic data discovery. To reuse a particular lookup in multiple tasks and reevaluate it each time, a playbook variable can be defined with a lookup value. Each time the playbook variable is referenced, the lookup will be executed, potentially providing different values over time.

## Variable precedence

As you learned in the previous section, there are several major types of variables that can be defined in a myriad of locations. This leads to a very important question: what happens when the same variable name is used in multiple locations? Ansible has a precedence for loading variable data, and thus, it has an order and a definition to decide which variable will win. Variable value overriding is an advanced usage of Ansible, so it is important to fully understand the semantics before attempting such a scenario.

## Precedence order

Ansible defines the following precedence order, with those closest to the top of the list winning. Note that this can change from release to release. In fact, it has changed quite significantly since Ansible 2.4 was released, so it is worth reviewing if you are upgrading from an older version of Ansible:

1. Extra `vars` (from the command line) always win.
2. The `ansible.builtin.include` parameters.
3. The role (and `ansible.builtin.include_role`) parameters.
4. The variables defined with `ansible.builtin.set_facts`, and those created with the `register` task directive.
5. The variables included in a play with `ansible.builtin.include_vars`.
6. Task `vars` (only for the specific task).
7. Block `vars` (only for the tasks within the block).
8. Role `vars` (defined in `main.yml` in the `vars` subdirectory of the role).
9. Play `vars_files`.
10. Play `vars_prompt`.
11. Play `vars`.
12. The host facts (and also the cached results of `ansible.builtin.set_facts`).
13. The `host_vars` playbook.
14. The `host_vars` inventory.
15. The inventory file (or script)-defined host `vars`.
16. The `group_vars` playbook.
17. The `group_vars` inventory.
18. The `group_vars/all` playbook.
19. The `group_vars/all` inventory.
20. The inventory file (or script)-defined group `vars`.
21. The role defaults.
22. The command-line values (for example, `-u REMOTE_USER`).

Ansible releases a porting guide with each release that details the changes you will need to make to your code in order for it to continue functioning as expected. It is important to review these as you upgrade your Ansible environment – the guides can be found at [https://docs.ansible.com/ansible-devel/porting\\_guides/porting\\_guides.html](https://docs.ansible.com/ansible-devel/porting_guides/porting_guides.html).

## Variable group priority ordering

The previous list of priority ordering is obviously helpful when writing Ansible playbooks, and, in most cases, it is apparent that variables should not clash. For example, a `var` task clearly wins over a `var` play, and all tasks and, indeed, plays are unique. Similarly, all hosts in the inventory will be unique; so again, there should be no clash of variables with the inventory either.

There is, however, one exception to this, that is, inventory groups. A one-to-many relationship exists between hosts and groups, and, as such, any given host can be a member of one or more groups. For example, let's suppose that the following code is our inventory file:

```
[frontend]
host1.example.com
host2.example.com

[web:children]
frontend

[web:vars]
http_port=80
secure=true

[proxy]
host1.example.com

[proxy:vars]
http_port=8080
thread_count=10
```

Here, we have two hypothetical frontend servers, `host1.example.com` and `host2.example.com`, in the `frontend` group. Both hosts are children of the `web` group, which means they are assigned the group variable `http_port=80` from the inventory. `host1.example.com` is also a member of the `proxy` group, which has an identically named variable but a different assignment: `http_port=8080`.

Both of these variable assignments are at the `group_vars` inventory level, and so the order of precedence does not define a winner. So, what happens in this scenario?

The answer is, in fact, predictable and deterministic. The `group_vars` assignments are done in alphabetical order of the group names (as described in the *Inventory ordering* section), with the last loaded group overriding all preceding variable values that coincide.

This means that any competing variables from `mastery2` will win over the other two groups. Those from the `mastery11` group will then take precedence of those from the `mastery1` group, so please be mindful of this when creating group names!

In our example, when the groups are processed in alphabetical order, `web` comes after `proxy`. Therefore, the `group_vars` assignments from `web` that coincide with those from any previously processed groups will win. Let's run the previous inventory file through this example playbook to take a look at the behavior:

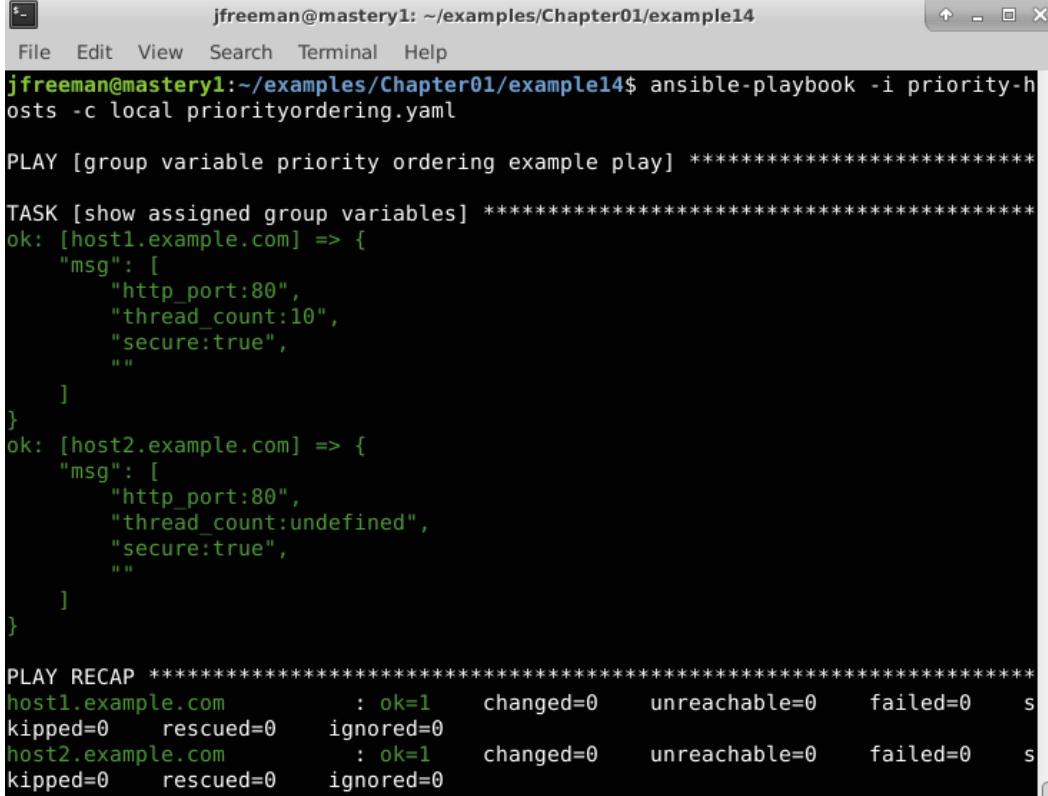
```
---
- name: group variable priority ordering example play
  hosts: all
  gather_facts: false

  tasks:
    - name: show assigned group variables
      vars:
        msg: |
          http_port:{{ hostvars[inventory_hostname] ['http_port'] }}
          thread_count:{{ hostvars[inventory_hostname] ['thread_count'] | default("undefined") }}
          secure:{{ hostvars[inventory_hostname] ['secure'] }}
      ansible.builtin.debug:
        msg: "{{ msg.split('\n') }}"
```

Let's try running the following command:

```
ansible-playbook -i priority-hosts -c local priorityordering.yaml
```

We should get the following output:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter01/example14". The terminal displays the output of an Ansible playbook run. The output shows two hosts being processed: host1.example.com and host2.example.com. For each host, a task is run to show assigned group variables. The variables shown are http\_port, thread\_count, and secure. For host1, http\_port is explicitly set to 80. For host2, http\_port is undefined. The secure variable is set to true for both hosts. The final PLAY RECAP section shows that both hosts completed successfully (ok=1) with 0 changes, 0 unreachable hosts, and 0 failed hosts.

```
jfreeman@mastery1: ~/examples/Chapter01/example14
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter01/example14$ ansible-playbook -i priority-hosts -c local priorityordering.yaml

PLAY [group variable priority ordering example play] ****
TASK [show assigned group variables] ****
ok: [host1.example.com] => {
    "msg": [
        "http_port:80",
        "thread_count:10",
        "secure:true",
        ""
    ]
}
ok: [host2.example.com] => {
    "msg": [
        "http_port:80",
        "thread_count:undefined",
        "secure:true",
        ""
    ]
}

PLAY RECAP ****
host1.example.com      : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
host2.example.com      : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
```

Figure 1.12 – A playbook run showing how variables can be overridden at the inventory group level

As expected, the value assigned to the `http_port` variable for both hosts in the inventory is 80. However, what if this behavior is not desired? Let's suppose that we want the value of `http_port` from the proxy group to take priority. It would be painful to have to rename the group and all associated references to it to change the alphanumerical sorting of the groups (although, this would work!). The good news is that Ansible 2.4 introduced the `ansible_group_priority` group variable, which can be used for just this eventuality. If not explicitly set, this variable defaults to 1, leaving the rest of the inventory file unchanged.

Let's set this as follows:

```
[proxy:vars]
http_port=8080
thread_count=10
ansible_group_priority=10
```

Now, when we run the same playbook using the same command as before, pay attention to how the value assigned to `http_port` changes, while all variable names that were not coincidental behave exactly as before:

```
jfreeman@mastery1: ~/examples/Chapter01/example15
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter01/example15$ ansible-playbook -i priority-hosts -c local priorityordering.yaml

PLAY [group variable priority ordering example play] ****
TASK [show assigned group variables] ****
ok: [host1.example.com] => {
    "msg": [
        "http_port:8080",
        "thread_count:10",
        "secure:true",
        ""
    ]
}
ok: [host2.example.com] => {
    "msg": [
        "http_port:80",
        "thread_count:undefined",
        "secure:true",
        ""
    ]
}

PLAY RECAP ****
host1.example.com      : ok=1    changed=0    unreachable=0    failed=0    s
kippe
host2.example.com      : ok=1    changed=0    unreachable=0    failed=0    s
kippe
```

Figure 1.13 – The effect of the `ansible_group_priority` variable on coincidental group variables

As your inventory grows with your infrastructure, be sure to make use of this feature to gracefully handle any variable assignment collisions between your groups.

## Merging hashes

In the previous section, we focused on the precedence with which variables will override each other. The default behavior of Ansible is that any overriding definition for a variable name will completely mask the previous definition of that variable. However, that behavior can be altered for one type of variable: the hash variable. A hash variable (or, in Python terms, a dictionary) is a dataset of keys and values. Values can be of different types for each key and can even be hashes themselves for complex data structures.

In some advanced scenarios, it is preferable to replace just one bit of a hash or add to an existing hash rather than replacing the hash altogether. To unlock this ability, a configuration change is necessary in the Ansible config file. The configuration entry is `hash_behavior`, which either takes the value `replace` or `merge`. A setting of `merge` will instruct Ansible to merge or blend the values of two hashes when presented with an override scenario rather than assume the default of `replace`, which will completely replace the old variable data with the new data.

Let's walk through an example of the two behaviors. We will start with a hash loaded with data and simulate a scenario where a different value for the hash is provided as a higher-priority variable.

This is the starting data:

```
hash_var:  
  fred:  
    home: Seattle  
    transport: Bicycle
```

This is the new data loaded via `include_vars`:

```
hash_var:  
  fred:  
    transport: Bus
```

With the default behavior, the new value for `hash_var` will be as follows:

```
hash_var:  
  fred:  
    transport: Bus
```

However, if we enable the `merge` behavior, we will get the following result:

```
hash_var:  
  fred:  
    home: Seattle  
    transport: Bus
```

There are even more nuances and undefined behaviors when using `merge`, and, as such, it is strongly recommended that you only use this setting if absolutely necessary – it is disabled by default for a good reason!

## Summary

While the design of Ansible focuses on simplicity and ease of use, the architecture itself is very powerful. In this chapter, we covered the key design and architecture concepts of Ansible, such as versions and configurations, playbook parsing, module transport and execution, variable types and locations, and variable precedence.

You learned that playbooks contain variables and tasks. Tasks link bits of code called modules with arguments, which can be populated by variable data. These combinations are transported to selected hosts from the inventory sources provided. The fundamental understanding of these building blocks is the platform on which you can build a mastery of all things Ansible!

In the next chapter, you will learn, in detail, about the big new features in Ansible 4.3, especially the Ansible collections and FQCNs that we have touched on in this chapter.

## Questions

1. Why is an inventory important to Ansible?
  - a) It forms part of Ansible's configuration management database.
  - b) It is used to audit your servers.
  - c) It tells Ansible which servers to perform automation tasks on.
  - d) None of the above.

2. When working with frequently changing infrastructures (such as public cloud deployments), Ansible users must manually update their inventory on a regular basis. Is this true or false?
  - a) True – this is the only way to do it.
  - b) False – dynamic inventories were invented for precisely this purpose.
3. By default, Ansible processes hosts in an inventory in which order?
  - a) In alphabetical order
  - b) In lexicographical order
  - c) In random order
  - d) In the order in which they appear in the inventory
4. By default, Ansible tasks in a simple playbook are executed in which order?
  - a) In the order in which they are written, but each task must be completed on all inventory hosts before the next is executed.
  - b) In the most optimal order.
  - c) In the order in which they are written but only on one inventory host at a time.
  - d) Something else.
5. Which variable type takes the highest priority, overriding all other variable sources?
  - a) Inventory variables
  - b) Extra variables (from the command line)
  - c) Role defaults
  - d) Variables source via `vars_prompt`
6. What is the name of the special Ansible variables that only exist at runtime?
  - a) Special variables
  - b) Runtime variables
  - c) Magic variables
  - d) User variables

7. If you wanted to access external data from a playbook, what would you use?
  - a) A lookup plugin
  - b) A lookup module
  - c) A lookup executable
  - d) A lookup role
8. What is Ansible's preferred default transport mechanism for most non-Windows hosts?
  - a) The REST API
  - b) RabbitMQ
  - c) RSH
  - d) SSH
9. What can inventory variables be used to do?
  - a) Define unique data for each host or group of hosts in an inventory.
  - b) Declare your playbook variables.
  - c) Define connection parameters for your inventory hosts.
  - d) Both (a) and (c).
10. How can you override the default Ansible configuration on your system?
  - a) By creating an Ansible configuration file in any location, and using the `ANSIBLE_CFG` environment variable to specify this location.
  - b) By creating a file called `ansible.cfg` in the current working directory.
  - c) By creating a file in your home directory called `~/ ansible.cfg`.
  - d) Any of the above.



# 2

# Migrating from Earlier Ansible Versions

As **Ansible** has grown over the years, certain headaches have presented themselves to the team that develops and manages the Ansible code base. In many ways, these headaches have been the price of Ansible's own growth and success, and have resulted in a need to structure the code a little differently. Indeed, anyone with a little prior experience of Ansible from versions before 2.10 will have noticed that our example code presented in this book looks a little different, along with a new term, **Collections**.

In this chapter, we will explain these changes in detail, along with how they came about. We will then take you through some practical examples so you can see how these changes work in the real world, before finally teaching you how to migrate any existing or legacy playbooks you might have to Ansible 4.3 and beyond.

Specifically, in this chapter, we will cover the following topics:

- Changes in Ansible 4.3
- Upgrading from earlier Ansible installations
- Installing Ansible from scratch
- What are Ansible Collections?
- Installing additional modules with `ansible-galaxy`
- How to port legacy playbooks to Ansible 4.3 (a primer)

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running **Ansible 4.3** or newer. Almost any flavor of Linux should do. For those interested in specifics, all the code presented in this chapter was tested on **Ubuntu Server 20.04 LTS** unless stated otherwise, and on **Ansible 4.3**. The example code that accompanies this chapter can be downloaded from GitHub at this URL: <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter02>. We will make use of a module that we develop in *Chapter 10, Extending Ansible*, to show you how to build your own collection, so it is worthwhile making sure you have a copy of the accompanying code for this book.

Check out the following video to view the Code in Action: <https://bit.ly/3DYi0Co>

## Changes in Ansible 4.3

While we touched on this topic in *Chapter 1, The System Architecture and Design of Ansible*, it is important that we look in greater depth at these changes to help you fully understand how Ansible 4.3 differs from prior releases. This will help you greatly in writing good playbooks and maintaining and upgrading your Ansible infrastructure – it is an essential step to mastery of Ansible 4.3!

First off, a little history. As we discussed in the preceding chapter, Ansible possesses a number of strengths in its design that have led to its rapid growth and uptake. Many of these strengths, such as its agentless design and easy-to-read YAML code, remain the same. Indeed, if you read the change logs for Ansible releases since 2.9, you will observe that there have been few changes of note to the core Ansible functionality since that release—rather, all the development effort has gone into another area.

Ansible's modules were undoubtedly one of its greatest strengths, and the fact that anyone, from individual contributors to hardware vendors and cloud providers, could submit their own modules meant that by the time of the 2.9 release, Ansible contained literally thousands of modules for every conceivable purpose.

This in itself became something of a headache for those managing the project. Let's say that a module had a bug in it that needed to be fixed, or someone added a great new feature to an existing one that was likely to be popular. The Ansible release itself contained all the modules—in short, they were tightly coupled to the release of Ansible itself. This meant that in order for a new module to get released, a whole new version of Ansible had to be released to the community.

Combine that with issues and pull requests from hundreds of module developers, and those managing the core Ansible code base had a real headache on their hands. It was clear that while these modules were a massive part of Ansible's success, they were also responsible for causing issues in release cycles and management of the code base. What was needed was a way to decouple the modules (or at least the bulk of them) from the releases of the Ansible engine—the core Ansible runtimes that we ran in *Chapter 1, The System Architecture and Design of Ansible*, of this book.

Thus, **Ansible Content Collections** (or just Collections for short) were born.

## Ansible Content Collections

While we will look in greater depth at these shortly, the important concept of note is that Collections are a package format for Ansible content, which for the sake of this discussion means all those thousands of modules. By distributing modules, especially those coded and maintained by third parties, using Collections, the Ansible team has effectively removed the coupling between the releases of the core Ansible product and the modules that make it so valuable to so many.

This of course leads to another nuance—when you installed, say, **Ansible 2.9.1**, you were installing a given version of the Ansible binaries and other core code, and all the modules that were submitted and approved for inclusion at that time.

Now, when we talk about installing Ansible 4.3, what we actually mean is this:

Ansible 4.3.0 is now a package that contains (at the time of writing) 85 Collections of modules, plugins, and other important functionality that will get as many people as possible started on their Ansible journey before they need to install further Collections. It is, in short, a *getting started* pack of Collections.

What is important here is that Ansible 4.3.0 does *not* contain any actual automation runtimes. If you were to install Ansible 4.3.0 in isolation, you would not actually be able to run Ansible! Fortunately, doing this is not possible, and Ansible 4.3.0 has a dependency on a package currently called **ansible-core**. This package contains the Ansible language runtimes, and a small number of the core plugins and modules, such as `ansible.builtin.debug`, which we used frequently in the examples in *Chapter 1, The System Architecture and Design of Ansible*.

Each release of the Ansible package will have dependencies on specific versions of `ansible-core`, such that it will always pair itself up with the right automation engine. For example, Ansible 4.3.0 depends upon `ansible-core >= 2.11` and `< 2.12`.

Ansible has switched to semantic versioning for the Ansible package itself, starting with the 3.0.0 release. For anyone who hasn't come across semantic versioning yet, it can be explained quite simply as follows:

- Ansible 4.3.0: This is the first semantic versioned release of a new Ansible package.
- Ansible 4.0.1: This (and all releases where the rightmost digit changes) will contain only backward-compatible bug fixes.
- Ansible 4.1.0: This (and all releases where the middle digit changes) will contain backward-compatible new features, and possibly also bug fixes.
- Ansible 5.0.0: This will contain changes that break backward compatibility and is known as a major release.

The `ansible-core` package is not adopting semantic versioning, so it is anticipated that Ansible 5.0.0 will have a dependency on `ansible-core >= 2.12`. Note that this release of `ansible-core`, not being under semantic versioning, could contain changes that break backward compatibility, and so it is important on our journey to mastery to be aware of these nuances in how Ansible is now versioned.

#### Important note

Finally, note that the `ansible-core` package was renamed from `ansible-base` in the 2.11 release, so if you see references to `ansible-base` please know it is simply the old name for the `ansible-core` package.

All these changes have been planned and executed over a significant period of time. While their implementation has been designed to make the journey for existing Ansible users as smooth as possible, there are implications that need to be addressed, starting with how you actually install and upgrade Ansible, and we will look at exactly that in the next section.

## Upgrading from earlier Ansible installations

The splitting of Ansible into two packages, one of which is dependent on the other, has created some headaches for package maintainers. Whereas packages were readily available for CentOS and RHEL, there are no current packages of Ansible 4.3.0 or ansible-core 2.11.1. A quick look inside the EPEL packages directory for CentOS/RHEL 8 shows that the latest RPM available for Ansible is version 2.9.18. The official Ansible installation guide goes further:

*Since Ansible 2.10 for RHEL is not available at this time, continue to use Ansible 2.9.*

This will change in due course as package maintainers work out the pros and cons of the various upgrade paths and packaging technologies, but at the time of writing, there is a very clear expectation on your upgrade path if you want to get started with Ansible 4.3.0 now, and the easiest way to get your hands on this latest and greatest release is to install it using the Python packaging technology, **pip**. However, it is not so much an upgrade we're performing, as an uninstallation followed by a re-installation.

## Uninstalling Ansible 3.0 or older

The radical changes in Ansible package structure mean that if you have Ansible 3.0 or earlier (including any of the 2.x releases) installed on your control node, sadly you cannot just upgrade your Ansible installation. Rather, you need to remove your existing Ansible installation before you install the later version.

### Tip

As with removing any software, you should make sure you take a backup of your important files, especially central Ansible configuration files and inventories, in case they get removed during the removal process.

The method for removing your package will depend upon your installation. For example, if you have Ansible 2.9.18 installed on CentOS 8 through an RPM, you can remove it with the following command:

```
sudo dnf remove ansible
```

Similarly, on Ubuntu you can run the following command:

```
sudo apt remove ansible
```

If you installed Ansible using `pip` previously, you can remove it with the following command:

```
pip uninstall ansible
```

In short, it doesn't matter how you installed Ansible 3.0 (or earlier) on your control node. Even if you installed it with `pip`, and you are going to install the new version with `pip`, you must first uninstall the old version before doing anything else.

When newer Ansible versions become available, it is advisable to check the documentation to see whether an uninstall is still required as part of the upgrade. For example, it was necessary to uninstall Ansible 3.0 before installing Ansible 4.3, partly due to the renaming of the `ansible-base` package to `ansible-core`.

Once you have removed your earlier version of Ansible, you are now ready to proceed with installing the new version on your control node, which we will cover in the next section.

## Installing Ansible from scratch

As discussed in the preceding section, Ansible 4.3 is largely packaged and distributed using a Python package manager called `pip`. This is likely to change in due course, but at the time of writing, the key installation method you will need to use is to install via `pip`. Now, it's fair to say that most modern Linux distributions already come with Python and `pip` pre-installed. If for any reason you get stuck and need to install it, the process is well documented on the official website here: <https://pip.pypa.io/en/stable/installing/>.

Once you have `pip` installed, the process of installing Ansible is as simple as running this command, and the beauty is, the command is the same on all operating systems (though note that on some operating systems, your `pip` command might be called `pip3` to differentiate between the Python 2.7 and Python 3 releases that may coexist):

```
sudo pip install ansible
```

There are, of course, a few variations on this command. For example, the command as we have given it will install the latest version of Ansible available for all users on the system.

If you want to test or stick to a specific version (perhaps for testing or qualification purposes), you could force `pip` to install a specific version with the following command:

```
sudo pip install ansible==4.3.0
```

This second command will ensure that Ansible 4.3.0 is installed for all users on your system, regardless of which is the latest release. We can go further too; to install Ansible but only for your user account, you can run the following command:

```
pip install --user ansible
```

One particularly handy trick is that when you start working with pip, you can use Python virtual environments to sandbox specific versions of Python modules. For example, you could create one virtual environment for Ansible 2.9 as follows:

1. Create the virtual environment in a suitable directory using the following command:

```
virtualenv ansible-2.9
```

This will create a new virtual environment in the directory where you run the command, with the environment (and directory containing it) being called `ansible-2.9`.

2. Activate the virtual environment as follows:

```
source ansible-2.9/bin/activate
```

3. Now you are ready to install Ansible 2.9. To install the latest version of Ansible 2.9, we will need to tell `pip` to install a version greater than (or equal to) 2.9, but less than 2.10, otherwise it would simply install Ansible 4.3:

```
pip install 'ansible>=2.9,<2.10'
```

4. Now, if you check your Ansible version, you should find that you are running the latest minor version of 2.9:

```
ansible --version
```

The downside to using virtual environments is that you need to remember to run the `source` command from *step 2* every time you log into your Ansible control machine. However, the upside is that you could repeat the preceding process with Ansible 4.3 in a separate virtual environment as follows:

```
virtualenv ansible-4.3
source ansible-4.3/bin/activate
pip install 'ansible>=4.3,<4.4'
ansible --version
```

The great thing about this is that you can now switch between the two versions of Ansible at will, simply by issuing the appropriate source command for the appropriate environment and then running Ansible in the usual way. This could be especially useful if you are in the process of migrating code from Ansible 2.9 to 4.3, or have some legacy code that won't yet work and you still need it before you have time to make the required changes.

Finally, if you want to upgrade your new installation of Ansible, you simply need to issue the appropriate pip command depending on your install method. For example, if you installed Ansible for all users, you would issue the following command:

```
sudo pip install -U ansible
```

If you had installed it only for your user account, the command would be similar:

```
pip install -U ansible
```

Now if you are working in a virtual environment, you must remember to activate the environment first. Once this is done, you can upgrade in the same way as before:

```
source ansible-2.9/bin/activate
```

```
pip install -U ansible
```

Note that the preceding example will upgrade whatever is installed in the Ansible 2.9 environment to the very latest version, which right now is 4.0. Also, a point to be noted is that, as discussed in the preceding section, *Upgrading from earlier Ansible installations*, this will break the install. To upgrade to the latest minor version, remember that you can specify version criteria just as we did when installing Ansible in this environment:

```
pip install -U 'ansible>=2.9,<2.10'
```

You can of course apply the version constraints to any of the other examples too. Their use is not limited in any way to a virtual environment.

Hopefully, by now you should have a pretty good idea of how to install Ansible 4.3, either from scratch, or to upgrade from an earlier installation. With this done, it's time we took a look at **Ansible Collections** as they are the driver behind all these changes.

## What are Ansible Collections?

Ansible Collections represent a major departure from the traditional monolithic approach to Ansible releases, where over 3,600 modules were being released along with the Ansible executables at one point. This, as you can imagine, was making Ansible releases unmanageable, and also meant that end users had to wait for an entirely new release of Ansible to receive a feature update or bug fix to a single module—obviously a very inefficient approach.

Thus, Ansible Collections were born, and their premise is quite simple: they are a mechanism for building, distributing, and consuming multiple different types of Ansible content. When you first migrate from Ansible 2.9 or earlier, your experience with Ansible Collections will come in the form of modules. As we discussed earlier in this chapter, what we call Ansible 4.3 is actually a package comprising around 85 collections...it does not contain the Ansible executables at all! Each of these collections contains a number of different modules, some maintained by the community, some maintained by specific vendors. Ansible 4.3 depends upon `ansible-core` 2.11.x, and this package contains the Ansible executables and the core `ansible.builtin` modules only (such as `debug`, `file`, and `copy`).

Let's take a look in more detail at the anatomy of a collection so that we can understand more fully how they work. Each collection has a name comprising two parts: the namespace and the collection name.

For example, the `ansible.builtin` collection has a namespace of `ansible` and a collection name of `builtin`. Similarly, in *Chapter 1, The System Architecture and Design of Ansible*, we installed a collection called `amazon.aws`. Here, `amazon` is the namespace and `aws` is the collection name. All namespaces must be unique, but collection names can be the same within a namespace (thus you could theoretically have `ansible.builtin` and `amazon.builtin`).

Although you can work with collections in a number of ways, including simply building and installing them locally all from your own machine, or directly from a Git repository, the central home for collections is **Ansible Galaxy**, and it is here that you will find all the collections included with the Ansible 4.3 package, along with many more. The Ansible Galaxy website is accessible at <https://galaxy.ansible.com> and there is a command-line tool (which we saw in *Chapter 1, The System Architecture and Design of Ansible*) called `ansible-galaxy` that can be used to interact with this website (for example, to install collections). We will use this tool quite extensively throughout the rest of this chapter, so you will get a chance to get better acquainted with it.

You can freely create your own account on Ansible Galaxy by logging in with your GitHub credentials, and when you do, your namespace is automatically created to be the same as your GitHub username. You can learn more about Ansible Galaxy namespaces here: <https://galaxy.ansible.com/docs/contributing/namespaces.html>.

Now that you have an understanding of how Ansible Collection names are created, let's take a deeper look at how Collections are put together and how they work.

## The anatomy of an Ansible collection

The easiest way to understand how a collection works under the hood is to build a simple one for ourselves, so let's get started on that. As with all aspects of Ansible, the developers have produced a system in collections that is powerful yet easy to work with, and if you already have prior experience of working with Ansible roles you will find collections work in a similar way. If you haven't, however, don't worry; we'll teach you all you need to know here.

A collection comprises a series of directories, each with a special name and each intended to hold a specific type of content. Any of these directories can be empty; you don't have to include all types of content in a collection. In fact, there is only one mandatory file in a collection! Ansible even provides a tool to help you build an empty collection to get started with. Let's use this now to create a new empty collection to learn with by running the following command:

```
ansible-galaxy collection init masterybook.demo
```

When you run this, you should see that it creates a directory tree as follows:

```
masterybook/
| -- demo
|   | -- README.md
|   | -- docs
|   | -- galaxy.yml
|   | -- plugins
|   |   | -- README.md
|   | -- roles
```

You can see from the preceding directory tree that this command created a top-level directory using our `masterybook` namespace, and then a subdirectory with the collection name of `demo`. It then created two files and three directories.

The purpose of these is as follows:

- `README.md`: This is the README file for the collection and should provide helpful information to anyone who is looking at the module code for the first time.
- `docs`: This directory is used to store general documentation for the collection. All documentation should be in Markdown format and should not be placed in any subfolders. Modules and plugins should still have their documentation embedded using Python docstrings, which we will learn more about in *Chapter 10, Extending Ansible*.
- `galaxy.yml`: This is the only mandatory file in the collection structure and contains all the information necessary to build the collection, including version information, author details, license information, and so on. The file created by the command run previously is a complete template with comments to explain each parameter, so you should find it easy to go through it and complete it to your requirements.
- `plugins`: This directory should contain all the Ansible plugins that you develop. Modules should also be included in separate modules/subdirectories, which you will need to create under the `plugins` folder. We will learn about creating plugins and modules for Ansible in *Chapter 10, Extending Ansible*.
- `roles`: Before Ansible 3.0, Ansible Galaxy existed only to distribute roles: reusable sets of Ansible code that can readily be distributed and used elsewhere to solve common automation challenges. We will learn all about roles in *Chapter 8, Composing Reusable Ansible Content with Roles*, so don't worry for now if you haven't come across them yet. Roles can still be distributed using Ansible Galaxy but can also be included in collections, which in time will probably become the norm.

In addition to this, collections can also contain the following:

- `tests`: This directory is used to store files related to testing Ansible Collections prior to release, and to be included in the top-level Ansible package, collections must pass the Ansible test process. You don't need to do this to use your own collection internally, but if you want it included in the main Ansible package you will have to complete this part of the development process. More details are available here: [https://docs.ansible.com/ansible/latest/dev\\_guide/developing\\_collections.html#testing-collections](https://docs.ansible.com/ansible/latest/dev_guide/developing_collections.html#testing-collections).

- `meta/runtime.yml`: This file and directory is used to specify important metadata about the collection, such as the version of the `ansible-core` package required, and various namespace routing and redirection stanzas to assist with the migration from Ansible 2.9 and earlier (where there were no namespaces) to Ansible 4.3 and beyond.
- `playbooks`: This directory will be supported in future versions of Ansible to include playbooks with the collection, though the official documentation on this is not complete at the time of writing.

Now that you've created and understood the collection directory structure, let's add our own module to it. When we've done this, we'll package it up and then install it on our system and use it in a playbook: a complete end-to-end test of how collections work. We'll borrow the module code for this from *Chapter 10, Extending Ansible*, so don't worry about understanding this code in depth at this stage as it is fully explained there. The full code listing is several pages long so we won't repeat it in the book here. Download the code accompanying this book or refer to the code listings in *Chapter 10, Extending Ansible*, to obtain the `remote_copy.py` module code. It is included in the `Chapter10/example08/library` directory of the example code accompanying this book.

Create a `modules/` subdirectory inside the `plugins/` directory, and add the `remote_copy.py` code there.

When you have reviewed the information in `galaxy.yml`, feel free to add your own name and other details in there, and you're done! That's all there is to creating your first collection. It really is beautifully simple, a set of files in a well-ordered directory structure.

**Tip**

Ansible Collections are expected to follow semantic versioning, as discussed earlier in this chapter, so be sure to adopt this when you create and build your own modules.

Your completed module directory structure should look something like this:

```
masterybook/
| -- demo
|   | -- README.md
|   | -- docs
|   | -- galaxy.yml
|   | -- plugins
```

```
| -- modules
|   | -- remote_copy.py
|   | -- README.md
| -- roles
```

When all the files are in place, it's time to build your collection. This is very simple, and is done by changing up to the same collection top-level directory (the one where `galaxy.yml` resides) and running this command:

```
cd masterybook/demo
ansible-galaxy collection build
```

This creates a tarball containing your collection files, which you can now use as you wish! You could publish this straight away to Ansible Galaxy, but first, let's test it locally to see if it works.

By default, Ansible stores collections locally in your home directory, under `~/ ansible/collections`. However, as we are testing a collection we just built, let's alter the behavior of Ansible slightly and install it in a local directory.

To try this out, create a new empty directory for a simple test playbook to reside in, and then create a directory called `collections` for us to install our newly created collection in:

```
mkdir collection-test
cd collection-test
mkdir collections
```

By default, Ansible won't know to look in this directory for collections, so we must override its default configuration to tell it to look in here. Within your directory, create a new `ansible.cfg` file (this file is always read if present and overrides the settings in any central configuration file, for example, `/etc/ansible/ansible.cfg`). This file should contain the following:

```
[defaults]
collections_paths=~/collections:~/ansible/collections:/usr/
share/ansible/collections
```

This configuration directive tells Ansible to look in the `collections` subdirectory within our current directory before checking the default locations on the system.

Now you're ready to install the collection we built earlier. Assuming you built this in your home directory, the command to do this is as follows:

```
ansible-galaxy collection install ~/masterybook/demo/  
masterybook-demo-1.0.0.tar.gz -p ./collections
```

If you explore your local `collections` directory, you should find it now contains the collection you created earlier, plus a couple of extra files created during the build process.

Finally, let's create a simple playbook to make use of our module. As a spoiler to *Chapter 10, Extending Ansible*, this module performs a simple file copy on the system Ansible is controlling, so let's create a test file in a publicly writeable directory such as `/tmp`, and put our module to work creating a copy. Consider the following playbook code:

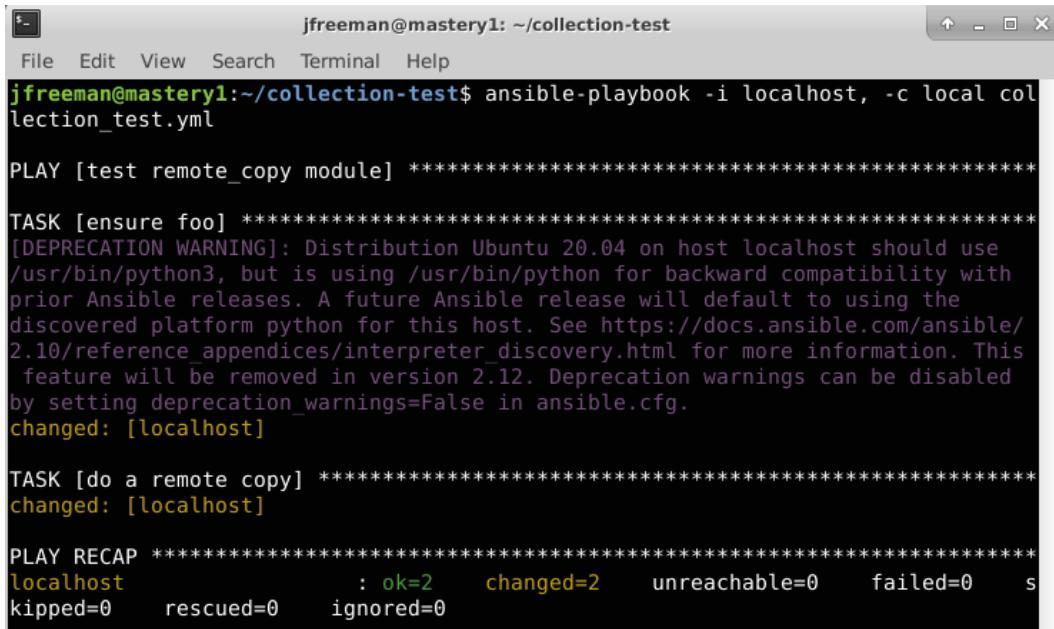
```
---  
- name: test remote_copy module  
  hosts: localhost  
  gather_facts: false  
  
  tasks:  
    - name: ensure foo  
      ansible.builtin.file:  
        path: /tmp/rcfoo  
        state: touch  
  
    - name: do a remote copy  
      masterybook.demo.remote_copy:  
        source: /tmp/rcfoo  
        dest: /tmp/rcbar
```

We have two tasks in our playbook here. One uses the `file` module from the `ansible.builtin` collection to create an empty file for our module to copy. The second task uses our new module, referencing it with the fully qualified collection name, to copy the file.

You can run this playbook code in the normal manner. For example, to run it against your local machine, run this command:

```
ansible-playbook -i localhost, -c local collection_test.yml
```

Note the comma after the `localhost` inventory item. This tells Ansible we are listing inventory hosts on the command line rather than having to create a local inventory file—a handy little shortcut when you're testing code! If all goes according to plan, your playbook run should look as shown in *Figure 2.1*:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/collection-test". The terminal displays the output of running the playbook `collection_test.yml`. The output shows two tasks being executed on the host `localhost`: one to ensure a file named `foo` exists, and another to do a remote copy. Both tasks result in changes. The final recap summary shows `ok=2`, `changed=2`, and no other metrics.

```
jfreeman@mastery1:~/collection-test$ ansible-playbook -i localhost, -c local collection_test.yml

PLAY [test remote_copy module] ****
TASK [ensure foo] ****
[DEPRECATION WARNING]: Distribution Ubuntu 20.04 on host localhost should use /usr/bin/python3, but is using /usr/bin/python for backward compatibility with prior Ansible releases. A future Ansible release will default to using the discovered platform python for this host. See https://docs.ansible.com/ansible/2.10/reference_appendices/interpreter_discovery.html for more information. This feature will be removed in version 2.12. Deprecation warnings can be disabled by setting deprecation_warnings=False in ansible.cfg.
changed: [localhost]

TASK [do a remote copy] ****
changed: [localhost]

PLAY RECAP ****
localhost                  : ok=2    changed=2    unreachable=0    failed=0    s
kipped=0      rescued=0      ignored=0
```

Figure 2.1 – The output from running our example playbook against our demo collection

Congratulations, you have just created, built, and run your first Ansible collection! Collections are often more complex than this, of course, and may contain many modules, plugins, and even roles and other artifacts, as outlined earlier. However, to get started, this is all you need to know.

Your final step when you are happy with your collection may very well be to publish it to Ansible Galaxy. Assuming you have already logged in to Ansible Galaxy and created your namespace, you simply need to navigate to your profile preferences page and click the **Show API Key** button, as shown in *Figure 2.2*:

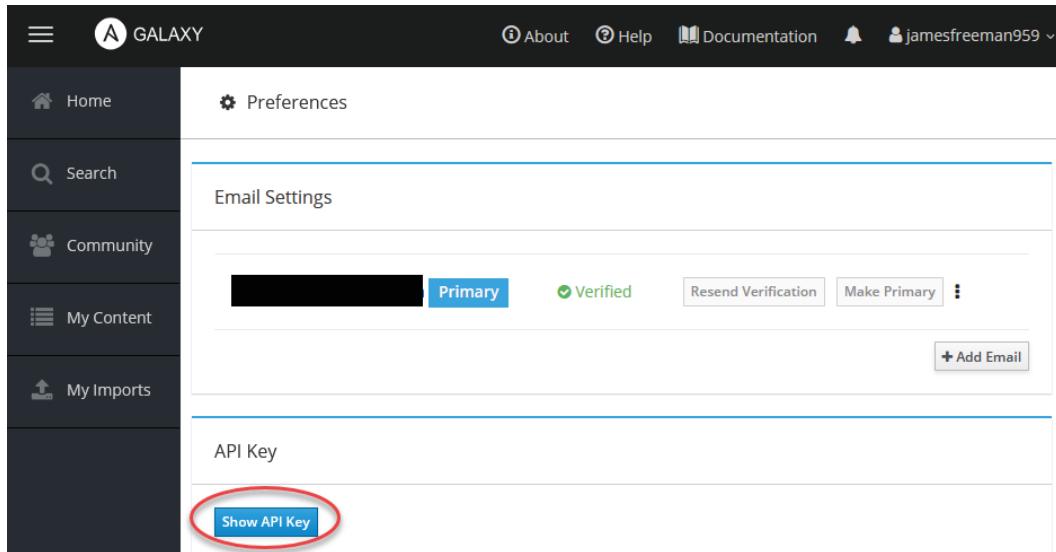


Figure 2.2 – Obtaining your API key from Ansible Galaxy

You can then feed this API key into the `ansible-galaxy` command-line tool to publish your collection. For example, to publish our collection from this chapter, you could run the following command:

```
ansible-galaxy collection publish ~/masterybook/demo/  
masterybook-demo-1.0.0.tar.gz --token=<API key goes here>
```

That concludes our look at collections and how they are built and used. As we mentioned, there are several ways to install collections, and indeed, Ansible modules are now distributed across a variety of collections. In the next section, we will take a look at ways to locate the module you require, and how to install and reference collections from within your automation code.

# Installing additional modules with ansible-galaxy

Most of the time when you work with collections, you won't be building them yourself. There are already 780 available on Ansible Galaxy at the time of writing and, there will probably be many more by the time you read this book. Nonetheless, it is the author's personal belief that we all learn better when we can get our hands dirty, and thus, developing our own, albeit simple, collection was a great way for us to look at how they are put together and how they are referenced.

However, let's focus now on finding and then working with pre-existing collections on Ansible, as this is where your focus is likely to be most of the time. As we have already mentioned, the Ansible 4.3 package includes a set of collections for you to begin your automation journey with, along with the `ansible.builtin` collection included with the `ansible-core` package.

If you want to see which collections got installed when you installed Ansible 4.3 on your system, simply run the following command:

```
ansible-galaxy collection list
```

This will return a list of all the installed collections in the format `<namespace>.<collection>`, along with their version numbers. Remember that collections are now independent of the Ansible version you install, so you can upgrade them without upgrading your entire Ansible installation. We will look at this shortly. The full list of collections installed as part of Ansible can also be found here: <https://docs.ansible.com/ansible/latest/collections/index.html>.

When you come to need a module for a specific purpose, it is worth noting that collections are normally named to give you clues about what they contain. For example, suppose you want to perform some cloud provisioning in Amazon Web Services with Ansible; a quick glance at the collections index reveals two likely candidates: the `amazon.aws` collection and the `community.aws` one. Similarly, if you want to automate the functionality of a Cisco IOS switch, the `cisco.ios` collection looks like a good place to start. You can explore the modules present in each collection on the Ansible documentation website, or explore the modules in a collection by making use of the `ansible-doc` command. For example, to list all the modules contained in the `cisco.ios` collection, you could run the following command:

```
ansible-doc -l cisco.ios
```

The `community.*` packages aim to provide the same functionality that was present in Ansible 2.9, naturally with newer versions of modules and plugins, thus helping you to port playbooks from earlier Ansible versions without too much pain.

Of course, if you can't find what you need in the Ansible 4.3 package, you can simply head over to the Ansible Galaxy website to find many more.

Once you have established which collections you are going to need for your playbook development, it's time to install them. We have already seen in the previous section that we can install a collection directly from a local file on disk. In *Chapter 1, The System Architecture and Design of Ansible*, we ran the following command:

```
ansible-galaxy collection install amazon.aws
```

This installed the latest version of the `amazon.aws` collection directly from Ansible Galaxy. The eagle-eyed among you might be thinking, "hold on, `amazon.aws` is already included as part of the Ansible 4.3 package." Indeed, it is. However, the decoupled nature of Ansible and its collections means we are free to install and upgrade collection versions without having to upgrade Ansible. Indeed, when we ran the preceding command, it installed the latest version of `amazon.aws` inside the users local collections path (`~/.ansible/collections`) as this is the default. Note that this is different to the behavior observed when we tested our own collection earlier in this chapter, as we specifically created an Ansible configuration file specifying a different collections path.

We find out what happened by running another collection list using the `ansible-galaxy` command, only this time we will only filter on the `amazon.aws` collection:

```
ansible-galaxy collection list amazon.aws
```

The output will be something like this:

```
jfreeman@mastery1: ~$ ansible-galaxy collection list amazon.aws
# /home/jfreeman/.ansible/collections/ansible_collections
Collection Version
-----
amazon.aws 1.4.0

# /usr/local/lib/python3.8/dist-packages/ansible_collections
Collection Version
-----
amazon.aws 1.3.0
```

Figure 2.3 – Listing multiple versions of an installed collection

Here, we can see that the `1.3.0` version of this collection was installed alongside our Ansible installation itself, but that the later `1.4.0` version is installed in my home directory's `.ansible/collections` folder, the latter taking precedence when a playbook references it and is run from my user account. Note that playbooks run from other user accounts on this system would only see the version `1.3.0` collection as this is installed system wide, and they would not normally be referencing the folder in my home directory.

As you might expect, you can specify the version of a collection you want when you install it. If I had wanted to install the latest development version of the `amazon.aws` collection, I could have installed it locally using the following command:

```
ansible-galaxy collection install amazon.aws==1.4.2-dev9  
--force
```

The `--force` option is required as `ansible-galaxy` won't overwrite a release version of a collection with a development version unless you force it to—a sensible safety precaution!

As well as installing collections from local files and from Ansible Galaxy, you can also install them directly from a Git repository. For example, to install the latest commit on the `stable` branch of a hypothetical GitHub repository, you could run the following command:

```
ansible-galaxy collection install git+https://github.com/  
jamesfreeman959/repo_name.git,stable
```

There are many possible permutations here, including accessing private Git repositories and even local ones.

All of these are perfectly valid ways to install collections. However, imagine you require ten different collections for your playbook to run successfully. The last thing you want to do is to have to run ten different `ansible-galaxy` commands every time you deploy the automation code somewhere new! Plus, this could very easily get out of hand, with different collection versions on different hosts.

Thankfully, Ansible has your back here too, and the `requirements.yml` file (which was present in earlier versions of Ansible and used to install roles from Ansible Galaxy before collections became a reality) can be used to specify a set of collections to install.

As an example, consider the following `requirements.yml` file:

```
---
collections:
- name: geerlingguy.k8s

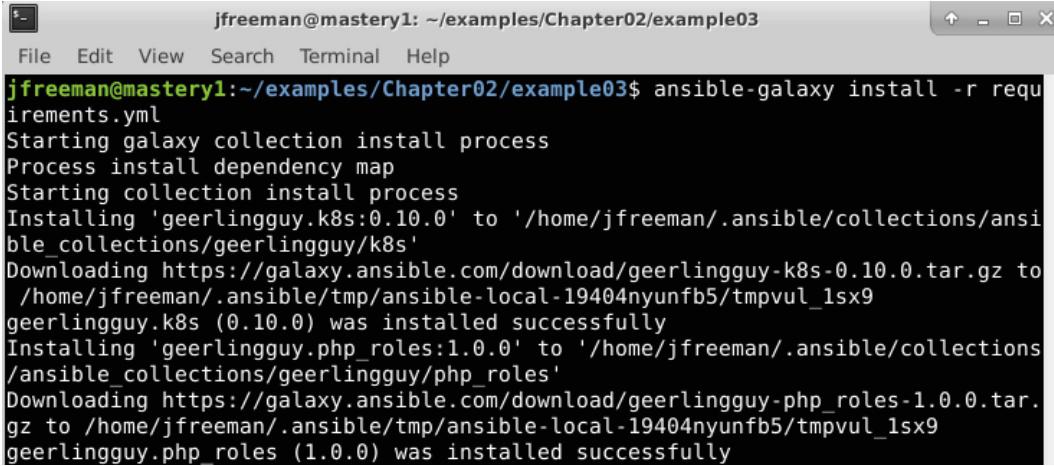
- name: geerlingguy.php_roles
  version: 1.0.0
```

This file describes a requirement for two collections. The namespace for both is `geerlingguy`, and the collections are called `k8s` and `php_roles`. The `k8s` collection will have the latest stable version installed, whereas only version `1.0.0` of the `php_roles` collection will be installed regardless of the latest release version.

To install all the requirements specified in `requirements.yml`, simply run the following command:

```
ansible-galaxy install -r requirements.yml
```

The output of the command should look something like *Figure 2.4*:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter02/example03". The window shows the command "ansible-galaxy install -r requirements.yml" being run. The output details the installation process for two collections: "geerlingguy.k8s" and "geerlingguy.php\_roles". Both collections are installed at their specified versions (0.10.0 and 1.0.0 respectively) from the "geerlingguy" namespace.

```
jfreeman@mastery1:~/examples/Chapter02/example03$ ansible-galaxy install -r requirements.yml
Starting galaxy collection install process
Process install dependency map
Starting collection install process
Installing 'geerlingguy.k8s:0.10.0' to '/home/jfreeman/.ansible/collections/ansible_collections/geerlingguy/k8s'
Downloading https://galaxy.ansible.com/download/geerlingguy-k8s-0.10.0.tar.gz to /home/jfreeman/.ansible/tmp/ansible-local-19404nyunfb5/tmpvul_1sx9
geerlingguy.k8s (0.10.0) was installed successfully
Installing 'geerlingguy.php_roles:1.0.0' to '/home/jfreeman/.ansible/collections/ansible_collections/geerlingguy/php_roles'
Downloading https://galaxy.ansible.com/download/geerlingguy-php_roles-1.0.0.tar.gz to /home/jfreeman/.ansible/tmp/ansible-local-19404nyunfb5/tmpvul_1sx9
geerlingguy.php_roles (1.0.0) was installed successfully
```

Figure 2.4 – Installing collections using a requirements.yml file

As you can see from this output, both collections that we specified in the `requirements.yml` file have been installed at the appropriate versions. This is a very simple and powerful way to capture the collection requirements for your playbooks, and to have them all installed in one go, while retaining the correct versions where this is required.

At this stage, you should have a robust understanding of the big changes in Ansible 4.3, especially collections, how to find the right ones for your automation needs, and how to install them (and even how to create your own if you need to!). In the final part of this chapter, we will provide a brief primer on how to port your playbooks to Ansible 4.3 from version 2.9 and earlier.

## How to port legacy playbooks to Ansible 4.3 (a primer)

No two Ansible playbooks (or roles or templates for that matter) are alike, and they vary in complexity from the simple to the intricate and complex. However, they are all important to their authors and users, and with all the major changes that were made in the transition from Ansible 2.9 through to 4.0, this book would not be complete without a primer on how to port your code to the newer Ansible versions.

Before we get too deep into this subject, let's look at an example. In the first ever edition of this book, written in 2015 about Ansible version 1.9, an example appeared that renders a **Jinja2** template using a small Ansible playbook. We will still learn about an updated version of this code in *Chapter 6, Unlocking the Power of Jinja2 Templates*, of this book, but for now let's look at the original code. The template, called `demo.j2`, looks like this:

```
setting = {{ setting }}
{% if feature.enabled %}
feature = True
{% else %}
feature = False
{% endif %}
another_setting = {{ another_setting }}
```

The playbook that renders this template looks like this:

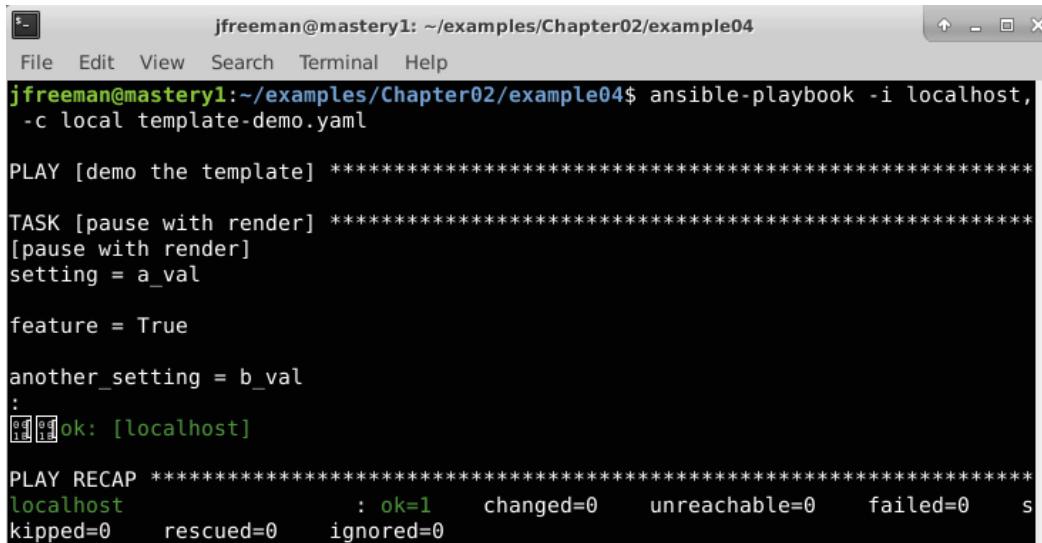
```
---
- name: demo the template
  hosts: localhost
  gather_facts: false
  vars:
    setting: a_val
    feature:
      enabled: true
```

```
another_setting: b_val
tasks:
  - name: pause with render
    pause:
      prompt: "{{ lookup('template', 'demo.j2') }}"
```

This is the exact same code that appeared in the first edition of this book, and as it was written for Ansible 1.9 and so much has changed in the transition to 4.3, you could be forgiven for thinking that this code would never run on Ansible 4.3. However, let's do exactly that. We'll run this code with the following command:

```
ansible-playbook -i localhost, -c local template-demo.yaml
```

The output from this command, run on Ansible 4.3 with ansible-core 2.11.1, looks like *Figure 2.5*:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter02/example04". The window shows the command "ansible-playbook -i localhost, -c local template-demo.yaml" being run. The output shows the playbook running on a single host, "localhost", which is listed twice in the inventory. The tasks are executed sequentially: a "pause with render" task, followed by setting "another\_setting" to "b\_val", and finally a "feature = True" task. The output concludes with a recap showing 1 host managed, 1 ok, 0 changed, 0 unreachable, 0 failed, 0 skipped, 0 rescued, and 0 ignored.

```
jfreeman@mastery1:~/examples/Chapter02/example04$ ansible-playbook -i localhost, -c local template-demo.yaml

PLAY [demo the template] ****
TASK [pause with render] ****
[pause with render]
setting = a_val
feature = True

another_setting = b_val
:
ok: [localhost]

PLAY RECAP ****
localhost          : ok=1    changed=0    unreachable=0    failed=0    s
kiped=0    rescued=0    ignored=0
```

Figure 2.5 – Running an example playbook from the first edition of this book on Ansible 4.3

You would be forgiven for asking, why does this work, and why all the detail about collections when code that was first written for Ansible 1.9 still works in 4.3 without modification? Ansible 4.3 was coded specifically to provide users with the least painful path possible, and it is even stated in the porting guide for Ansible 2.10:

*Your playbooks should continue to work without any changes.*

This will hold true as long as the module names remain unique. However, there is nothing to stop module name clashes any more—they only need to be unique within their own collection now. So, for example, we used the `pause` module in the preceding playbook, which has a **fully qualified collection name (FQCN)** of `ansible.builtin.pause` in Ansible 4.3. The preceding code worked because there was no other module called `pause` within our collections. However, consider the `masterybook.demo` collection we created earlier in this chapter. There is nothing to stop us from creating our own module called `pause` in here that does something completely different. How would Ansible know which module to choose?

The answer comes from inside Ansible itself, which has been coded to search all of the collections that form part of the Ansible 4.3 package; thus, a reference to `pause` resolves to `ansible.builtin.pause`. It will never resolve to `masterybook.demo.pause` (assuming we created that module) and so we would need to use the FQCN if we wanted to use our hypothetical module in a task.

The recommendation from Ansible on this topic is to always use the FQCNs in your code to make sure that you never receive unexpected results from a module name clash. However, what if you wanted to avoid a lot of typing in a set of tasks? For example, typing `masterybook.demo.remote_copy` is a lot of typing if you have to do it repetitively.

The answer comes in the form of a new `collections` : key defined at the play level in your playbook. When we tested our newly built collection earlier in this chapter, we used the FCQN to reference it. However, that same playbook could have been written as follows:

```
---
- name: test remote_copy module
  hosts: localhost
  gather_facts: false
  collections:
    - masterybook.demo

  tasks:
    - name: ensure foo
      ansible.builtin.file:
        path: /tmp/rcfoo
        state: touch

    - name: do a remote copy
```

```
remote_copy:  
  source: /tmp/rcfoo  
  dest: /tmp/rcbar
```

Note the presence of the `collections`: key up at the play level. This in essence creates an ordered *search path* for references that are not specified by FQCNs. Thus, we have instructed our play to search the `masterybook.demo` namespace for modules, roles, and plugins before searching the included namespaces, such as `ansible.builtin`. Indeed, you can change the module reference on the `ensure foo` task from `ansible.builtin.file` to `file`, and the play will still work as intended. The `collections` directive does not overwrite these internal namespace search paths, it simply prepends namespaces to it.

Of note though, is that when you start working with roles (which we will cover later in this book), the collections search path specified in the play does not get inherited by the roles, so they will all need to have this defined manually. You can define the collections search paths for a role by creating a `meta/main.yml` file within your role, which could contain, for example, the following:

```
collections:  
  - masterybook.demo
```

In addition, it is important to mention that these collection search paths do not affect items such as lookups, filters or tests that you might include in your collection. For example, if we included a lookup in our collection, it would need to be referenced using the FQCN regardless of whether the `collections` key appears in the play or role. Finally, note that you must always install your collections as demonstrated earlier in this chapter. Including the `collections` keyword in your code does not cause Ansible to automatically install or download the collections; it is simply a search path for them.

On balance, you will probably find it easier to work with FQCNs throughout your code, but the important lesson from this part of the section is that while it is best practice to use FQCNs throughout your code, it is by no means mandatory at this time, and if you are upgrading to Ansible 4.3, you don't have to go through every single playbook you ever wrote and update all the references to modules, plugins, and so on. You can do this over time, but it is advisable to do it.

Of course, if we review all the changes that have happened in Ansible since even the 2.7 release, which the third edition of this book was based on, there are many. However, they will only affect certain playbooks as they relate to the specific behavior of certain play aspects, or to the way some modules work. Indeed, some modules get deprecated and removed as newer releases of Ansible are produced, and new ones get added.

Whenever you are looking to upgrade your Ansible installation, it is advisable to review the porting guides that are produced by Ansible for each release since 2.0. They can be found here: [https://docs.ansible.com/ansible/devel/porting\\_guides/porting\\_guides.html](https://docs.ansible.com/ansible/devel/porting_guides/porting_guides.html).

As for the example we started this chapter off with, you may very well find that your code needs no modifications at all. However, it is always best to plan your upgrade rather than simply hope for the best, only to hit some unexpected behavior that breaks your automation code.

It is hoped that this section on playbook porting has shown you how to handle the introduction of collections in your playbooks, and given you some pointers about where to look for guidance when you upgrade Ansible.

## Summary

Since the last release of this book, there have been many changes to Ansible, but the most notable (which is expected to impact everyone reading this book) is the introduction of collections to manage modules, roles, plugins, and more, and decoupling them from the core release of Ansible. Probably the most noticeable change to Ansible code is in the introduction of FQCNs and the need to install collections if they are not part of the Ansible 4.3 package.

In this chapter, you learned about the reasons for the introduction of collections in Ansible, and how they impact everything from your playbook code to the way you install, maintain, and upgrade Ansible itself. You learned that collections are easy to build from scratch, and even how to build your own, before looking at ways to install and manage collections for your playbook. Finally, you learned the fundamentals of porting your Ansible code from earlier releases.

In the next chapter, you will learn how to secure secret data while working with Ansible.

## Questions

1. Collections can contain:
  - a) Roles
  - b) Modules
  - c) Plugins
  - d) All of the above
2. Collections mean that Ansible Module versioning is independent of the version of the Ansible engine.
  - a) True
  - b) False
3. The Ansible 4.3 package:
  - a) includes the Ansible automation engine.
  - b) has a dependency on the Ansible automation engine.
  - c) bears no relation to the Ansible automation engine.
4. It is possible to upgrade directly from Ansible 2.9 to Ansible 4.3.
  - a) True
  - b) False
5. In Ansible 4.3, module names are guaranteed to be unique between different namespaces.
  - a) True
  - b) False
6. To ensure that you always access the correct module you intend, you should start using which of the following now in your tasks?
  - a) Fully Qualified Domain Names
  - b) Short form module names
  - c) Fully Qualified Collection Names
  - d) None of the above

7. Which file can be used to list all the required Collections from Ansible Galaxy, ensuring they can easily be installed when needed?
  - a) `site.yml`
  - b) `ansible.cfg`
  - c) `collections.yml`
  - d) `requirements.yml`
8. When you create an account on Ansible Galaxy for the purposes of contributing your own Collections, your namespace is:
  - a) randomly generated.
  - b) chosen by you.
  - c) automatically generated based on your GitHub user ID.
9. Collections are stored in which common file format?
  - a) `.tar.gz`
  - b) `.zip`
  - c) `.rar`
  - d) `.rpm`
10. How could you list all the Collections installed with your Ansible package?
  - a) `ansible --list-collections`
  - b) `ansible-doc -l`
  - c) `ansible-galaxy --list-collections`
  - d) `ansible-galaxy collections list`



# 3

# Protecting Your Secrets with Ansible

Secrets are meant to stay secret. Whether they are login credentials to a cloud service or passwords to database resources, they are secret for a reason. Should they fall into the wrong hands, they can be used to discover trade secrets, customers' private data, create infrastructure for nefarious purposes, or worse. All of this could cost you and your organization a lot of time, money, and headaches! When the second edition of this book was published, it was only possible to encrypt your sensitive data in external vault files, and all data had to exist entirely in either an encrypted or unencrypted form. It was also only possible to use one single Vault password per playbook run, meaning it was not possible to segregate your secret data and use different passwords for items of different sensitivities. All that has now changed, with multiple Vault passwords permissible at playbook runtime, as well as the possibility of embedding encrypted strings in otherwise plain **YAML Ain't Markup Language (YAML)** files.

In this chapter, we will describe how to take advantage of these new features, and thus keep your secrets safe with Ansible, by covering the following topics:

- Encrypting data at rest
- Creating and editing encrypted files
- Executing `ansible-playbook` with encrypted files

- Mixing encrypted data with plain YAML
- Protecting secrets while operating

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running **Ansible 4.3** or newer. Almost any flavor of Linux should do—for those interested in specifics, all the code presented in this chapter was tested on Ubuntu Server 20.04 **Long Term Support (LTS)**, unless stated otherwise, and on Ansible 4.3. The example code that accompanies this chapter can be downloaded from GitHub at this **Uniform Resource Locator (URL)**: <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter03>.

Check out the following video to see the Code in Action: <https://bit.ly/2Z4xB42>

## Encrypting data at rest

As a configuration management system or an orchestration engine, Ansible has great power. To wield that power, it is necessary to entrust secret data to Ansible. An automation system that prompts the operator for passwords at each connection is not very efficient—indeed, it's hardly fully automated if you have to sit there and type in passwords over and over! To maximize the power of Ansible, secret data must be written to a file that Ansible can read and from which it can utilize the data.

This creates a risk, though! Your secrets are sitting there on your filesystem in plaintext. This is a physical as well as a digital risk. Physically, the computer could be taken from you and pored over for secret data. Digitally, any malicious software that can break the boundaries set upon it is capable of reading any data to which your user account has access. If you utilize a source control system, the infrastructure that houses the repository is just as much at risk.

Thankfully, Ansible provides a facility to protect your data at rest. That facility is **Vault**. This facility allows for the encryption of text files so that they are stored at rest in an encrypted format. Without the key or a significant amount of computing power, the data is indecipherable, yet can still be used within Ansible plays as easily as unencrypted data.

The key lessons to learn when dealing with encrypting data at rest include the following:

- Valid encryption targets
- Securing differing data with multiple passwords and vault **identifiers (IDs)**
- Creating new encrypted files
- Encrypting existing unencrypted files
- Editing encrypted files
- Changing the encryption password on files
- Decrypting encrypted files
- Encrypting data inline in an otherwise unencrypted YAML file (for example, a playbook)
- Running `ansible-playbook` while referencing encrypted files

## Vault IDs and passwords

Before the release of **Ansible 2.4**, it was only possible to use one Vault password at a time. While you could have multiple secrets for multiple purposes stored in several locations, only one password could be used. This was obviously fine for smaller environments, but as the adoption of Ansible has grown, so has the requirement for better and more flexible security options. For example, we have already discussed the potential for Ansible to manage both a development and production environment through the use of groups in the inventory. It is realistic to expect that these environments would have different security credentials. Similarly, you would expect core network devices to have different credentials from servers. In fact, it is a good security practice to do so.

Given this, it seems unreasonable to then protect any secrets under a single master password using Vault. Ansible 2.4 introduced the concept of Vault IDs as a solution, and while at present, the old single password commands are all still valid, it is recommended to use Vault IDs when working with Ansible on the command line. Each Vault ID must have one single password associated with it, but multiple secrets can share the same ID.

Ansible Vault passwords can come from one of the following three sources:

- A user-entered string, which Ansible will prompt for when it is required
- A flat text file containing the Vault password in plain unencrypted text (obviously, it is vital this file is kept secure!)
- An executable that fetches the password (for example, from a credential management system) and outputs it on a single line for Ansible to read

The syntax for each of these three options is broadly similar. If you only have one vault credential and hence aren't using IDs (although you could if you wanted to, and this is strongly recommended as you might later wish to add a second vault ID), you would, therefore, enter the following line of code to run a playbook and prompt for the Vault password:

```
ansible-playbook --vault-id @prompt playbook.yaml
```

If you want to obtain the Vault password from a text file, you would run the following command:

```
ansible-playbook --vault-id /path-to/vault-password-text-file
playbook.yaml
```

Finally, if you are using an executable script, you would run the following command:

```
ansible-playbook --vault-id /path-to/vault-password-script.py
playbook.yaml
```

If you are working with IDs, simply add the ID in front of the password source, followed by the @ character—if the ID for your vault is `prod`, for example, the three preceding examples become the following:

```
ansible-playbook --vault-id prod@prompt playbook.yaml
ansible-playbook --vault-id prod@/path-to/vault-password-text-
file playbook.yaml
ansible-playbook --vault-id prod@/path-to/vault-password-
script.py playbook.yaml
```

Multiple combinations of these can be combined into one command, as follows:

```
ansible-playbook --vault-id prod@prompt testing@/path-to/vault-
password-text-file playbook.yaml
```

We will use the `vault-id` command-line options throughout the rest of this chapter.

## Things Vault can encrypt

The Vault feature can be used to encrypt any **structured data** used by Ansible. This can either be almost any YAML (or **JavaScript Object Notation (JSON)**) file that Ansible uses during its operation or even a single variable within an otherwise unencrypted YAML file, such as a playbook or role. Examples of encrypted files that Ansible can work with include the following:

- `group_vars/` files
- `host_vars/` files
- `include_vars` targets
- `vars_files` targets
- `--extra-vars` targets
- Role variables
- Role defaults
- Task files
- Handler files
- Source files for the `copy` module (these are an exception in this list—they don't have to be YAML-formatted)

If a file can be expressed in YAML and read by Ansible, or if a file is to be transported with the `copy` module, it is a valid file for encryption in Vault. Because the entire file will be unreadable at rest, care should be taken to not be overzealous in picking which files to encrypt. Any source control operations with the files will be done with the encrypted content, making it very difficult to peer-review.

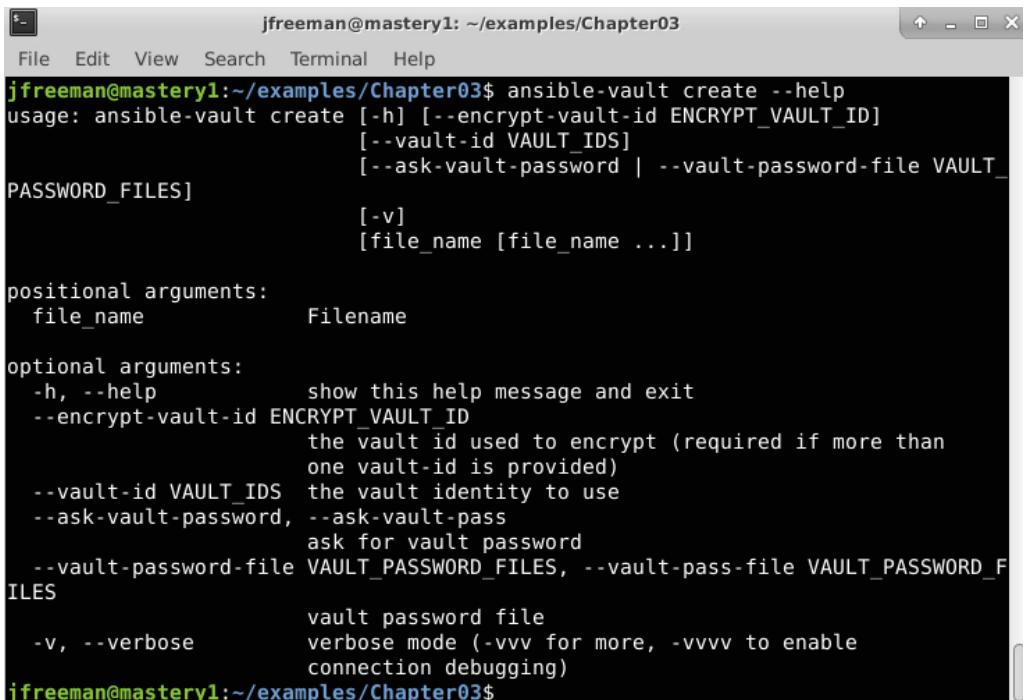
As a best practice, the smallest possible amount of data should be encrypted, which may even mean moving some variables into a file all by themselves. It is for this reason that Ansible 2.3 added the `encrypt_string` feature to `ansible-vault`, allowing for individual secrets to be placed inline with otherwise unencrypted YAML, saving the user from encrypting the entire file. We will cover this later in the chapter.

## Creating and editing encrypted files

To create new files, Ansible provides a program called `ansible-vault`. This program is used to create and interact with Vault-encrypted files. The subcommand to create encrypted files is `create`, and you can see the options available under this subcommand by running the following command:

```
ansible-vault create --help
```

The output of this command is shown in the following screenshot:



A terminal window titled "jfreeman@mastery1: ~/examples/Chapter03" displays the help output for the `ansible-vault create` command. The output shows the usage, positional arguments (file\_name), optional arguments (-h, --help, --encrypt-vault-id, --vault-id, --ask-vault-password, --vault-password-file), and other flags (-v, --verbose).

```
jfreeman@mastery1:~/examples/Chapter03$ ansible-vault create --help
usage: ansible-vault create [-h] [--encrypt-vault-id ENCRYPT_VAULT_ID]
                            [--vault-id VAULT_IDS]
                            [--ask-vault-password | --vault-password-file VAULT_
PASSWORD_FILES]
                            [-v]
                            [file_name [file_name ...]]

positional arguments:
  file_name            Filename

optional arguments:
  -h, --help           show this help message and exit
  --encrypt-vault-id ENCRYPT_VAULT_ID
                      the vault id used to encrypt (required if more than
                      one vault-id is provided)
  --vault-id VAULT_IDS the vault identity to use
  --ask-vault-password, --ask-vault-pass
                      ask for vault password
  --vault-password-file VAULT_PASSWORD_FILES, --vault-pass-file VAULT_PASSWORD_F
ILES
                      vault password file
  -v, --verbose        verbose mode (-vvv for more, -vvvv to enable
                      connection debugging)
jfreeman@mastery1:~/examples/Chapter03$
```

Figure 3.1 – The options available when creating an Ansible Vault instance

To create a new file, you'll need to know two things ahead of time. The first is the password `ansible-vault` will be using to encrypt the file, and the second is the filename itself. Once provided with this information, `ansible-vault` will launch a text editor (as defined in the `EDITOR` environment variable—this defaults to `vi` or `vim` in many cases). Once you save the file and exit the editor, `ansible-vault` will use the supplied password as a key to encrypt the file with the `AES256` cipher.

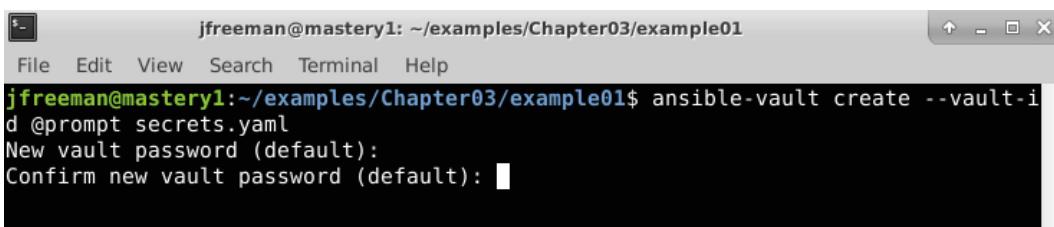
Let's walk through a few examples of creating encrypted files. First, we'll create one and be prompted for a password, then we will provide a password file, and lastly, we'll create an executable to deliver the password.

## Password prompt

Getting `ansible-vault` to request a password from the user at runtime is the easiest way to get started with vault creation, so let's go through a simple example and create a vault containing a variable we want to encrypt. Run the following command to create a new vault, and to be prompted for the password:

```
ansible-vault create --vault-id @prompt secrets.yaml
```

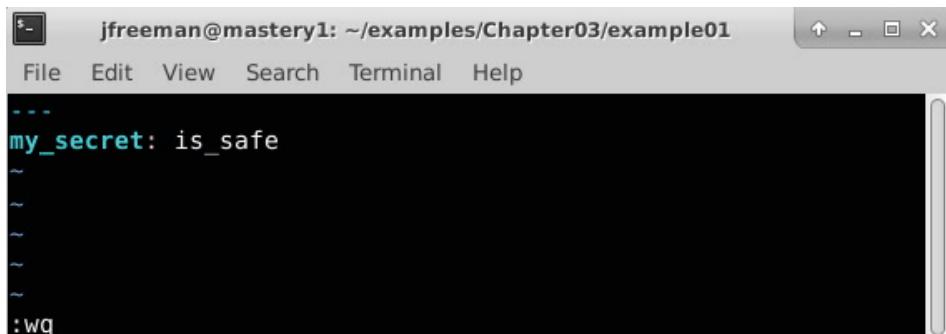
The output should look something like this:



```
jfreeman@mastery1: ~/examples/Chapter03/example01
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter03/example01$ ansible-vault create --vault-i
d @prompt secrets.yaml
New vault password (default):
Confirm new vault password (default):
```

Figure 3.2 – Creating a new Ansible Vault instance while being prompted for the password

Once the passphrase is entered, our editor opens and we're able to put content into the file, as shown in the following screenshot:



```
jfreeman@mastery1: ~/examples/Chapter03/example01
File Edit View Search Terminal Help
...
my_secret: is_safe
~
~
~
~
~
:wq
```

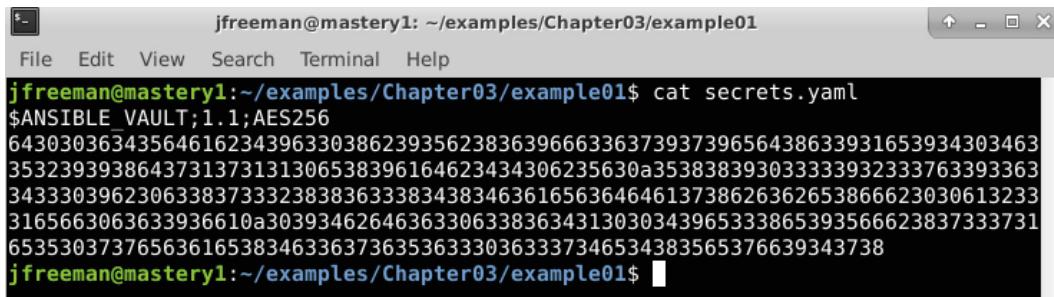
Figure 3.3 – Adding content to the new Ansible Vault instance using the vim editor

On my system, the configured editor is **Vim**. Your system may be different, and you may wish to set your preferred editor as the value for the `EDITOR` environment variable if you are not happy with the default selection.

Now, we save the file. If we try to read the content using the following command, we'll see that they are in fact encrypted:

```
cat secrets.yaml
```

There is just a small header hint for Ansible to use later, as shown in the following screenshot:



A terminal window titled "jfreeman@mastery1: ~/examples/Chapter03/example01". The window shows the command "cat secrets.yaml" being run, which outputs a large block of encrypted data. The data starts with "ANSIBLE\_VAULT;1.1;AES256" followed by several lines of binary-looking characters.

```
jfreeman@mastery1:~/examples/Chapter03/example01$ cat secrets.yaml
ANSIBLE_VAULT;1.1;AES256
6430306343564616234396330386239356238363966633637393739656438633931653934303463
35323939386437313731313065383961646234306235630a35383839303333932333763393363
3433303962306338373332383836333834383463616563646413738626362653866623030613233
3165663063633936610a303934626463633063383634313030343965333865393566623837333731
65353037376563616538346336373635363330363337346534383565376639343738
jfreeman@mastery1:~/examples/Chapter03/example01$
```

Figure 3.4 – Showing the content of our new Ansible Vault instance, which are encrypted at rest

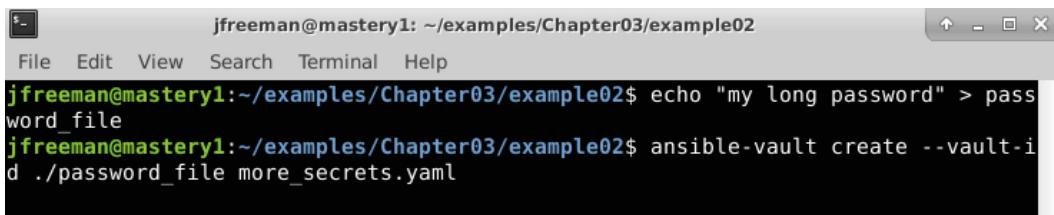
As you can see from the headers, AES256 is used for vault encryption, meaning that as long as you use a good password when creating your vault, your data is very secure.

## Password file

To use `ansible-vault` with a password file, you first need to create such a file. Simply echoing a password into a file can do this. Once complete, you can now reference this file when calling `ansible-vault` to create another encrypted file. Try this out by running the following commands:

```
echo "my long password" > password_file
ansible-vault create --vault-id ./password_file more_secrets.yaml
```

This should look something like the output shown in the following screenshot:



A terminal window titled "jfreeman@mastery1: ~/examples/Chapter03/example02". The window shows two commands being run: "echo 'my long password' > password\_file" and "ansible-vault create --vault-id ./password\_file more\_secrets.yaml".

```
jfreeman@mastery1:~/examples/Chapter03/example02$ echo "my long password" > password_file
jfreeman@mastery1:~/examples/Chapter03/example02$ ansible-vault create --vault-id ./password_file more_secrets.yaml
```

Figure 3.5 – Creating an Ansible Vault instance using a password file

When you run the preceding commands, you will note that you are not prompted for a password—this time, the password for the vault is the `my long password` string, which has been read from the content of `password_file`. The default editor will open, and data can be written just like before after this point, though.

## Password script

This last example uses a password script. This is useful for designing a system where a password can be stored in a central system for storing credentials and shared with contributors to the playbook tree. Each contributor could have their own password for the shared credentials store, where the Vault password would be retrieved from. Our example will be far more straightforward: just a simple output to STDOUT with a password. This file will be saved as `password.sh`. Create this file now with the following content:

```
#!/bin/sh
echo "a long password"
```

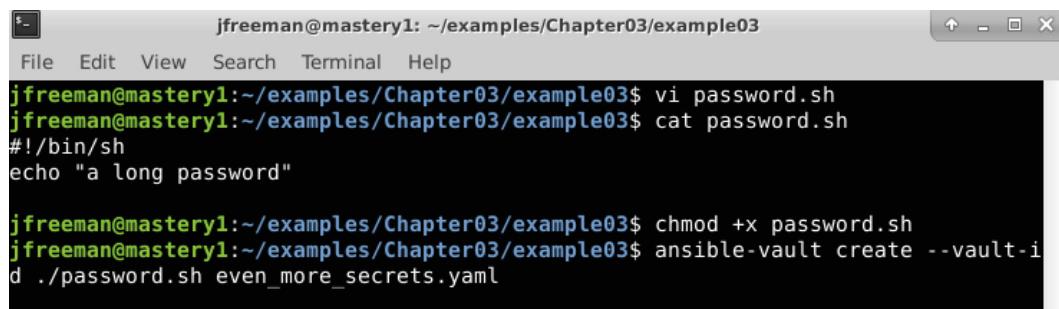
For Ansible to use this script, it must be marked as executable—run the following command against it so that it is:

```
chmod +x password.sh
```

Finally, you can create a new vault secured with the `a long password` as output by our simple script, by running the following command:

```
ansible-vault create --vault-id ./password.sh even_more_
secrets.yaml
```

The output from this process should look something like this:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter03/example03". The window shows the following command history:

```
jfreeman@mastery1:~/examples/Chapter03/example03$ vi password.sh
jfreeman@mastery1:~/examples/Chapter03/example03$ cat password.sh
#!/bin/sh
echo "a long password"

jfreeman@mastery1:~/examples/Chapter03/example03$ chmod +x password.sh
jfreeman@mastery1:~/examples/Chapter03/example03$ ansible-vault create --vault-id ./password.sh even_more_secrets.yaml
```

Figure 3.6 – Creating an Ansible Vault instance using a simple password script

Try this for yourself and see how it works—you should find that `ansible-vault` creates a vault with the `a long password` password, as written to STDOUT by the script. You could even try editing using the following command:

```
ansible-vault edit --vault-id @prompt even_more_secrets.yaml
```

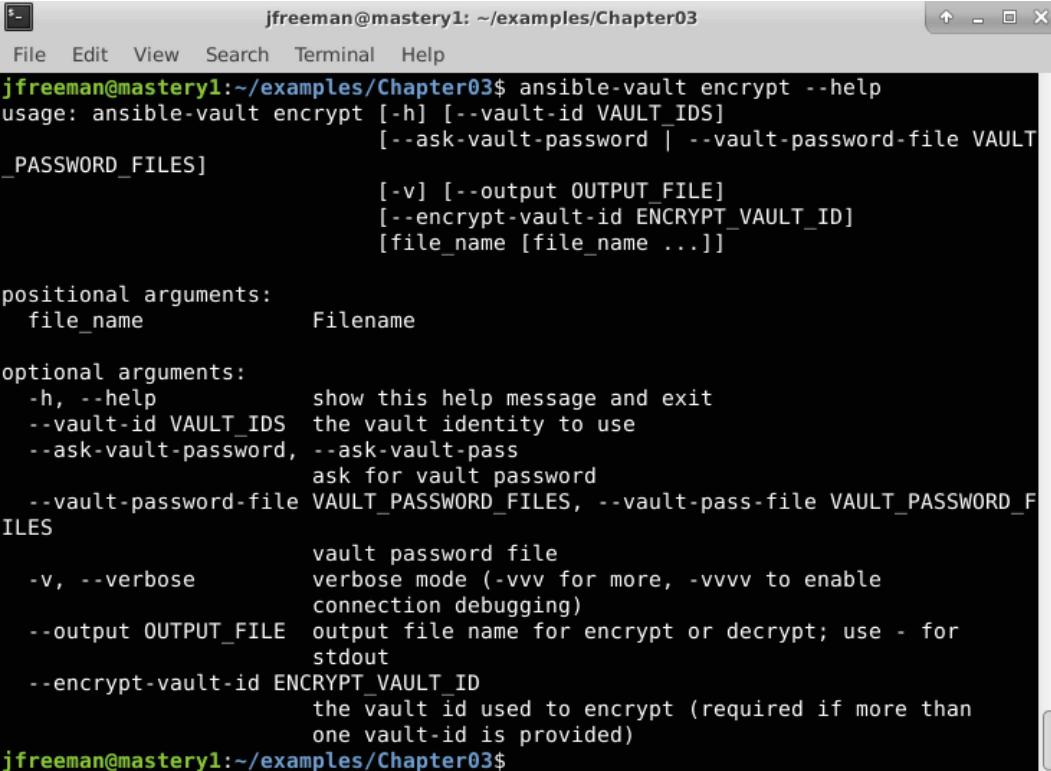
You should now see enter a `long password` when prompted—and you can now edit the vault successfully!

## Encrypting existing files

The previous examples all dealt with creating new encrypted files using the `create` subcommand. But what if we want to take an established file and encrypt it? A subcommand exists for this as well. It is named `encrypt`, and you can see the options for this subcommand by running the following command:

```
ansible-vault encrypt --help
```

The output will look similar to that shown in the following screenshot:



```
jfreeman@mastery1:~/examples/Chapter03$ ansible-vault encrypt --help
usage: ansible-vault encrypt [-h] [--vault-id VAULT_IDS]
                               [--ask-vault-password | --vault-password-file VAULT_PASSWORD_FILES]
                               [-v] [--output OUTPUT_FILE]
                               [--encrypt-vault-id ENCRYPT_VAULT_ID]
                               [file_name [file_name ...]]

positional arguments:
  file_name           Filename

optional arguments:
  -h, --help          show this help message and exit
  --vault-id VAULT_IDS  the vault identity to use
  --ask-vault-password, --ask-vault-pass
                        ask for vault password
  --vault-password-file VAULT_PASSWORD_FILES, --vault-pass-file VAULT_PASSWORD_FILES
                        vault password file
  -v, --verbose        verbose mode (-vvv for more, -vvvv to enable
                      connection debugging)
  --output OUTPUT_FILE  output file name for encrypt or decrypt; use - for
                      stdout
  --encrypt-vault-id ENCRYPT_VAULT_ID
                        the vault id used to encrypt (required if more than
                        one vault-id is provided)

jfreeman@mastery1:~/examples/Chapter03$
```

Figure 3.7 – The options available for the Ansible Vault encrypt subcommand

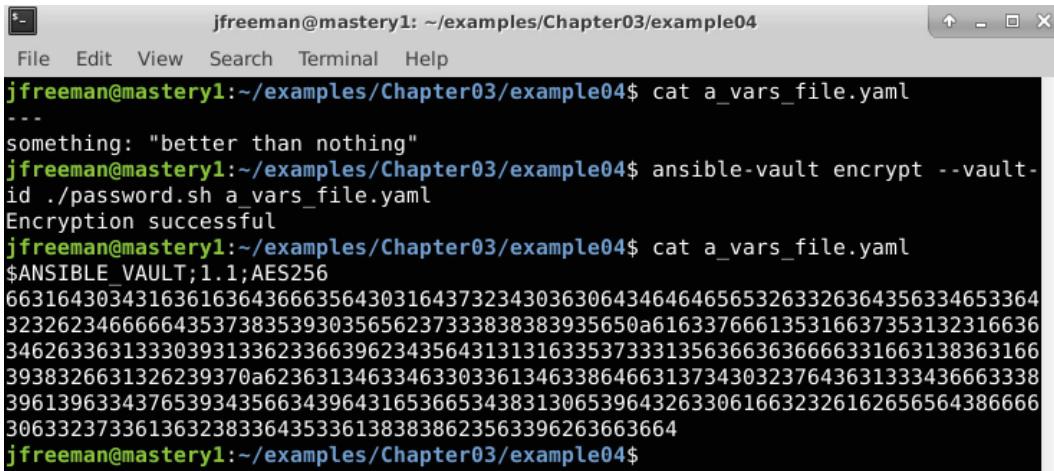
As with `create`, `encrypt` expects a password (or password file or executable) and the path to a file to be encrypted. Once the appropriate password is received, an editor opens up, this time with our original content in plaintext already visible to us.

Note that the file to be encrypted must already exist.

Let's demonstrate this by encrypting an existing file we have from *Chapter 1, The System Architecture and Design of Ansible*, called `Chapter01/example09/a_vars_file.yaml`. Copy this file to a convenient location and then encrypt it with the following command:

```
ansible-vault encrypt --vault-id ./password.sh a_vars_file.yaml
```

The output of this process should look something like that shown in the following screenshot:



A terminal window titled "jfreeman@mastery1: ~/examples/Chapter03/example04". The window shows the following command sequence:

```
jfreeman@mastery1:~/examples/Chapter03/example04$ cat a_vars_file.yaml
---
something: "better than nothing"
jfreeman@mastery1:~/examples/Chapter03/example04$ ansible-vault encrypt --vault-id ./password.sh a_vars_file.yaml
Encryption successful
jfreeman@mastery1:~/examples/Chapter03/example04$ cat a_vars_file.yaml
$ANSIBLE_VAULT;1.1;AES256
6631643034316361636436663564303164373234303630643464646565326332636435633465364
32326234666664353738353930356562373338383935650a616337666135316637353132316636
34626336313330393133623366396234356431313163353733313563663636666331663138363166
3938326631326239370a623631346334633033613463386466313734303237643631333436663338
39613963343765393435663439643165366534383130653964326330616632326162656564386666
3063323733613632383364353361383838623563396263663664
jfreeman@mastery1:~/examples/Chapter03/example04$
```

Figure 3.8 – Encrypting an existing variables file with Ansible Vault

In this example, we can see the file content before and after the call to `encrypt`, whereafter the content are indeed encrypted. Unlike the `create` subcommand, `encrypt` can operate on multiple files, making it easy to protect all the important data in one action. Simply list all the files to be encrypted, separated by spaces.

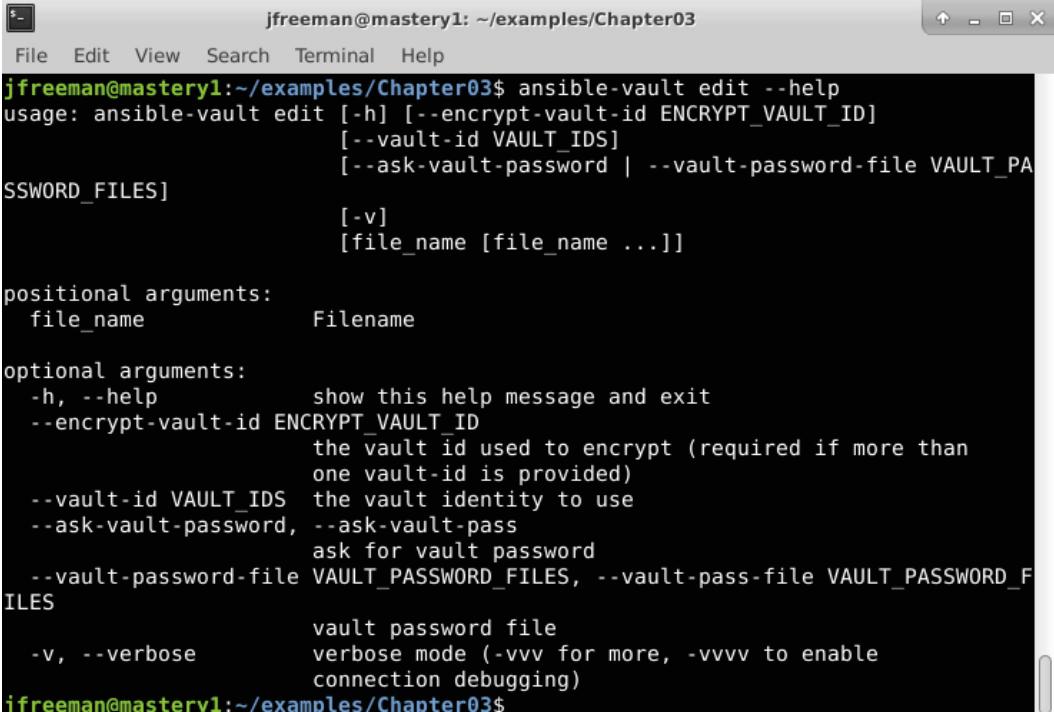
Attempting to encrypt already encrypted files will result in an error.

## Editing encrypted files

Once a file has been encrypted with `ansible-vault`, it cannot be directly edited. Opening the file in an editor would result in the encrypted data being shown. Making any changes to the file would damage the file, and Ansible would be unable to read the content correctly. We need a subcommand that will first decrypt the content of a file, allow us to edit those content, and then encrypt the new content before saving it back to the file. Such a subcommand exists in `edit`, and you can see the options available to you for this subcommand by running the following command:

```
ansible-vault edit --help
```

The output should look similar to that shown in the following screenshot:



A terminal window titled "jfreeman@mastery1: ~/examples/Chapter03". The window shows the help output for the `ansible-vault edit` command. The output includes usage information, positional arguments (file\_name), optional arguments (-h, --help, --encrypt-vault-id, --vault-id, --ask-vault-password, --vault-password-file, -v, --verbose), and detailed descriptions for each option.

```
jfreeman@mastery1:~/examples/Chapter03$ ansible-vault edit --help
usage: ansible-vault edit [-h] [--encrypt-vault-id ENCRYPT_VAULT_ID]
                           [--vault-id VAULT_IDS]
                           [--ask-vault-password | --vault-password-file VAULT_PA
SSWORD_FILES]
                           [-v]
                           [file_name [file_name ...]]

positional arguments:
  file_name           Filename

optional arguments:
  -h, --help          show this help message and exit
  --encrypt-vault-id ENCRYPT_VAULT_ID
                      the vault id used to encrypt (required if more than
                      one vault-id is provided)
  --vault-id VAULT_IDS  the vault identity to use
  --ask-vault-password, --ask-vault-pass
                      ask for vault password
  --vault-password-file VAULT_PASSWORD_FILES, --vault-pass-file VAULT_PASSWORD_F
ILES
                      vault password file
  -v, --verbose        verbose mode (-vvv for more, -vvvv to enable
                      connection debugging)
jfreeman@mastery1:~/examples/Chapter03$
```

Figure 3.9 – The options available for the `edit` subcommand of Ansible Vault

As we've already seen, our editor will open with our content in plaintext visible to us. All of our familiar `vault-id` options are back, as before, as well as the file(s) to edit. As such, we can now edit the file we just encrypted using the following command:

```
ansible-vault edit --vault-id ./password.sh a_vars_file.yaml
```

Notice that `ansible-vault` opens our editor with a temporary file as the file path. When you save and exit the editor, the temporary file gets written, and then `ansible-vault` will encrypt it and move it to replace the original file. The following screenshot shows the unencrypted content of our previously encrypted vault available for editing:

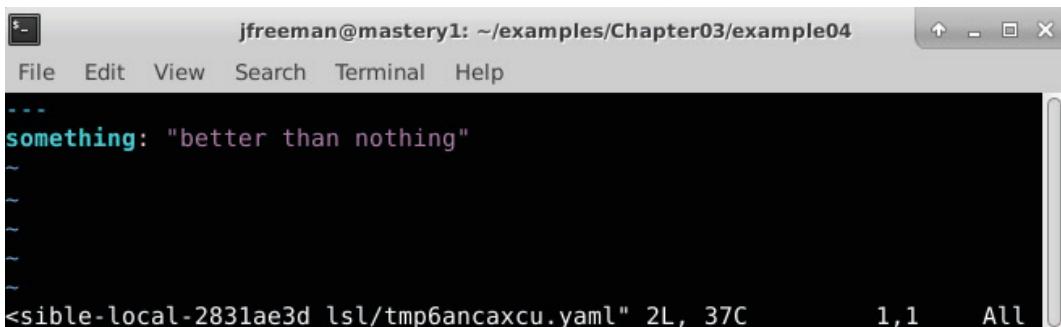
A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter03/example04". The window has a standard OS X-style title bar with icons for close, minimize, and maximize. Below the title bar is a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The main terminal area displays a YAML configuration file. The content includes a single key-value pair: "something: "better than nothing"". There are several blank lines above and below this entry. At the bottom of the terminal window, there is status information: "<visible-local-2831ae3d lsl/tmp6ancaxcu.yaml" 2L, 37C" on the left, and "1,1 All" on the right.

Figure 3.10 – Editing our previously encrypted Ansible Vault

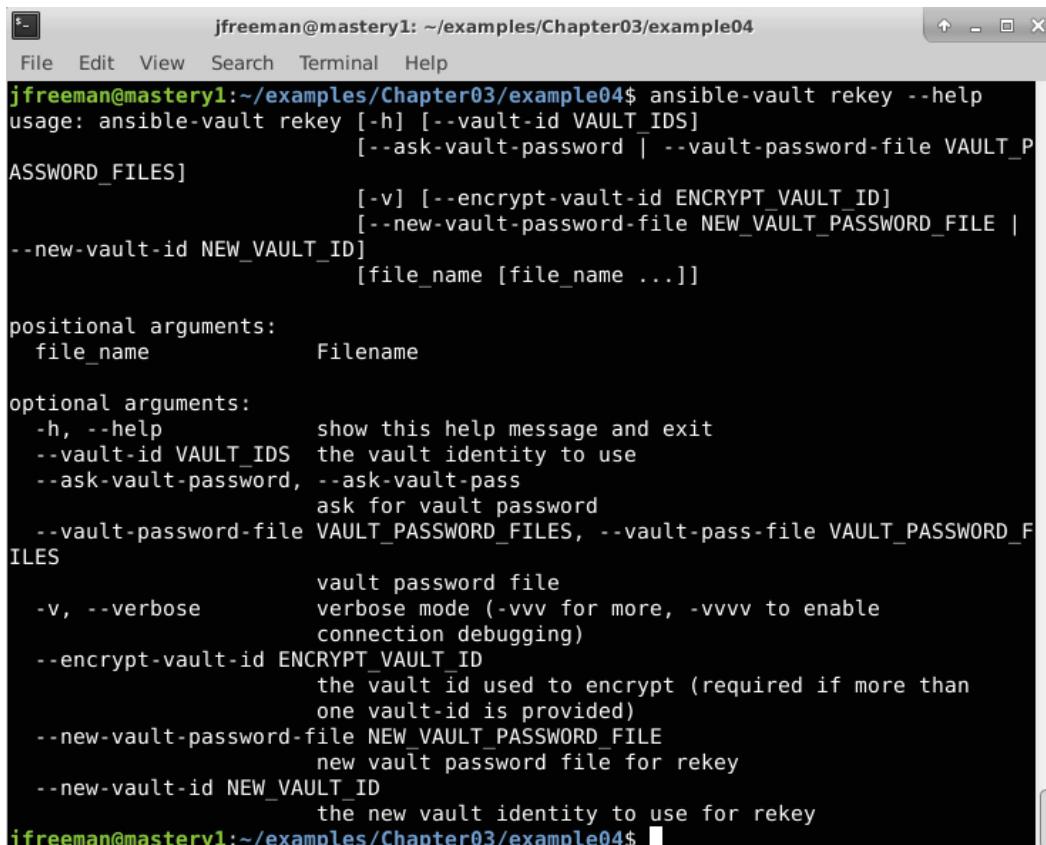
The temporary file you can see in the editor window (`.../tmp6ancaxcu.yaml`) will be removed once the file is successfully encrypted by `ansible-vault`.

## Password rotation on encrypted files

Over time, as contributors come and go, it is a good idea to rotate the password used to encrypt your secrets. Encryption is only as good as the protection of the password. `ansible-vault` provides a `rekey` subcommand that allows us to change the password, and you can explore the options available with this subcommand by running the following command:

```
ansible-vault rekey --help
```

The output should look similar to that shown in the following screenshot:



```
jfreeman@mastery1: ~/examples/Chapter03/example04
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter03/example04$ ansible-vault rekey --help
usage: ansible-vault rekey [-h] [--vault-id VAULT_IDS]
                           [--ask-vault-password | --vault-password-file VAULT_P
ASSWORD_FILES]
                           [-v] [--encrypt-vault-id ENCRYPT_VAULT_ID]
                           [--new-vault-password-file NEW_VAULT_PASSWORD_FILE | --new-vault-id NEW_VAULT_ID]
                           [file_name [file_name ...]]

positional arguments:
  file_name           Filename

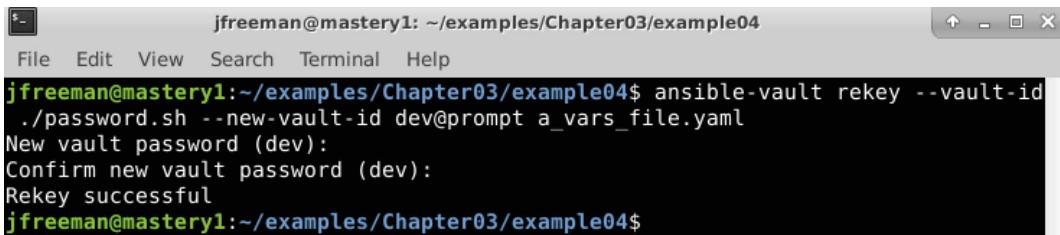
optional arguments:
  -h, --help          show this help message and exit
  --vault-id VAULT_IDS  the vault identity to use
  --ask-vault-password, --ask-vault-pass
                        ask for vault password
  --vault-password-file VAULT_PASSWORD_FILES, --vault-pass-file VAULT_PASSWORD_F
ILES
                        vault password file
  -v, --verbose        verbose mode (-vvv for more, -vvvv to enable
                        connection debugging)
  --encrypt-vault-id ENCRYPT_VAULT_ID
                        the vault id used to encrypt (required if more than
                        one vault-id is provided)
  --new-vault-password-file NEW_VAULT_PASSWORD_FILE
                        new vault password file for rekey
  --new-vault-id NEW_VAULT_ID
                        the new vault identity to use for rekey
jfreeman@mastery1:~/examples/Chapter03/example04$
```

Figure 3.11 – The options available with the Ansible Vault rekey subcommand

The `rekey` subcommand operates much like the `edit` subcommand. It takes in an optional password, file, or executable, and one or more files to rekey. You then need to use the `--new-vault-id` parameter to define a new password (and ID if required), which again can be through a prompt, file, or executable. Let's rekey our `a_vars_file.yaml` file in the following example, and change the ID to dev by running the following command—for now, we'll prompt for the new password, though we know we can obtain the original password using our password script:

```
ansible-vault rekey --vault-id ./password.sh --new-vault-id
dev@prompt a_vars_file.yaml
```

The output should look like that shown in the following screenshot:



```
jfreeman@mastery1: ~/examples/Chapter03/example04
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter03/example04$ ansible-vault rekey --vault-id
./password.sh --new-vault-id dev@prompt a_vars_file.yaml
New vault password (dev):
Confirm new vault password (dev):
Rekey successful
jfreeman@mastery1:~/examples/Chapter03/example04$
```

Figure 3.12 – Rekeying an existing Ansible Vault and changing the ID at the same time

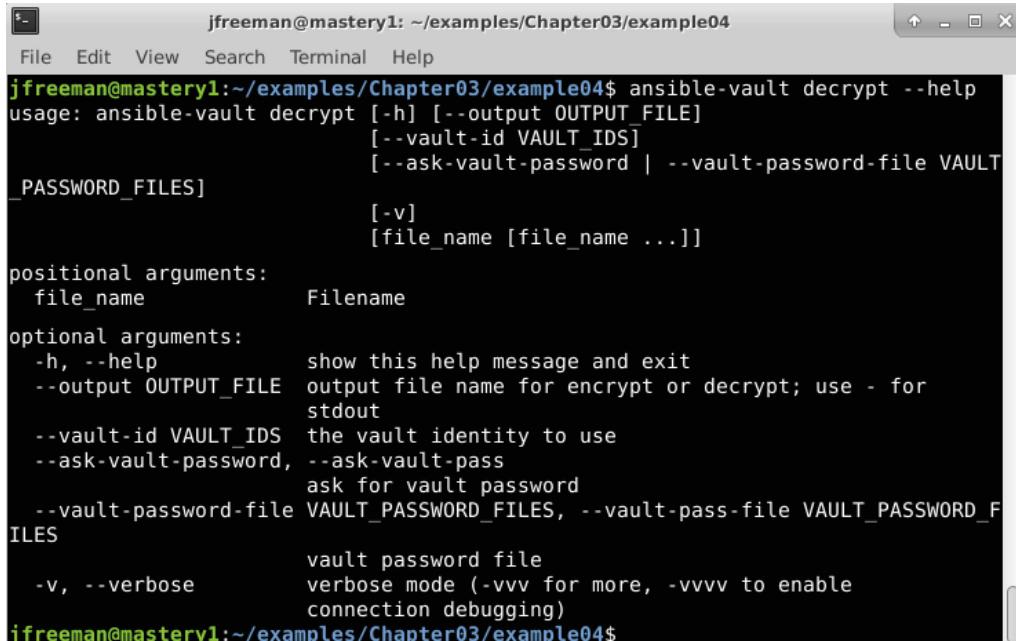
Remember that all the encrypted files **with the same ID** need to have a matching password (or key). Be sure to rekey all the files with the same ID at the same time.

## Decrypting encrypted files

If at some point, the need to encrypt data files goes away, `ansible-vault` provides a subcommand that can be used to remove encryption for one or more encrypted files. This subcommand is (surprisingly) named `decrypt`, and you can view the options for this subcommand by running the following command:

```
ansible-vault decrypt --help
```

The output should look similar to that shown in the following screenshot:



```
jfreeman@mastery1: ~/examples/Chapter03/example04
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter03/example04$ ansible-vault decrypt --help
usage: ansible-vault decrypt [-h] [--output OUTPUT_FILE]
                               [--vault-id VAULT_IDS]
                               [--ask-vault-password | --vault-password-file VAULT
                               _PASSWORD_FILES]
                               [-v]
                               [file_name [file_name ...]]

positional arguments:
  file_name           Filename

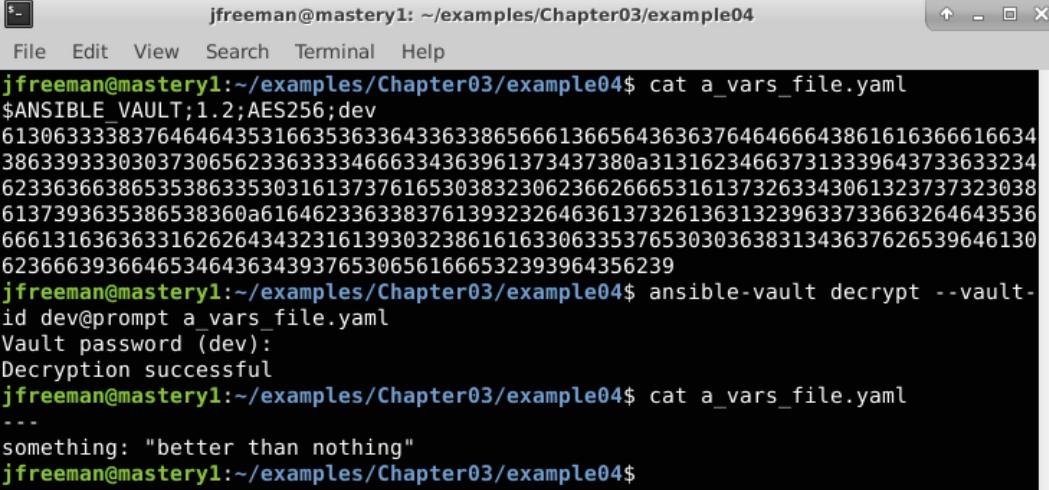
optional arguments:
  -h, --help          show this help message and exit
  --output OUTPUT_FILE output file name for encrypt or decrypt; use - for
                        stdout
  --vault-id VAULT_IDS the vault identity to use
  --ask-vault-password, --ask-vault-pass
                        ask for vault password
  --vault-password-file VAULT_PASSWORD_FILES, --vault-pass-file VAULT_PASSWORD_F
ILES
  -v, --verbose       vault password file
                      verbose mode (-vvv for more, -vvvv to enable
                      connection debugging)
jfreeman@mastery1:~/examples/Chapter03/example04$
```

Figure 3.13 – The options available using the `decrypt` subcommand of Ansible Vault

Once again, we have our familiar `--vault-id` options, and then one or more file paths to decrypt. Let's decrypt the file we rekeyed just now, by running the following command:

```
ansible-vault decrypt --vault-id dev@prompt a_vars_file.yaml
```

If successful, your decryption process should look something like that shown in the following screenshot:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter03/example04". The terminal content is as follows:

```
jfreeman@mastery1:~/examples/Chapter03/example04$ cat a_vars_file.yaml
$ANSIBLE_VAULT;1.2;AES256;dev
61306333383764646435316635363364336338656661366564363637646466643861616366616634
386339333030373065623363334666334363961373437380a313162346637313339643733633234
62336366386535386335303161373761653038323062366266653161373263343061323737323038
6137393635386538360a616462336338376139323264636137326136313239633733663264643536
666131636363316262643432316139303238616163306335376530363831343637626539646130
6236663936646534643634393765306561666532393964356239
jfreeman@mastery1:~/examples/Chapter03/example04$ ansible-vault decrypt --vault-
id dev@prompt a_vars_file.yaml
Vault password (dev):
Decryption successful
jfreeman@mastery1:~/examples/Chapter03/example04$ cat a_vars_file.yaml
---
something: "better than nothing"
jfreeman@mastery1:~/examples/Chapter03/example04$
```

Figure 3.14 – Decrypting an existing vault

In the next section, we will see how to execute `ansible-playbook` when referencing encrypted files.

## Executing `ansible-playbook` with encrypted files

To make use of our encrypted content, we first need to be able to inform `ansible-playbook` of how to access any encrypted data it might encounter. Unlike `ansible-vault`, which exists solely to deal with file encryption or decryption, `ansible-playbook` is more general-purpose, and it will not assume it is dealing with encrypted data by default. Fortunately, all of our familiar `--vault-id` parameters from the previous examples work just the same in `ansible-playbook` as they do in `ansible-vault`. Ansible will hold the provided passwords and IDs in memory for the duration of the playbook execution.

Let's now create a simple playbook named `show_me.yaml` that will print out the value of the variable inside of `a_vars_file.yaml`, which we encrypted in a previous example, as follows:

```
---
- name: show me an encrypted var
  hosts: localhost
  gather_facts: false

  vars_files:
    - a_vars_file.yaml

  tasks:
    - name: print the variable
      ansible.builtin.debug:
        var: something
```

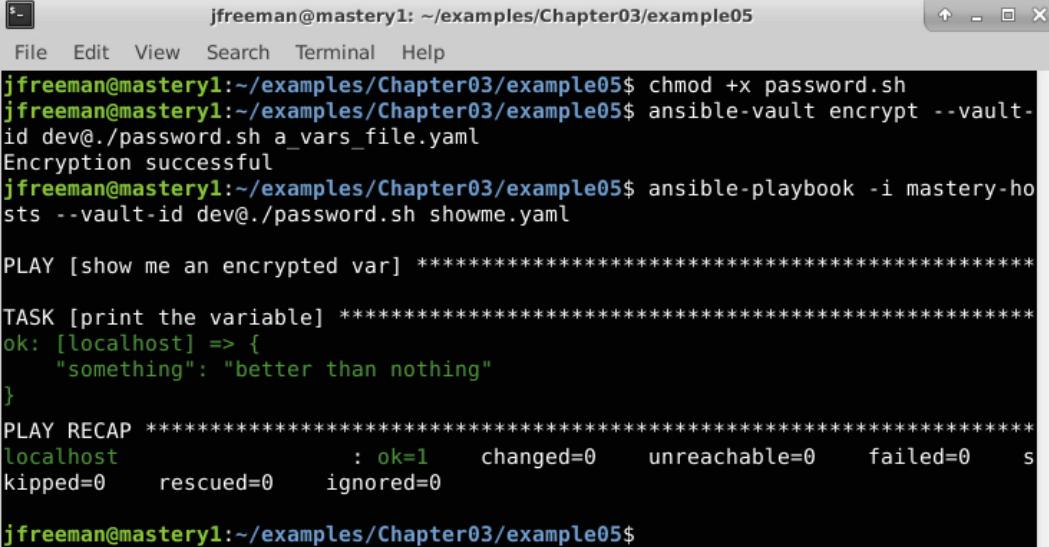
Now, let's run the playbook and see what happens. Note how we use the `--vault-id` parameter in exactly the same way as we did with `ansible-vault`; continuity is maintained between the two tools, so you are able to apply everything you learned earlier in the chapter about using `--vault-id`. If you didn't already complete this step earlier, encrypt your variables file with the following command:

```
chmod +x password.sh
ansible-vault encrypt --vault-id dev@./password.sh a_vars_file.yaml
```

With this done, now run the playbook with the following command—note the presence of the `--vault-id` parameter, similar to before:

```
ansible-playbook -i mastery-hosts --vault-id dev@./password.sh
showme.yaml
```

When you have completed this, your output should look something like that shown in the following screenshot:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter03/example05". The terminal displays the following command-line session:

```
jfreeman@mastery1:~/examples/Chapter03/example05$ chmod +x password.sh
jfreeman@mastery1:~/examples/Chapter03/example05$ ansible-vault encrypt --vault-id dev@./password.sh a_vars_file.yaml
Encryption successful
jfreeman@mastery1:~/examples/Chapter03/example05$ ansible-playbook -i mastery-hosts --vault-id dev@./password.sh showme.yaml

PLAY [show me an encrypted var] ****
TASK [print the variable] ****
ok: [localhost] => {
    "something": "better than nothing"
}
PLAY RECAP ****
localhost                  : ok=1      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0

jfreeman@mastery1:~/examples/Chapter03/example05$
```

Figure 3.15 – Running a simple playbook including an encrypted Ansible Vault instance

As you can see, the playbook runs successfully and prints out the unencrypted value of the variable, even though the source variable file we included was an encrypted Ansible Vault instance. Naturally, you wouldn't print a secret value to the terminal in a real playbook run, but this demonstrates how easy it is to access data from a vault.

In all our examples so far, we have created vaults as external entities—files that live outside of the playbooks themselves. However, it is possible to add encrypted vault data to an otherwise unencrypted playbook, which reduces the number of files we need to track and edit. Let's have a look at how this is achieved in the next section.

## Mixing encrypted data with plain YAML

Before the release of Ansible 2.3, secure data had to be encrypted in a separate file. For the reasons we discussed earlier, it is desirable to encrypt as little data as possible. This is now possible (and also saves a need for too many individual files as part of a playbook) through the use of the `encrypt_string` subcommand of `ansible-vault`, which produces an encrypted string that can be placed into an Ansible YAML file. Let's start with the following basic playbook as an example:

```
---
- name: inline secret variable demonstration
```

```

hosts: localhost
gather_facts: false

vars:
    my_secret: secure_password

tasks:
    - name: print the secure variable
      ansible.builtin.debug:
        var: my_secret

```

We can run this code (insecure though it is!) with the following command:

```
ansible-playbook -i mastery-hosts inline.yaml
```

When this playbook runs, the output should look similar to that shown in the following screenshot:

```
jfreeman@mastery1:~/examples/Chapter03/example06$ ansible-playbook -i mastery-hosts inline.yaml

PLAY [inline secret variable demonstration] ****
TASK [print the secure variable] ****
ok: [localhost] => {
    "my_secret": "secure_password"
}

PLAY RECAP ****
localhost : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0   rescued=0   ignored=0

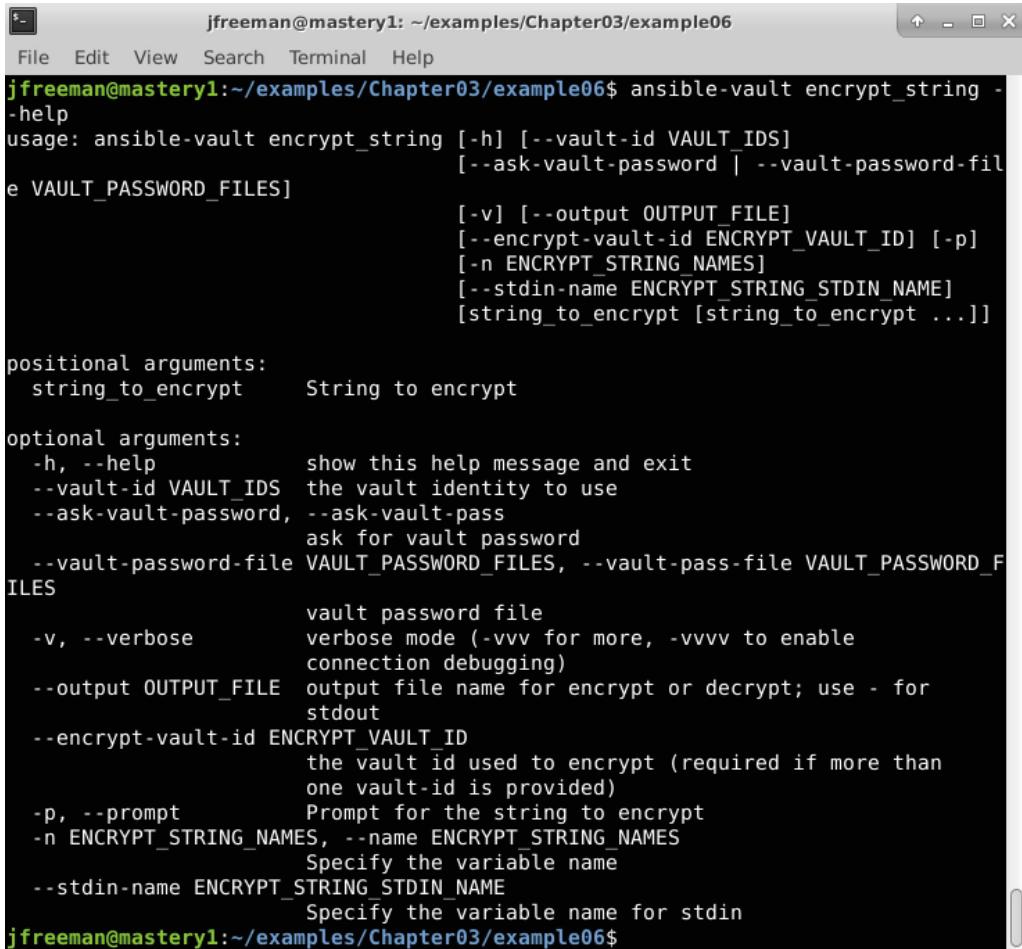
jfreeman@mastery1:~/examples/Chapter03/example06$
```

Figure 3.16 – Running an unencrypted playbook containing sensitive data

Now, it obviously isn't clever to leave a secure password in plaintext like this. So, rather than leave it like this, we will encrypt it using the `encrypt_string` subcommand of `ansible-vault`. If you want to see the options available to you when running this subcommand, you can execute the following command:

```
ansible-vault encrypt_string --help
```

The output of this command should look similar to that shown in the following screenshot:



```
jfreeman@mastery1:~/examples/Chapter03/example06
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter03/example06$ ansible-vault encrypt_string -help
usage: ansible-vault encrypt_string [-h] [--vault-id VAULT_IDS]
                                     [--ask-vault-password | --vault-password-file VAULT_PASSWORD_FILES]
                                     [-v] [--output OUTPUT_FILE]
                                     [--encrypt-vault-id ENCRYPT_VAULT_ID] [-p]
                                     [-n ENCRYPT_STRING_NAMES]
                                     [--stdin-name ENCRYPT_STRING_STDIN_NAME]
                                     [string_to_encrypt [string_to_encrypt ...]]

positional arguments:
  string_to_encrypt      String to encrypt

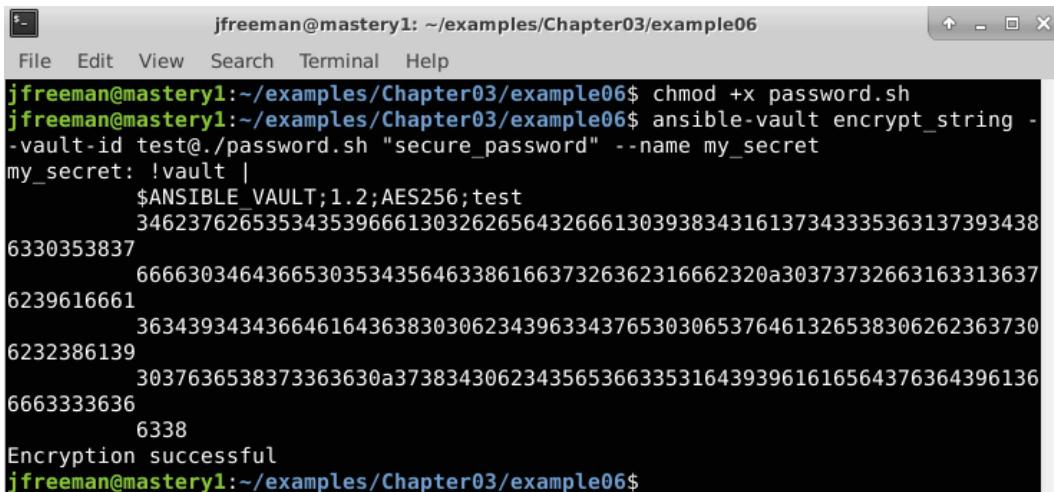
optional arguments:
  -h, --help            show this help message and exit
  --vault-id VAULT_IDS  the vault identity to use
  --ask-vault-password, --ask-vault-pass
                        ask for vault password
  --vault-password-file VAULT_PASSWORD_FILES, --vault-pass-file VAULT_PASSWORD_FILES
                        vault password file
  -v, --verbose         verbose mode (-vvv for more, -vvvv to enable
                        connection debugging)
  --output OUTPUT_FILE   output file name for encrypt or decrypt; use - for
                        stdout
  --encrypt-vault-id ENCRYPT_VAULT_ID
                        the vault id used to encrypt (required if more than
                        one vault-id is provided)
  -p, --prompt          Prompt for the string to encrypt
  -n ENCRYPT_STRING_NAMES, --name ENCRYPT_STRING_NAMES
                        Specify the variable name
  --stdin-name ENCRYPT_STRING_STDIN_NAME
                        Specify the variable name for stdin
jfreeman@mastery1:~/examples/Chapter03/example06$
```

Figure 3.17 – The options available for the encrypt\_string subcommand of Ansible Vault

So, if we wanted to create an encrypted block of text for our `my_secret` variable with the `secure_password` encrypted string, using the `test` Vault ID and the `password.sh` script we created earlier for the password, we would run the following commands:

```
chmod +x password.sh
ansible-vault encrypt_string --vault-id test@./password.sh
"secure_password" --name my_secret
```

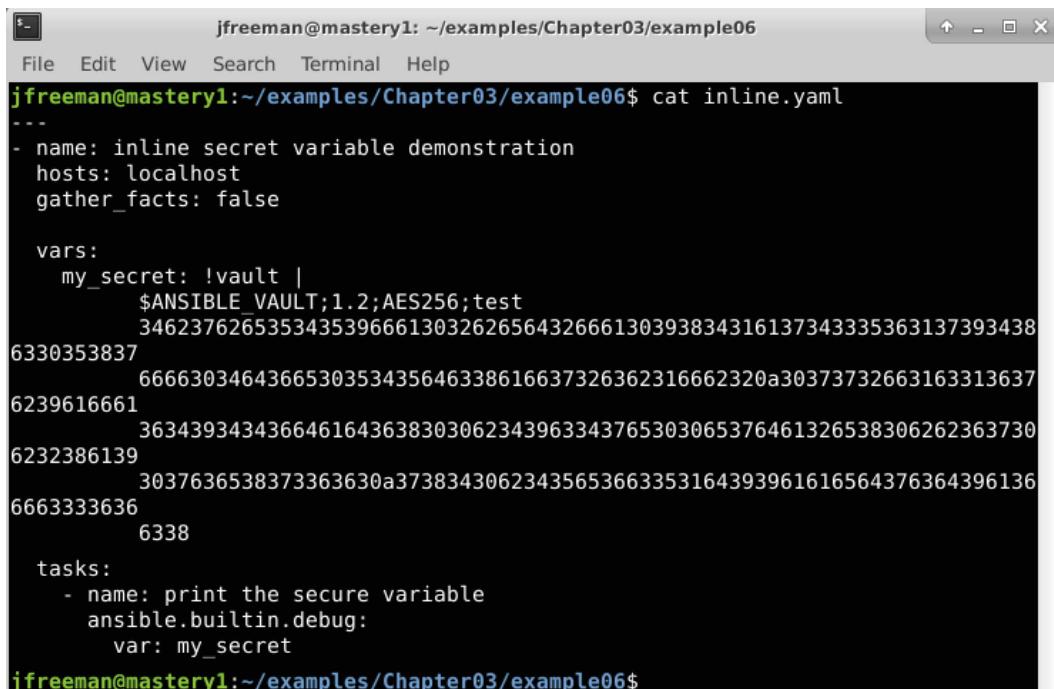
The output of these commands will give you the encrypted string to include in your existing playbook, and an example is shown in the following screenshot:



```
jfreeman@mastery1: ~/examples/Chapter03/example06$ chmod +x password.sh
jfreeman@mastery1:~/examples/Chapter03/example06$ ansible-vault encrypt_string -
-vault-id test@./password.sh "secure_password" --name my_secret
my_secret: !vault |
    $ANSIBLE_VAULT;1.2;AES256;test
    3462376265353435396661303262656432666130393834316137343335363137393438
6330353837
    6666303464366530353435646338616637326362316662320a30373732663163313637
6239616661
    3634393434366461643638303062343963343765303065376461326538306262363730
6232386139
    3037636538373363630a37383430623435653663353164393961616564376364396136
6663333636
    6338
Encryption successful
jfreeman@mastery1:~/examples/Chapter03/example06$
```

Figure 3.18 – Encrypting a variable to a secure string using Ansible Vault

We can now copy and paste that output into our playbook, ensuring our variable is no longer human-readable, as demonstrated in the following screenshot:



```
jfreeman@mastery1: ~/examples/Chapter03/example06$ cat inline.yaml
---
- name: inline secret variable demonstration
  hosts: localhost
  gather_facts: false

  vars:
    my_secret: !vault |
        $ANSIBLE_VAULT;1.2;AES256;test
        3462376265353435396661303262656432666130393834316137343335363137393438
6330353837
        6666303464366530353435646338616637326362316662320a30373732663163313637
6239616661
        3634393434366461643638303062343963343765303065376461326538306262363730
6232386139
        3037636538373363630a37383430623435653663353164393961616564376364396136
6663333636
        6338

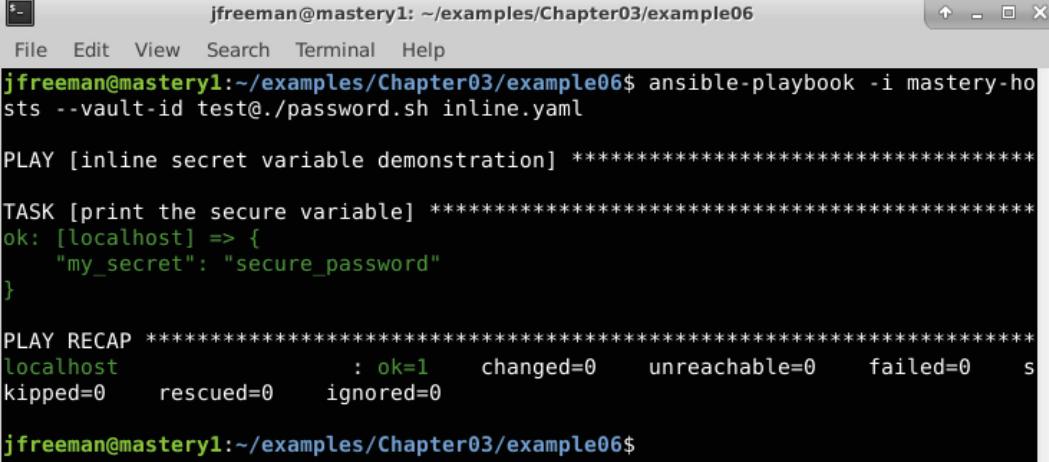
  tasks:
    - name: print the secure variable
      ansible.builtin.debug:
        var: my_secret
jfreeman@mastery1:~/examples/Chapter03/example06$
```

Figure 3.19 – Replacing the unencrypted variable with the encrypted string data in our existing playbook

Even though we have now embedded an Ansible Vault encrypted variable directly inside our playbook, we can run this playbook using the appropriate `--vault-id` just like we did before—the following command will be used here:

```
ansible-playbook -i mastery-hosts --vault-id test@./password.sh
inline.yaml
```

You will observe the playbook running and see that the information can be accessed just as any other vault data can, and your output should look like that shown in the following screenshot:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter03/example06". The window includes standard menu options: File, Edit, View, Search, Terminal, and Help. The terminal content is as follows:

```
jfreeman@mastery1:~/examples/Chapter03/example06$ ansible-playbook -i mastery-hosts --vault-id test@./password.sh inline.yaml

PLAY [inline secret variable demonstration] *****

TASK [print the secure variable] *****
ok: [localhost] => {
    "my_secret": "secure_password"
}

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

jfreeman@mastery1:~/examples/Chapter03/example06$
```

Figure 3.20 – Running an Ansible playbook containing an encrypted string

You can see that the playbook runs exactly as it did the first time we tested it when all the data was open for the world to see! Now, however, we have successfully mixed our encrypted data with an otherwise unencrypted YAML playbook, all without a need to create a separate Vault file.

In the next section, we will delve deeper into some of the operational aspects of running playbooks in conjunction with Ansible Vault.

## Protecting secrets while operating

In the previous section of this chapter, we covered how to protect your secrets at rest on the filesystem. However, that is not the only concern when operating Ansible with secrets. That secret data is going to be used in tasks as module arguments, loop inputs, or any number of other things. This may cause the data to be transmitted to remote hosts, logged to local or remote log files, or even displayed onscreen. This section of the chapter will discuss strategies for protecting your secrets during operation.

## Secrets transmitted to remote hosts

As we learned in *Chapter 1, The System Architecture and Design of Ansible*, Ansible combines module code and arguments and writes this out to a temporary directory on the remote host. This means your secret data is transferred over the wire *and* written to the remote filesystem. Unless you are using a connection plugin other than **Secure Shell (SSH)** or **Secure Sockets Layer (SSL)-encrypted Windows Remote Management (WinRM)**, the data over the wire is already encrypted, preventing your secrets from being discovered by simple snooping. If you are using a connection plugin other than SSH, be aware of whether or not data is encrypted while in transit. Using any connection method that is not encrypted is strongly discouraged.

Once the data is transmitted, Ansible may write this data out in clear form to the filesystem. This can happen if pipelining (which we learned about in *Chapter 1, The System Architecture and Design of Ansible*) is not in use, or if Ansible has been instructed to leave remote files in place via the `ANSIBLE_KEEP_REMOTE_FILES` environment variable. Without pipelining, Ansible will write out the module code, plus arguments, into a temporary directory that is to be deleted immediately after execution. Should there be a loss of connectivity between writing out the file and executing it, the file will be left on the remote filesystem until manually removed. If Ansible is explicitly instructed to keep remote files in place, then even if pipelining is enabled, Ansible will write and leave a remote file in place. Care should be taken with these options when dealing with highly sensitive secrets, even though typically, only the user Ansible authenticates with on the remote host (or becomes via privilege escalation) should have access to the leftover file. Simply deleting anything in the `~/.ansible/tmp/` path for the remote user will suffice to clean secrets.

## Secrets logged to remote or local files

When Ansible operates on a host, it will attempt to log the action to `syslog` (if verbosity level 3 or more is used). If this action is being done by a user with appropriate rights, it will cause a message to appear in the `syslog` file of the host. This message includes the module name and the arguments passed along to that command, which could include your secrets. To prevent this from happening, a play-and-task key exists, called `no_log`. Setting `no_log` to `true` will prevent Ansible from logging the action to `syslog`.

Ansible can also be instructed to log its actions locally. This is controlled either through `log_path` in the Ansible config file or through an environment variable called `ANSIBLE_LOG_PATH`. By default, logging is off and Ansible will only log to `STDOUT`. Turning logging on in the config file causes Ansible to log its activities to the file defined in the `logpath` config setting.

Alternatively, setting the `ANSIBLE_LOG_PATH` variable to a path that can be written to by the user running `ansible-playbook` will also cause Ansible to log actions to this path. The verbosity of this logging matches that of the verbosity shown onscreen. By default, no variables or return details are displayed onscreen. With a verbosity level of 1 (`-v`), return data is displayed onscreen (and potentially in the local log file). With verbosity turned up to level 3 (`-vvv`), the input parameters may also be displayed. Since this can include secrets, the `no_log` setting applies to the onscreen display as well. Let's take our previous example of displaying an encrypted secret and add a `no_log` key to the task to prevent showing its value, as follows:

```
---
```

```
- name: show me an encrypted var
  hosts: localhost
  gather_facts: false

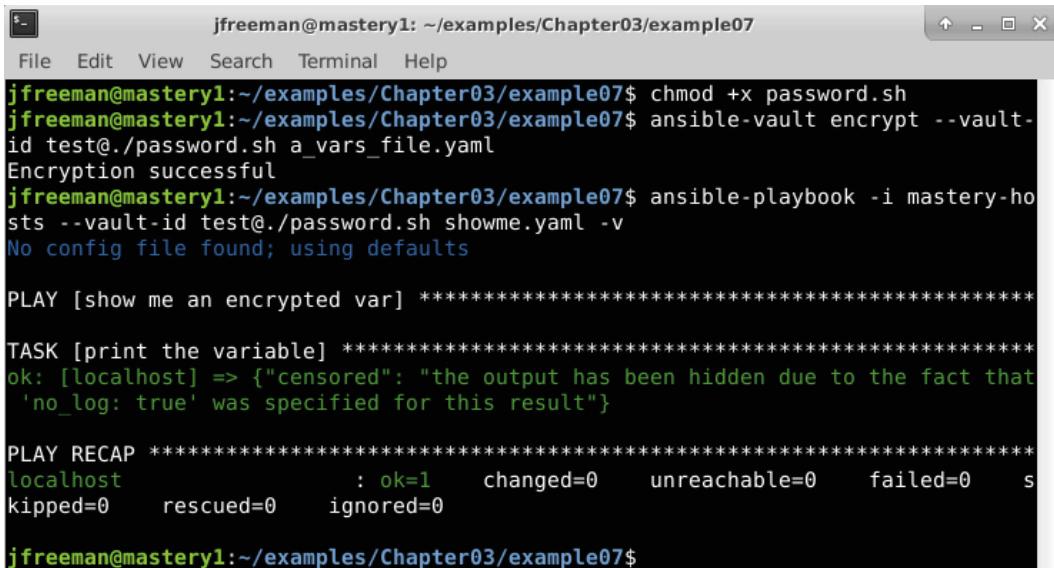
  vars_files:
    - a_vars_file.yaml

  tasks:
    - name: print the variable
      ansible.builtin.debug:
        var: something
        no_log: true
```

We will execute this playbook in the same manner as we have before (but with added verbosity, as specified with the `-v` flag) by running the following command—remember to encrypt the variables file first if you need to:

```
ansible-playbook -i mastery-hosts --vault-id test@./password.
sh showme.yaml -v
```

We should see that our secret data is protected, even though we deliberately attempted to print it using `ansible.builtin.debug`, as shown in the following screenshot:



```
jfreeman@mastery1: ~/examples/Chapter03/example07
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter03/example07$ chmod +x password.sh
jfreeman@mastery1:~/examples/Chapter03/example07$ ansible-vault encrypt --vault-id test@./password.sh a_vars_file.yaml
Encryption successful
jfreeman@mastery1:~/examples/Chapter03/example07$ ansible-playbook -i mastery-ho
sts --vault-id test@./password.sh showme.yaml -v
No config file found; using defaults

PLAY [show me an encrypted var] ****
TASK [print the variable] ****
ok: [localhost] => {"censored": "the output has been hidden due to the fact that
'no_log: true' was specified for this result"}

PLAY RECAP ****
localhost : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0   rescued=0   ignored=0

jfreeman@mastery1:~/examples/Chapter03/example07$
```

Figure 3.21 – Encrypting a variables file and running a playbook with the sensitive data protected  
As you can see, Ansible censored itself to prevent showing sensitive data. The `no_log` key can be used as a directive for a play, a role, a block, or a task.

That concludes our look at operational usage of Ansible Vault, and indeed the topic of Ansible Vault—it is hoped that this chapter has proved useful in teaching you how to secure your sensitive data when performing automation with Ansible.

## Summary

In this chapter, we covered how Ansible can deal with sensitive data effectively and securely, harnessing the latest Ansible features, including securing differing data with different passwords and mixing encrypted data with plain YAML. We have also shown how this data is stored at rest and how this data is treated when utilized, and that with a little care and attention, Ansible can keep your secrets secret.

You learned how to use the `ansible-vault` tool to protect sensitive data by creating, editing, and modifying encrypted files, and the variety of methods available for providing the Vault password, including prompting the user, obtaining the password from a file, and running a script to retrieve it. You also learned how to mix encrypted strings with plain YAML files, and how this simplifies playbook layout. Finally, you learned the operational aspects of using Ansible Vault, thus preventing Ansible from leaking data to remote log files or onscreen displays.

In our next chapter, we will explore how the power of Ansible is now available for Windows hosts, and how to harness this.

## Questions

1. Ansible Vault encrypts your data at rest using which encryption technology?
  - a) Triple DES/3DES
  - b) MD5
  - c) AES
  - d) Twofish
2. Ansible Vault instances must always exist as separate files to the playbook itself:
  - a) True
  - b) False
3. You can ingest data from more than one Ansible Vault instance when running a playbook:
  - a) True
  - b) False
4. When executing a playbook that makes use of Vault-encrypted data, you can provide the password:
  - a) Interactively at playbook launch
  - b) Using a plaintext file containing just the password
  - c) Using a script to retrieve the password from another source
  - d) All of the above
5. Ansible will never print vault data to the terminal during a playbook run:
  - a) True
  - b) False
6. You can prevent Ansible from inadvertently printing vault data to the terminal during a playbook run using the following task parameter:
  - a) no\_print
  - b) no\_vault
  - c) no\_log

7. An interrupted playbook run could leave sensitive unencrypted data on a remote host:
  - a) True
  - b) False
8. What is used to differentiate different vaults (which may have different passwords) at runtime?
  - a) Vault names
  - b) Vault IDs
  - c) Vault specifiers
  - d) None of the above
9. You can edit an existing encrypted vault using which Ansible command?
  - a) `ansible-vault vi`
  - b) `ansible-vault change`
  - c) `ansible-vault update`
  - d) `ansible-vault edit`
10. Why might you not want to mix sensitive and non-sensitive data in a vault?
  - a) Doing so makes it difficult to run `diff` commands and see changes in a **version control system (VCS)**.
  - b) Only sensitive data is allowed in Ansible Vault.
  - c) Ansible Vault has a limited capacity.
  - d) Ansible Vault makes it difficult to access secured data.



# 4

# **Ansible and Windows – Not Just for Linux**

A great deal of the work on Ansible has been performed on Linux OSes; indeed, the first two editions of this book were based entirely around the use of Ansible in a Linux-centric environment. However, most environments are not like that, and, at the very least, are liable to have at least some Microsoft Windows server and desktop machines. Since the third edition of this book was published, much work has gone into Ansible to create a really robust cross-platform automation tool that is equally at home in both a Linux data center and a Windows one. There are fundamental differences in the way Windows and Linux hosts operate, of course, and so it should come as no surprise that there are some fundamental differences between how Ansible automates tasks on Linux, and how it automates tasks on Windows.

We will cover those fundamentals in this chapter, so as to give you a rock-solid foundation to begin automating your Windows tasks with Ansible, specifically covering the following areas:

- Running Ansible from Windows
- Setting up Windows hosts for Ansible control
- Handling Windows authentication and encryption
- Automating Windows tasks with Ansible

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running Ansible 4.3 or newer. Almost any flavor of Linux should do; for those interested in specifics, all the code presented in this chapter was tested on Ubuntu Server 20.04 LTS unless stated otherwise, and on Ansible 4.3.

Where Windows is used in this chapter, the example code was tested and run on Windows Server 2019, version 1809, build 17763.1817. Screenshots of the Windows Store were taken from Windows 10 Pro, version 20H2, build 19042.906.

The example code that accompanies this chapter can be downloaded from GitHub at this URL: <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter04>.

Check out the following video to see the Code in Action: <https://bit.ly/3B2zmvL>.

## Running Ansible from Windows

If you browse the official installation documentation for Ansible, you will find a variety of instructions for most mainstream Linux variants, Solaris, macOS, and FreeBSD. You will note, however, that there is no mention of Windows. There is a good reason for this – for those interested in the technical detail, Ansible makes extensive use of the POSIX `fork()` syscall in its operations, and no such call exists on Windows. POSIX compatibility projects, such as the venerable Cygwin, have attempted to implement `fork()` on Windows, but sometimes this does not work correctly even today. As a result, despite there being a viable Python implementation for Windows, Ansible cannot be run natively on this platform without the presence of this important syscall.

The good news is that, if you are running recent versions of Windows 10, or Windows Server 2016 or 2019, installing and running Ansible is now incredibly easy thanks to **Windows Subsystem for Linux (WSL)**. There are now two versions of this technology, the original WSL release (which was featured in the third edition of this book), and the newer **WSL2**. **WSL2** is, at the time of writing, only available on Windows 10, version 1903 (or higher) with build 18362 (or higher). Both of these technologies allow Windows users to run unmodified Linux distributions on top of Windows without the complications or overheads of a virtual machine (though peer under the hood and you'll see that **WSL2** runs on top of Hyper-V, albeit in a seamless manner). As such, these technologies lend themselves perfectly to running Ansible, as it can be installed and run with ease and with a reliable implementation of the `fork()` syscall.

Let's pause to take a look at two important points before we move on. First of all, WSL or WSL2 are only required to run Ansible from Windows to control other machines (running any OS) – they are not required to control a Windows machine with Ansible. We'll see more about this later in the chapter. Secondly, don't let the lack of an official build of WSL2 for Windows Server impede you – if you have Windows bastion hosts, and wish to run Ansible from them, it is as home on **WSL** as it is on **WSL2**. At the time of writing, there is talk of **WSL2** being available for the latest Insider Previews of Windows Server; however, as I anticipate most readers will be looking for a stable, production-ready solution, we will focus more on **WSL** than **WSL2** in this chapter.

The official Ansible installation documentation can be found at [https://docs.ansible.com/ansible/latest/installation\\_guide/intro\\_installation.html](https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html).

## Checking your build

WSL is only available on specific builds of Windows, as follows:

- Windows 10—version 1607 (build 14393) or later:
  - Note that you will require build 16215 or later if you want to install Linux through the Microsoft Store.
  - If you do want to use WSL2, you will need version 1903 or later (build 18362 or later) of Windows 10.
  - Only 64-bit Intel and ARM versions of Windows 10 are supported.
- Windows Server 2016 version 1803 (build 16215) or later
- Windows Server 2019 version 1709 (build 16237) or later

You can easily check your build and version number in PowerShell by running the following command:

```
systeminfo | Select-String "OS Name", "OS Version"
```

If you are running an earlier version of Windows, running Ansible is still possible, either through a virtual machine or via Cygwin. However, these methods are beyond the scope of this book.

## Enabling WSL

Once you have verified your build, enabling WSL is easy. Simply open PowerShell as an administrator and run the following command:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-  
Windows-Subsystem-Linux
```

Once the installation completes successfully, you will be able to select and install your preferred Linux distribution. A number are available, but for running Ansible, it makes sense to choose one of those listed in the official Ansible installation instructions, such as Debian or Ubuntu.

## Installing Linux under WSL

If you have a recent enough build of Windows 10, then installing your preferred Linux is as easy as opening the Microsoft Store and searching for it. For example, search for Ubuntu and you should find it easily. *Figure 4.1* shows the latest LTS build of Ubuntu available for download in the Microsoft Store on Windows 10:

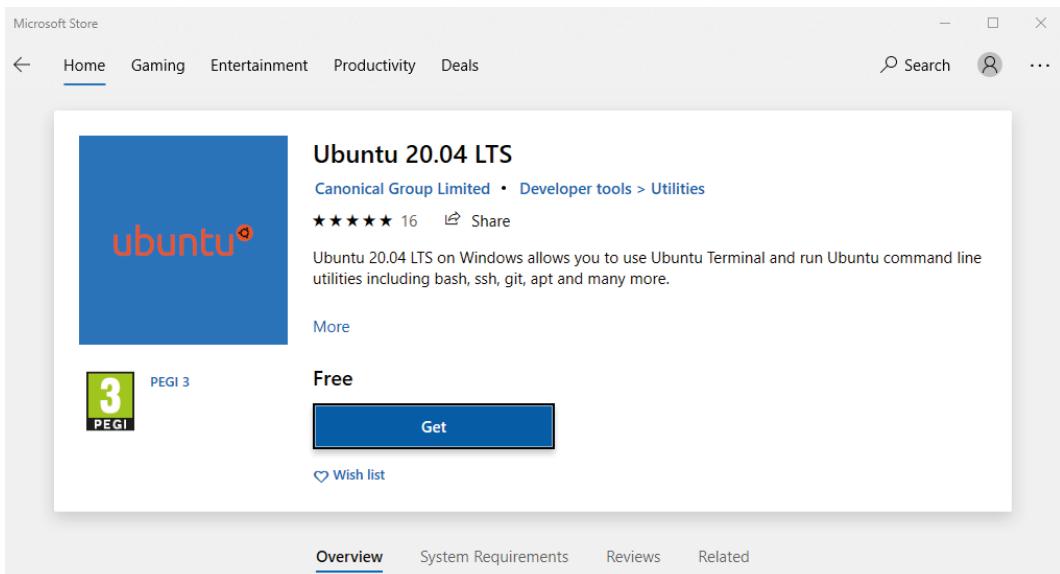


Figure 4.1 – One of the Linux distributions available for WSL and WSL2  
in the Microsoft Store app on Windows 10

To install Ubuntu under WSL, simply click on the **Get** button and wait for the installation to complete.

If you are running Windows 10, but a supported build earlier than 16215, or indeed any supported build of Windows Server 2016/2019, then the installation of Linux is a slightly more manual process. First of all, download your preferred Linux distribution from Microsoft—for example, Ubuntu 20.04 can be downloaded using the following PowerShell command:

```
Invoke-WebRequest -Uri https://aka.ms/wslubuntu2004 -OutFile Ubuntu.appx -UseBasicParsing
```

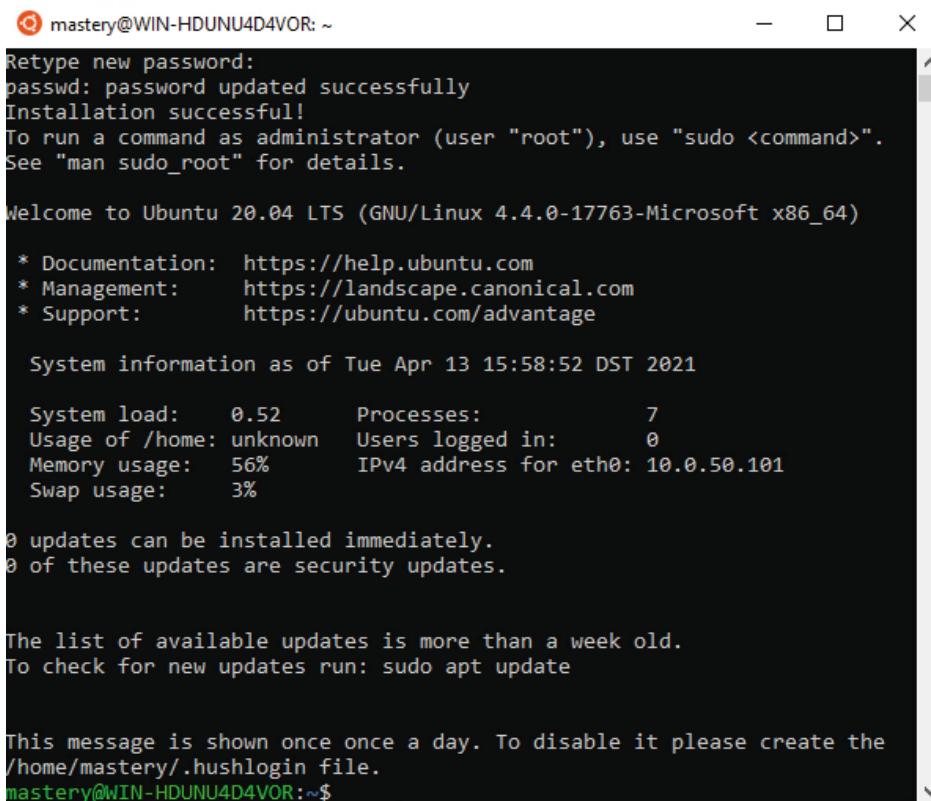
Once successfully downloaded, unzip the Ubuntu.appx file—this can be unzipped to any location provided that it is on the system (boot) drive, normally C:. If you want to keep your Linux distribution private, it can be unzipped somewhere within your profile directory, otherwise, you can unzip the file anywhere on the system drive. For example, the following PowerShell commands would unzip the archive into C:\WSL\:

```
Rename-Item Ubuntu.appx Ubuntu.zip
Expand-Archive Ubuntu.zip C:\WSL\Ubuntu
```

Once completed, you can launch your newly installed Linux distribution using the executable named after the distribution itself. In the case of our Ubuntu example, you would run the following through Explorer (or your preferred method):

```
C:\WSL\Ubuntu\ubuntu2004.exe
```

The first time you run your newly installed Linux distribution, whether it was installed through the Microsoft Store or installed manually, it will initialize itself. As part of this process, it will ask you to create a new user account. Please note that this account is independent of your Windows username and password, so be sure to remember the password you set here! You will need it every time you run commands through `sudo` (for example), although, as with any Linux distribution, you can customize this behavior through `/etc/sudoers` if you wish. This is demonstrated in *Figure 4.2*:



A screenshot of a terminal window titled "mastery@WIN-HDUNU4D4VOR: ~". The window shows the initial boot sequence of Ubuntu 20.04 LTS. It starts with a password prompt, followed by a message indicating the password was updated successfully and the installation was successful. It provides instructions for running commands as root using sudo. The terminal then displays the welcome message for Ubuntu 20.04 LTS, version 4.4.0-17763-Microsoft x86\_64. It lists documentation, management, and support links. A section titled "System information as of Tue Apr 13 15:58:52 DST 2021" provides system load, memory usage, and network information. It then checks for updates, stating there are 0 updates available. Finally, it reminds the user that the update list is old and suggests running sudo apt update to check for new updates.

```
mastery@WIN-HDUNU4D4VOR: ~
Retype new password:
passwd: password updated successfully
Installation successful!
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

Welcome to Ubuntu 20.04 LTS (GNU/Linux 4.4.0-17763-Microsoft x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 System information as of Tue Apr 13 15:58:52 DST 2021

 System load:   0.52      Processes:          7
 Usage of /home: unknown   Users logged in:   0
 Memory usage:  56%       IPv4 address for eth0: 10.0.50.101
 Swap usage:    3%

 0 updates can be installed immediately.
 0 of these updates are security updates.

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

This message is shown once once a day. To disable it please create the
/home/mastery/.hushlogin file.
mastery@WIN-HDUNU4D4VOR:~$
```

Figure 4.2 – The WSL Ubuntu terminal output during its first run

Congratulations! You now have Linux running under WSL. From here, you should follow the standard installation process for Ansible, and you can run it from your Linux subsystem just as you would on any other Linux box.

## Setting up Windows hosts for Ansible control using WinRM

So far, we have talked about running Ansible itself from Windows. This is helpful, especially in a corporate environment where perhaps Windows end user systems are the norm. However, what about actual automation tasks? The good news is that, as already stated, automation of Windows with Ansible does not require WSL. One of Ansible's core premises is to be agentless, and that remains just as true for Windows as for Linux. It is fair to assume that almost any modern Linux host will have SSH access enabled, and similarly, most modern Windows hosts have a remote management protocol built in, called WinRM. Ardent followers of Windows will know that Microsoft has, in a more recent edition, added both the OpenSSH client and server packages, and since the last edition of this book was published, experimental support for these has been added to Ansible. For security reasons, both of these technologies are disabled by default, and so, in this part of the book, we walk through the processes for enabling and securing WinRM for remote management with Ansible. We will also take a brief look at setting up and using OpenSSH Server on Windows – however, as support for this by Ansible is currently experimental and carries a number of warnings about stability and backward-incompatible changes in future releases, most users will wish to use WinRM, especially in stable production environments.

With this in mind, let's get started on looking at automating tasks on Windows hosts using WinRM in the next part of this chapter.

## System requirements for automation with Ansible using WinRM

The use of WinRM by Ansible means a wide array of support for Windows versions new and old—under the hood, just about any Windows version that supports the following will work:

- PowerShell 3.0
- .NET 4.0

In practice, this means that the following Windows versions can be supported, provided the preceding requirements are met:

- **Desktop:** Windows 7 SP1, 8.1, and 10
- **Server:** Windows Server 2008 SP2, 2008 R2 SP1, 2012, 2012 R2, 2016, and 2019

Note that the older OSes listed previously (such as Windows 7 or Server 2008) did not ship with .NET 4.0 or PowerShell 3.0, and these will need to be installed before they can be used with Ansible. As you would expect, newer versions of PowerShell are supported, and, equally, there may be security patches for .NET 4.0. As long as you can meet these minimum requirements, you should be fine to start automating Windows tasks with Ansible, even in business settings where older OSes are still dominant.

If you are using an older (but supported) version of PowerShell such as 3.0, be aware that a bug exists in WinRM under PowerShell 3.0 that limits the memory available to the service, which, in turn, can cause some Ansible commands to fail. This is resolved by ensuring KB2842230 is applied to all hosts running PowerShell 3.0, so do be sure to check your hotfixes and patches if you are automating tasks in Windows via PowerShell 3.0.

## Enabling the WinRM listener

Once all the system requirements have been met, as detailed previously, the task that remains is to enable and secure the WinRM listener. With this achieved, we can actually run Ansible tasks against the Windows host itself! WinRM can run over both HTTP, and HTTPS protocols, and, while it is quickest and easiest to get up and running over plain HTTP, this leaves you vulnerable to packet sniffers and the potential for sensitive data to be revealed on the network. This is especially true if basic authentication is being used. By default, and perhaps unsurprisingly, Windows does not allow remote management with WinRM over HTTP or using basic authentication.

Sometimes, basic authentication is sufficient (for example, in a development environment), and if it is to be used, then we will definitely want to enable HTTPS as the transport for WinRM! However, later in the chapter, we will look at Kerberos authentication, which is preferable, and also enables the use of domain accounts. For now though, to demonstrate the process of connecting Ansible to a Windows host with a modicum of security, we will enable WinRM over HTTPS using a self-signed certificate, and enable basic authentication to allow us to work with the local `Administrator` account.

For WinRM to function over HTTPS, a certificate must exist that has the following:

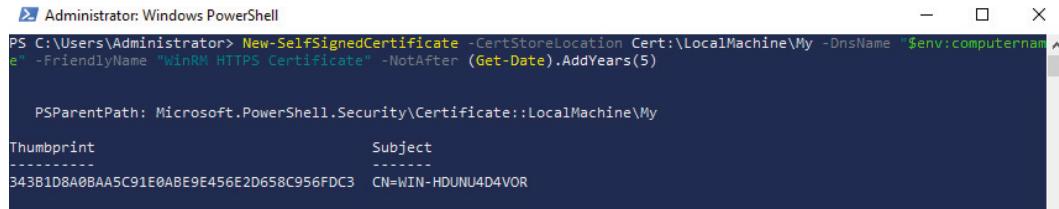
- A CN value matching the hostname
- Server Authentication (1.3.6.1.5.5.7.3.1) in the **Enhanced Key Usage** field

Ideally, this should be generated by a central **certificate authority (CA)** to prevent man-in-the-middle attacks and similar—more on this later. However, to provide all readers with an example they will be able to test out, we will generate a self-signed certificate. Run the following command in PowerShell to generate a suitable certificate:

```
New-SelfSignedCertificate -CertStoreLocation Cert:\LocalMachine\My -DnsName "$env:computername" -FriendlyName "WinRM HTTPS Certificate" -NotAfter (Get-Date).AddYears(5)
```

The `New-SelfSignedCertificate` command is only available on newer versions of Windows—if it is not available on your system, consider using the automated PowerShell script provided by Ansible available at <https://raw.githubusercontent.com/ansible/ansible/devel/examples/scripts/ConfigureRemotingForAnsible.ps1>.

This should yield something like that shown in *Figure 4.3*—make a note of the certificate thumbprint, as you will need it later:



```
Administrator: Windows PowerShell
PS C:\Users\Administrator> New-SelfSignedCertificate -CertStoreLocation Cert:\LocalMachine\My -DnsName "$env:computername" -FriendlyName "WinRM HTTPS Certificate" -NotAfter (Get-Date).AddYears(5)

PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\My

Thumbprint Subject
----- -----
343B1D8A0BAA5C91E0ABE9E456E2D658C956FDC3 CN=WIN-HDUNU4D4VOR
```

Figure 4.3 – Creating a self-signed certificate for the WinRM HTTPS listener using PowerShell

With the certificate in place, we can now set up a new WinRM listener with the following command:

```
New-Item -Path WSMAN:\localhost\Listener -Transport HTTPS
-Address * -CertificateThumbprint <thumbprint of certificate>
```

When successful, that command sets up a WinRM HTTPS listener on port 5986 with the self-signed certificate we generated earlier. To enable Ansible to automate this Windows host through WinRM, we need to perform two more steps—open up this port on the firewall and enable basic authentication so that we can test using the local Administrator account. This is achieved with the following two commands:

```
New-NetFirewallRule -DisplayName "WinRM HTTPS Management"
-Profile Domain,Private -Direction Inbound -Action Allow
-Protocol TCP -LocalPort 5986

Set-Item -Path "WSMAN:\localhost\Service\Auth\Basic" -Value
$true
```

You should see output from the previous commands similar to that shown in *Figure 4.4*:

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command `New-Item` is used to create a new WSMAN Listener item at the path `WSMan:\localhost\Listener`. The transport is set to HTTPS and the address is specified as \*. The certificate thumbprint is provided. A confirmation message asks if the user wants to continue, and the response is "y". The resulting configuration is displayed as a table:

Type	Keys	Name
Container	{Transport=HTTPS, Address=*}	Listener_1305953032

Next, the command `New-NetFirewallRule` is used to create a new firewall rule named "WinRM HTTPS Management" for the Domain and Private profiles. The rule is inbound, allows TCP traffic on port 5986, and is set to Allow. The resulting rule configuration is shown:

Name	DisplayName	Description	DisplayGroup	Group	Enabled	Profile	Platform	Direction	Action	EdgeTraversalPolicy	LooseSourceMapping	LocalOnlyMapping	Owner	PrimaryStatus	Status	EnforcementStatus	PolicyStoreSource	PolicyStoreSourceType
	: {d6b9ae3e-684f-41e7-8023-1f7c6fe3e0f5}	: WinRM HTTPS Management	:	:	: True	: Domain, Private	: {}	: Inbound	: Allow	: Block	: False	: False	:	: OK	: The rule was parsed successfully from the store. (65536)	: NotApplicable	: PersistentStore	: Local

Finally, the command `Set-Item` is used to enable the basic authentication service on the WinRM listener.

Figure 4.4 – Creating and enabling access to the WinRM HTTPS listener in PowerShell

These commands have been broken out individually to give you an idea of the process involved in setting up a Windows host for Ansible connectivity. For automated deployments and systems where `New-SelfSignedCertificate` isn't available, consider using the `ConfigureRemotingForAnsible.ps1` script available on the official Ansible GitHub account, which we referenced earlier in this section. This script performs all the steps we completed previously (and more), and can be downloaded and run in PowerShell as follows:

```
$url = "https://raw.githubusercontent.com/ansible/ansible/
devel/examples/scripts/ConfigureRemotingForAnsible.ps1"
$file = "$env:temp\ConfigureRemotingForAnsible.ps1"
```

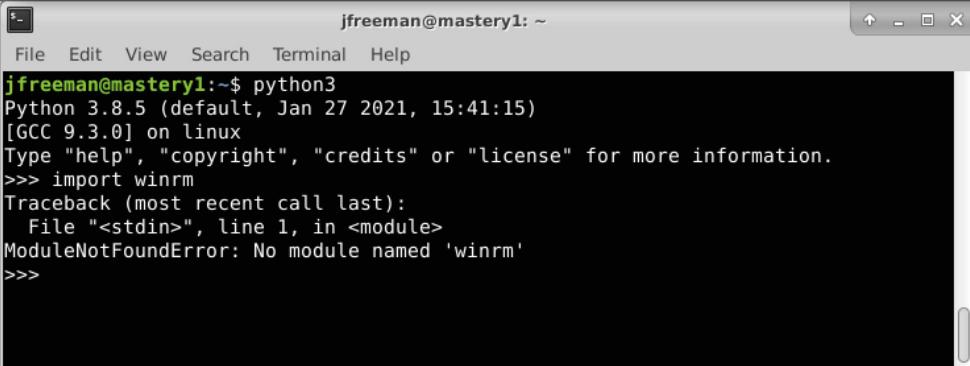
```
(New-Object -TypeName System.Net.WebClient).DownloadFile($url,  
$file)  
  
powershell.exe -ExecutionPolicy ByPass -File $file
```

There are many other ways to roll out the required configuration of WinRM for Ansible, including via Group Policy, which will almost certainly be preferable in corporate environments. The information provided in this section of the chapter should by now have provided you with all the fundamentals you need to set up WinRM in your environment, ready to enable Ansible management of your Windows hosts.

## Connecting Ansible to Windows using WinRM

Once WinRM is configured, getting Ansible talking to Windows is fairly straightforward, provided you bear two caveats in mind—it expects to use the SSH protocol, and if you don't specify a user account, it will attempt to use the same user account that Ansible is being run under to connect. This is almost certainly not going to work with a Windows username.

Also, note that Ansible requires the `winrm` Python module installed to connect successfully. This is not always installed by default, so it is worth testing for it on your Ansible system before you start working with Windows hosts. If it is not present, you will see something like the error shown in *Figure 4.5*:



```
jfreeman@mastery1:~$ python3  
Python 3.8.5 (default, Jan 27 2021, 15:41:15)  
[GCC 9.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import winrm  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ModuleNotFoundError: No module named 'winrm'  
>>>
```

Figure 4.5 – A simple test for the presence of the `winrm` Python module on Ubuntu Server 20.04

If you see this error, you will need to install the module before proceeding any further. There may be a prepackaged version available for your OS—for example, on Ubuntu Server 20.04, you can install it with the following command:

```
sudo apt install python3-winrm
```

If a packaged version is not available, install it directly from pip using the following command. Note that in *Chapter 2, Migrating from an Earlier Ansible Versions*, we discussed the use of Python virtual environments for installing Ansible – if you have done this, you must be sure to activate your virtualenv, and then run the following command without sudo:

```
sudo pip3 install "pywinrm>=0.3.0"
```

Once this is complete, we can test to see whether our earlier WinRM configuration work was successful. For SSH-based connectivity, there is an Ansible module called `ansible.builtin.ping`, which performs a full end-to-end test to ensure connectivity, successful authentication, and a usable Python environment on the remote system. Similarly, there exists a module called `win_ping` (from the `ansible.windows` collection), which performs an analogous test on Windows.

In my test environment, I would prepare an inventory as follows to connect to my newly configured Windows host:

```
[windows]
10.50.0.101

[windows:vars]
ansible_user=Administrator
ansible_password="Password123"
ansible_port=5986
ansible_connection=winrm
ansible_winrm_server_cert_validation=ignore
```

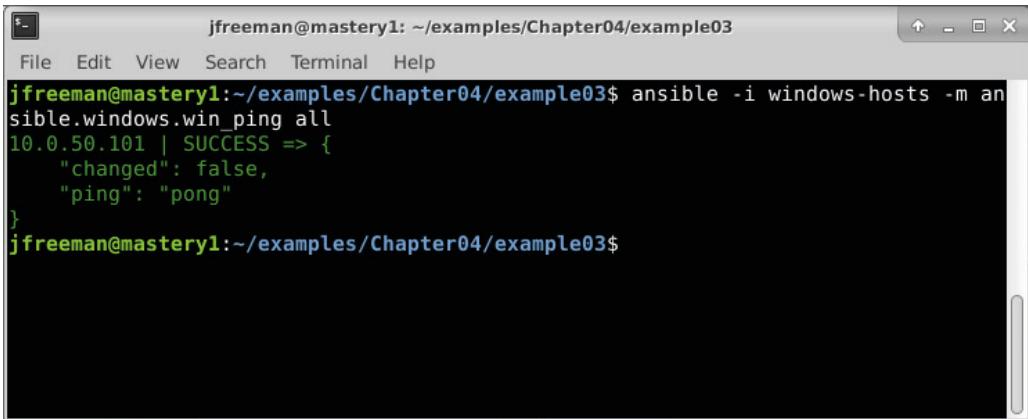
Note the connection-specific variables beginning `ansible_` that are being set in the `windows:vars` section of the playbook. At this stage, they should be fairly self-explanatory, as they were covered in *Chapter 1, The System Architecture and Design of Ansible*, but, in particular, note the `ansible_winrm_server_cert_validation` variable, which needs to be set to `ignore` when working with self-signed certificates. Obviously, in a real-world example, you would not leave the `ansible_password` parameter in clear text—it would either be placed in an Ansible vault or prompted for upon launch by using the `--ask-pass` parameter.

Certificate-based authentication is also possible with WinRM, which carries with it more or less the same benefits and risks as SSH key-based authentication.

Using the previous inventory (with appropriate changes for your environment such as hostname/IP addresses and authentication details), we can run the following command to test connectivity:

```
ansible -i windows-hosts -m ansible.windows.win_ping all
```

If all goes well, you should see some output like that shown in *Figure 4.6*:

A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter04/example03". The window shows the command "ansible -i windows-hosts -m ansible.windows.win\_ping all" being run. The output indicates a successful ping to the IP address 10.0.50.101, with the response "pong".

```
jfreeman@mastery1:~/examples/Chapter04/example03$ ansible -i windows-hosts -m ansible.windows.win_ping all
10.0.50.101 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
jfreeman@mastery1:~/examples/Chapter04/example03$
```

Figure 4.6 – Testing Windows host connectivity over WinRM using Ansible's  
ansible.windows.win\_ping module

That completes a successful end-to-end setup of an Ansible host to a Windows one! From such a setup, you can author and run playbooks just as you would on any other system, except that you must work with Ansible modules that specifically support Windows. Next, we will work on improving the security of our connection between Ansible and Windows, before finally moving on to some examples of Windows playbooks.

## Handling Windows authentication and encryption when using WinRM

Now that we have established the basic level of connectivity required for Ansible to perform tasks on a Windows host using WinRM, let's dig deeper into the authentication and encryption side of things. In the earlier part of the chapter, we used the basic authentication mechanism with a local account. While this is fine in a testing scenario, what happens in a domain environment? Basic authentication only supports local accounts, so clearly we need something else here. We also chose not to validate the SSL certificate (as it was self-signed), which again, is fine for testing purposes, but is not best practice in a production environment. In this section, we will explore options for improving the security of our Ansible communications with Windows.

## Authentication mechanisms

Ansible, in fact, supports five different Windows authentication mechanisms when WinRM is used, as follows:

- **Basic:** Supports local accounts only
- **Certificate:** Supports local accounts only, conceptually similar to SSH key-based authentication
- **Kerberos:** Supports AD accounts
- **NTLM:** Supports both local and AD accounts
- **CredSSP:** Supports both local and AD accounts

It is worth noting that Kerberos, NTLM, and CredSSP all provide message encryption over HTTP, which improves security. However, we have already seen how easy it is to set up WinRM over HTTPS, and WinRM management over plain HTTP is not enabled by default anyway, so we will assume that the communication channel is already encrypted. WinRM is a SOAP protocol, meaning it must run over a transport layer such as HTTP or HTTPS. To prevent remote management commands being intercepted on the network, it is best practice to ensure WinRM runs over the HTTPS protocol.

Of these authentication methods, the one that interests us most is Kerberos. Kerberos (for the purpose of this chapter) effectively supersedes NTLM for Ansible authentication against Active Directory accounts. CredSSP provides another mechanism, but there are also security risks relating to the interception of clear-text logons on the target host that are best understood before it is deployed—in fact, it is disabled by default.

Before we move on to configuring Kerberos, a brief note about certificate authentication. Although initially, this might seem appealing, as it is effectively passwordless, current dependencies in Ansible mean that the private key for the certificate authentication must be unencrypted on the Ansible automation host. In this regard, it is actually more secure (and wiser) to place the password for either a basic or Kerberos authentication session in an Ansible vault. We have already covered basic authentication, and so we will focus our efforts on Kerberos here.

As Kerberos authentication only supports Active Directory accounts, it is assumed that the Windows host to be controlled by Ansible is already joined to the domain. It is also assumed that WinRM over HTTPS has already been set up, as discussed earlier in the chapter.

With these requirements in place, the first thing we have to do is install a handful of Kerberos-related packages on the Ansible host itself. The exact packages will depend upon your chosen OS, but on Red Hat Enterprise Linux/CentOS 8, it would look like this:

```
sudo dnf -y install python3-devel krb5-devel krb5-libs krb5-workstation
```

On Ubuntu 20.04, you would install the following packages:

```
sudo apt-get install python3-dev libkrb5-dev krb5-user
```

#### Information

Package requirements for Kerberos support on a wider range of OSes are available in the Ansible documentation for Windows Remote Management: [https://docs.ansible.com/ansible/latest/user\\_guide/windows\\_winrm.html](https://docs.ansible.com/ansible/latest/user_guide/windows_winrm.html).

In addition to these packages, we also need to install the `pywinrm[kerberos]` Python module. Availability of this will vary—on Red Hat Enterprise Linux/CentOS 8, it is not available as an RPM, so we need to install it through `pip` as follows (again, if you have used a Python Virtual Environment, be sure to activate it and run the `pip3` command without `sudo`):

```
sudo dnf -y install gcc
sudo pip3 install pywinrm[kerberos]
```

Note that `gcc` is needed by `pip3` to build the module—this can be removed afterward if no longer required.

Next, ensure that your Ansible server can resolve your AD-related DNS entries. The procedure for this will vary according to the OS and network architecture, but it is vital that your Ansible controller must be able to resolve the name of your domain controller and related entries for the rest of this procedure to work.

Once you have configured your DNS settings for your Ansible control host, next, add your domain to `/etc/krb5.conf`. For example, my test domain is `mastery.example.com`, and my domain controller is `DEMODEM-O5NVEP9.mastery.example.com`, so the bottom of my `/etc/krb5.conf` file looks like this:

```
[realms]
MASTERY.EXAMPLE.COM = {
    kdc = DEMODEM-O5NVEP9.mastery.example.com
```

```
}
```

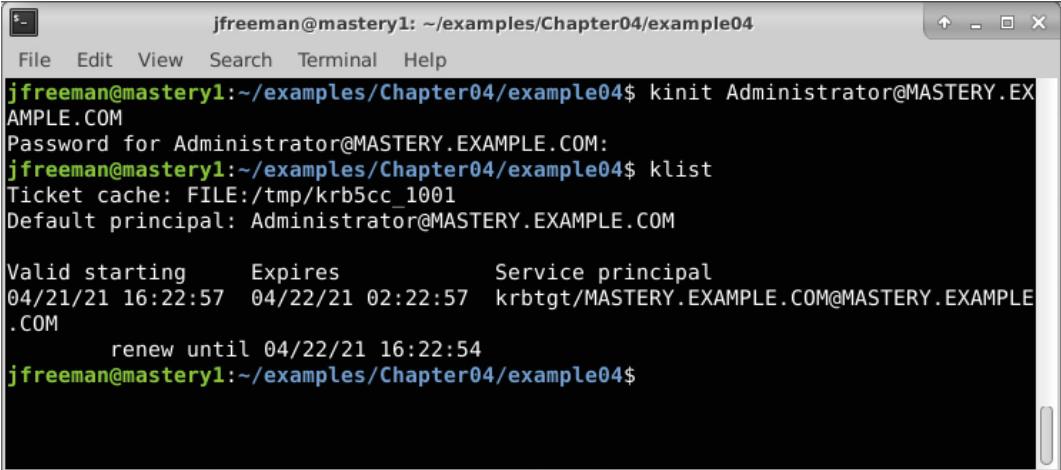
---

```
[domain_realm]
.mastery.example.com = MASTERY.EXAMPLE.COM
```

Note the capitalization—this is important! Test your Kerberos integration using the `kinit` command with a known domain user account. For example, I will test the integration with my test domain using the following commands:

```
kinit Administrator@MASTERY.EXAMPLE.COM
klist
```

A successful test should look like the one shown in *Figure 4.7*:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter04/example04". The window contains the following text:

```
jfreeman@mastery1:~/examples/Chapter04/example04$ kinit Administrator@MASTERY.EXAMPLE.COM
Password for Administrator@MASTERY.EXAMPLE.COM:
jfreeman@mastery1:~/examples/Chapter04/example04$ klist
Ticket cache: FILE:/tmp/krb5cc_1001
Default principal: Administrator@MASTERY.EXAMPLE.COM

Valid starting      Expires          Service principal
04/21/21 16:22:57  04/22/21 02:22:57  krbtgt/MASTERY.EXAMPLE.COM@MASTERY.EXAMPLE.COM
      renew until 04/22/21 16:22:54
jfreeman@mastery1:~/examples/Chapter04/example04$
```

Figure 4.7 – Testing Kerberos integration between an Ubuntu Ansible control host  
and a Windows domain controller

Finally, let's create a Windows host inventory—note that it is almost identical to the one we used in our basic authentication example; only this time, we have specified the Kerberos domain after the username:

```
[windows]
10.0.50.103

[windows:vars]
ansible_user=Administrator@MASTERY.EXAMPLE.COM
ansible_password="Password123"
```

```
ansible_port=5986
ansible_connection=winrm
ansible_winrm_server_cert_validation=ignore
```

Now, we can test connectivity just like before:

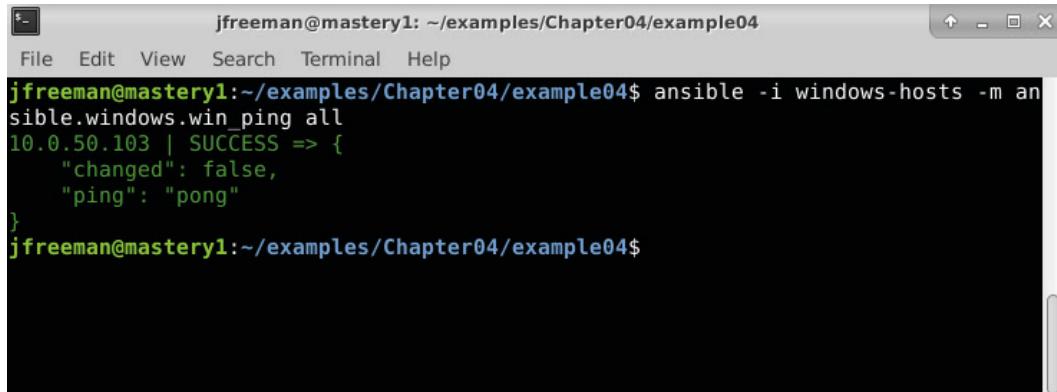
A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter04/example04". The window shows the command "ansible -i windows-hosts -m ansible.windows.win\_ping all" being run. The output indicates success for host "10.0.50.103", showing a "changed": false and a "ping": "pong" response. The terminal window has a standard Linux-style interface with a menu bar and a scroll bar on the right.

Figure 4.8 – An Ansible connectivity test using the `ansible.windows.win_ping` module and Kerberos authentication

Success! The previous result shows successful end-to-end connectivity with Windows, including successful authentication with a domain account using Kerberos, and access to the WinRM subsystem.

## A note on accounts

By default, WinRM is configured to only allow management by members of the local `Administrators` group on a given Windows host. This does not have to be the administrator account itself—we have used this here for demonstration purposes. It is possible to enable the use of less privileged accounts for WinRM management, but their use is likely to prove limited, as most Ansible commands require a degree of privileged access. Should you wish to have a less privileged account available to Ansible via WinRM, run the following command on the host:

```
winrm config SDDL default
```

Running this command opens a Windows dialog box. Use this to add and grant (as a minimum) the `Read` and `Execute` privileges to any user or group you wish to have WinRM remote management capabilities.

## Certificate validation over WinRM

So far, we have been ignoring the self-signed SSL certificates used in WinRM communication—obviously, this is less than ideal, and it is quite straightforward to get Ansible to validate SSL certificates if they are not self-signed.

The easiest way to do this if your Windows machines are members of a domain is to use **Active Directory Certificate Services (ADCS)**—however, most businesses will have their own certification process in place through ADCS, or another third-party service. It is assumed, in order to proceed with this section, that the Windows host in question has a certificate generated for remote management, and that the CA certificate is available in Base64 format.

Just as we did earlier on the Windows host, you will need to set up an HTTPS listener, but this time using the certificate signed by your CA. You can do so (if not already completed) using a command such as the following:

```
Import-Certificate -FilePath .\certnew.cer -CertStoreLocation  
Cert:\LocalMachine\My
```

Naturally, replace the `FilePath` certificate with the one that matches the location of your own certificate. If you need to, you can delete any previously created HTTPS WinRM listener with the following command:

```
winrm delete winrm/config/Listener?Address=*+Transport=HTTPS
```

Then, using the thumbprint from the imported certificate, create a new listener:

```
New-Item -Path WSMan:\localhost\Listener -Transport HTTPS  
-Address * -CertificateThumbprint <thumbprint of certificate>
```

Now to the Ansible controller. The first thing to do is to import the CA certificate for the WinRM listener into the CA bundle for your OS. The method and location for this will vary between OSes, but, on Ubuntu Server 20.04, you can place the Base64-encoded CA certificate in `/usr/share/ca-certificates/`. Note that in order to be recognized, the CA file must have the `.crt` extension.

Once this has been done, run the following commands:

```
sudo dpkg-reconfigure ca-certificates
```

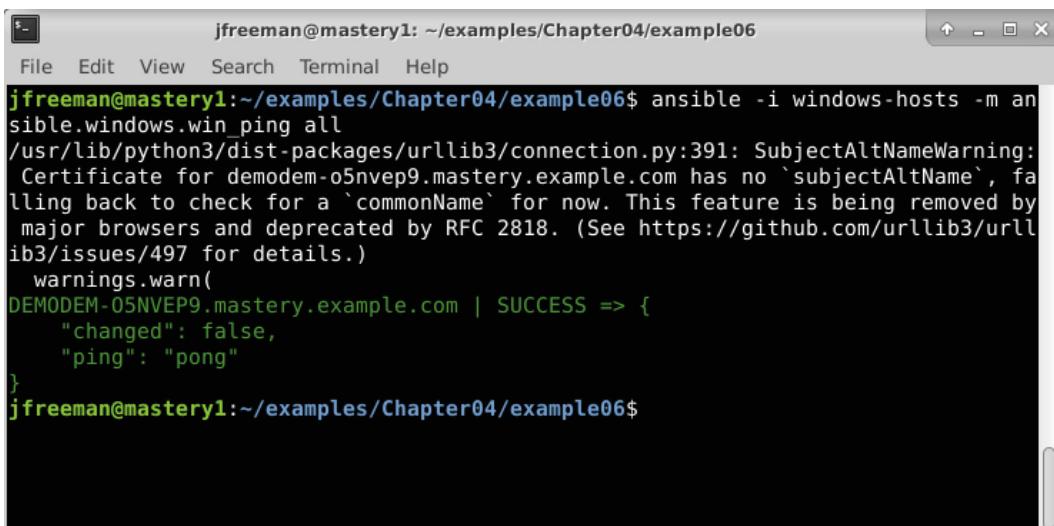
Select `Yes` when asked if you want to trust certificates from new certificate authorities, and ensure that your new certificate filename is selected in the list presented on the next screen.

Finally, we need to tell Ansible where to find the certificate. By default, Ansible uses the Python Certifi module and will use the default path for this unless we tell it otherwise. The above process updates the CA bundle, located in `/etc/ssl/certs/ca-certificates.crt`, and luckily, we can tell Ansible where to find this in the inventory file. Note the two further changes to the inventory file as shown in the following code—first of all, we have now specified the full hostname for the Windows host rather than the IP address, as the inventory hostname must match the CN value on the certificate for full validation to occur. Also, we have removed the `ansible_winrm_server_cert_validation` line, which means all SSL certificates are now implicitly validated:

```
[windows]
DEMODEM-05NVEP9.mastery.example.com

[windows:vars]
ansible_user=administrator@MASTERY.EXAMPLE.COM
ansible_password="Password123"
ansible_port=5986
ansible_connection=winrm
ansible_winrm_ca_trust_path=/etc/ssl/certs/ca-certificates.crt
```

If we run our ping test again, we should now see SUCCESS, as shown in *Figure 4.9*:

A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter04/example06". The window shows the command "ansible -i windows-hosts -m ansible.windows.win\_ping all" being run. The output includes a warning about a SubjectAltNameWarning regarding a certificate for "demodem-05nvep9.mastery.example.com". It then lists the results for each host: "DEMODEM-05NVEP9.mastery.example.com | SUCCESS => {"changed": false, "ping": "pong"}".

```
jfreeman@mastery1:~/examples/Chapter04/example06$ ansible -i windows-hosts -m ansible.windows.win_ping all
/usr/lib/python3/dist-packages/urllib3/connection.py:391: SubjectAltNameWarning: Certificate for demodem-05nvep9.mastery.example.com has no `subjectAltName`, falling back to check for a `commonName` for now. This feature is being removed by major browsers and deprecated by RFC 2818. (See https://github.com/urllib3/urllib3/issues/497 for details.)
  warnings.warn(
DEMODEM-05NVEP9.mastery.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
jfreeman@mastery1:~/examples/Chapter04/example06$
```

Figure 4.9 – Ansible ping test using Kerberos authentication and SSL validation to a Windows domain controller over WinRM

Obviously, we could improve our certificate generation to remove the `subjectAltName` warning, but for now, this demonstrates Ansible connectivity to Windows, with Kerberos authentication to a domain account and full SSL validation. This completes our look at setting up WinRM, and should provide you with all the fundamentals you need to set up Windows hosts in your infrastructure for automation with Ansible.

In the next part of this chapter, we will take a look at setting up the newly supported OpenSSH server on Windows to enable Ansible automation.

## Setting up Windows hosts for Ansible control using OpenSSH

Microsoft has made great strides in supporting and embracing the open source community, and has added a number of popular open source packages to their OSes. One of the most notable as far as Ansible automation is concerned is the venerable and incredibly popular OpenSSH package, which comes in both client and server flavors.

Support for automating tasks on Windows using SSH as the transport rather than WinRM was added in Ansible 2.8 – however, it should be noted that there are many warnings about this support in the official Ansible documentation – support is described as experimental, and users are warned that things might change in the future in a way that is not backward compatible. In addition, developers expect to uncover more bugs as they continue their testing.

For these reasons, we have put a lot of effort into describing the setup of WinRM for automating Windows hosts with Ansible. Nonetheless, this chapter would not be complete without a look at enabling Ansible automation for Windows using OpenSSH.

OpenSSH Server for Windows is supported on Windows 10 version 1809 and later, and also Windows Server 2019. If you are running an older version of Windows, you have two choices – either stay with WinRM as your communication protocol (after all, it is built in and easy to configure once you know how), or manually install the Win32-OpenSSH package – this process is described in detail here, and should support anything from Windows 7 onward: <https://github.com/PowerShell/Win32-OpenSSH/wiki/Install-Win32-OpenSSH>. Given the active development of this package, readers are advised to refer to this documentation if they want to install OpenSSH Server on an older version of Windows as the instructions might have changed by the time the book gets to print.

If, however, you are running one of the newer versions of Windows, installing the OpenSSH Server is a simple matter. Using a PowerShell session with Administrator privileges, first of all, use the following command to query the available OpenSSH options:

```
Get-WindowsCapability -Online | ? Name -like 'OpenSSH*'
```

The output from this command should look something like that in *Figure 4.10*:

```
Administrator: Windows PowerShell
PS C:\Users\Administrator\Downloads> Get-WindowsCapability -Online | ? Name -like 'OpenSSH*'

Name : OpenSSH.Client~~~0.0.1.0
State : Installed

Name : OpenSSH.Server~~~0.0.1.0
State : NotPresent

PS C:\Users\Administrator\Downloads>
```

Figure 4.10 – Showing available OpenSSH installation options in PowerShell on Windows Server 2019  
Using this output, run the following command to install the OpenSSH Server:

```
Add-WindowsCapability -Online -Name OpenSSH.Server~~~0.0.1.0
```

Next, run the following commands to ensure the SSH server service starts at boot time, that it is started, and that a suitable firewall rule exists to allow SSH traffic to the server:

```
Start-Service sshd
Set-Service -Name sshd -StartupType 'Automatic'
Get-NetFirewallRule -Name *ssh*
```

If an appropriate firewall rule isn't present, you can add one in with a command such as the following:

```
New-NetFirewallRule -Name sshd -DisplayName 'OpenSSH Server
(sshd)' -Enabled True -Direction Inbound -Protocol TCP -Action
Allow -LocalPort 22
```

Finally, the OpenSSH server for Windows defaults to cmd for its shell. This is fine for interactive tasks, but most of the native Ansible modules for Windows are written to support PowerShell – you can change the default shell for the OpenSSH server by running the following command in PowerShell:

```
New-ItemProperty -Path 'HKLM:\SOFTWARE\OpenSSH' -Name  
'DefaultShell' -Value 'C:\Windows\System32\WindowsPowerShell\  
v1.0\powershell.exe'
```

With all these tasks complete, we can finally test our `ansible.windows.win_ping` module just as we did before. Our inventory file will look a little different from the WinRM one – the following one should serve as a suitable example for your testing purposes:

```
[windows]  
DEMODEM-O5NVEP9.mastery.example.com  
  
[windows:vars]  
ansible_user=administrator@MASTERY.EXAMPLE.COM  
ansible_password="Password123"  
ansible_shell_type=powershell
```

Note that we are no longer concerned with certificate validation or port numbers as we are using SSH over the default port, 22. In fact, apart from the username and password (which you could easily specify as command-line arguments to the `ansible` command just as we did earlier in this book), the only inventory variable that needs setting is `ansible_shell_type`, which will default to a Bourne-compatible shell unless we tell it otherwise.

The `win_ping` module uses PowerShell when testing connectivity, enabling us to use our previous ad hoc command to test our new SSH connectivity to Windows. Simply run this command (which should look familiar by now!):

```
ansible -i windows-hosts -m ansible.windows.win_ping all
```

Even though we have now used a completely different communication protocol, the output from this command is exactly the same, and should look like the following *Figure 4.11*:

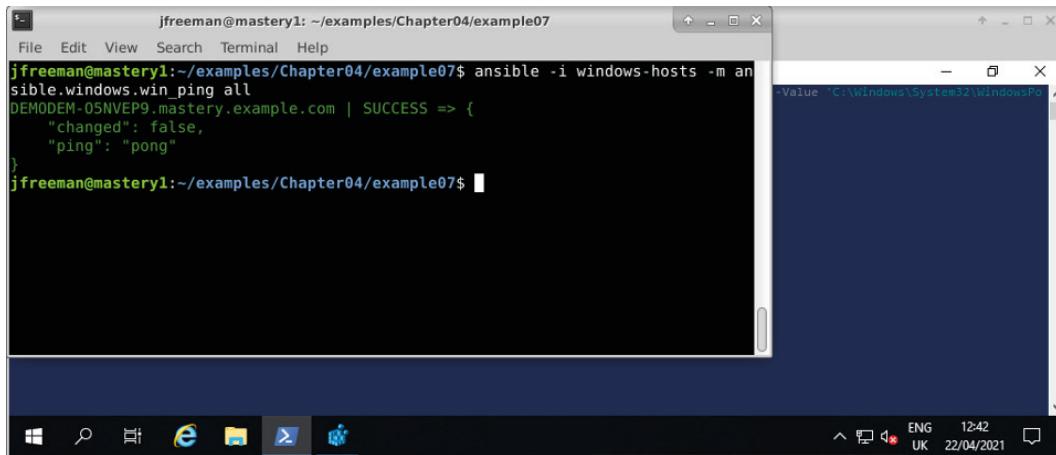
A screenshot of a Windows terminal window titled "jfreeman@mastery1: ~/examples/Chapter04/example07". The window shows the command "ansible -i windows-hosts -m ansible.windows.win\_ping all" being run. The output indicates success for the host "DEMODEM-05NVEP9.mastery.example.com", showing a "changed": false and a "ping": "pong" response. The terminal window is positioned over a dark blue Windows desktop background. The taskbar at the bottom shows various icons, and the system tray indicates the date and time as 22/04/2021.

Figure 4.11 – Testing Ansible integration with Windows using SSH as the transport mechanism

Thus, integrating Ansible with Windows hosts is really quite simple to set up – just be sure to keep your eyes on the release notes and porting guides for newer Ansible releases in case things change in some non-compatible way. However, I think you'll agree that integrating Ansible with Windows using OpenSSH is simple to set up. Of course, you can set up SSH key authentication in a similar manner to that on any other SSH-based host to ensure you can run playbooks without the need for user interaction.

Now, in demonstrating aspects of Windows integration with Ansible through both WinRM and SSH, we have only used the Ansible `ansible.windows.win_ping` module to test connectivity. Let's build on this by wrapping up the chapter with some simple example playbooks to help you get started on creating your own Windows automation solutions with Ansible.

## Automating Windows tasks with Ansible

A list of the Windows modules included with Ansible 4.3 is available at the following link, and it must be noted that, although you can use all the familiar Ansible constructs with Windows hosts such as `vars`, `handlers`, and `blocks`, you must use Windows-specific modules when defining tasks. The introduction of collections means it is quite easy to locate them, and the `ansible.windows` collection is a great place to start. This contains all the Windows-specific modules you were used to using in Ansible 2.9 and earlier: [https://docs.ansible.com/ansible/latest/collections/index\\_module.html#ansible-windows](https://docs.ansible.com/ansible/latest/collections/index_module.html#ansible-windows).

In this part of the chapter, we will run through a few simple examples of Windows playbooks to highlight a few of the things you need to know when writing playbooks for Windows.

## Picking the right module

If you were running Ansible against a Linux server, and wanted to create a directory and then copy a file into it, you would use the `ansible.builtin.file` and `ansible.builtin.copy` Ansible modules, in a playbook that looks something like the following:

```
---
- name: Linux file example playbook
  hosts: all
  gather_facts: false

  tasks:
    - name: Create temporary directory
      ansible.builtin.file:
        path: /tmp/mastery
        state: directory
    - name: Copy across a test file
      ansible.builtin.copy:
        src: mastery.txt
        dest: /tmp/mastery/mastery.txt
```

However, on Windows, this playbook would fail to run, as the `ansible.builtin.file` and `ansible.builtin.copy` modules are not compatible with PowerShell or cmd, regardless of whether you use WinRM or SSH as your communication protocol with the Windows machine. As a result, an equivalent playbook to perform the same task, but on Windows, would look like this:

```
---
- name: Windows file example playbook
  hosts: all
  gather_facts: false

  tasks:
    - name: Create temporary directory
      ansible.windows.win_file:
```

```
path: 'C:\Mastery Test'  
state: directory  
- name: Copy across a test file  
  ansible.windows.win_copy:  
    src: ~/src/mastery/mastery.txt  
    dest: 'C:\Mastery Test\mastery.txt'
```

Note the following differences between the two playbooks:

- `ansible.windows.win_file` and `ansible.windows.win_copy` are used in place of the `ansible.builtin.file` and `ansible.builtin.copy` modules.
- It is recommended in the documentation for the `ansible.windows.win_file` and `ansible.windows.win_copy` modules to use a backslash (\) when dealing with remote (Windows paths).
- Continue to use forward slashes (/) on the Linux host.
- Use single quotes (not double quotes) to quote paths that contain spaces.

It is always important to consult the documentation for the individual modules used in your playbooks. For example, reviewing the `ansible.windows.win_copy` module documentation, it recommends using the `ansible.windows.win_get_url` module for large file transfers because the WinRM transfer mechanism is not very efficient. Of course, if you are using the OpenSSH server in place of WinRM, this may not apply – at the time of writing, the documentation for this module has not been updated to take account of this.

Also note that, if a filename contains certain special characters (for example, square braces), they need to be escaped using the PowerShell escape character, `\``. For example, the following task would install the `c:\temp\setupdownloader_[aaff].exe` file:

```
- name: Install package  
  win_package:  
    path: 'c:\temp\setupdownloader_`[aaff`\].exe'  
    product_id: {00000000-0000-0000-0000-000000000000}  
    arguments: /silent /unattended  
    state: present
```

Many other Windows modules should suffice to complete your Windows playbook needs, and, combined with these tips, you will get the end results you need, quickly and with ease.

## Installing software

Most Linux systems (and indeed other Unix variants) have a native package manager that makes it easy to install a wide variety of software. The chocolatey package manager makes this possible for Windows, and the Ansible chocolatey.chocolatey.win\_chocolatey module makes installing software in an unattended manner with Ansible simple (note that this is not part of the ansible.windows collection that we have used so far, but instead lives in its own collection).

You can explore the chocolatey repository and find out more about it at <https://chocolatey.org>.

For example, if you wanted to roll out Adobe's Acrobat Reader across an estate of Windows machines, you could use either the ansible.windows.win\_copy or ansible.windows.win\_get\_url modules to distribute the installer, and then the ansible.windows.win\_package module to install it. However, the following code would perform the same task with less code:

```
- name: Install Acrobat Reader
  chocolatey.chocolatey.win_chocolatey:
    name: adobereader
    state: present
```

There are all manner of clever installation routines you can run using the chocolatey.chocolatey.win\_chocolatey module – for example, you can lock a package to a specific version, install a specific architecture, and much more – the documentation for this module includes a great many useful examples. The official Chocolatey website itself lists all the available packages – most of the common ones you would expect to need can be found there, so it should suffice for a great many installation scenarios you will come across.

## Extending beyond modules

Just as on any platform, there may come a time when the exact functionality required is not available from a module. Although writing a custom module (or modifying an existing one) is a viable solution to this, sometimes, a more immediate solution is required. To this end, the `ansible.windows.win_command` and `ansible.windows.win_shell` modules come to the rescue—these can be used to run literal PowerShell commands on Windows. Many examples are available in the official Ansible documentation, but the following code, for example, would create the `C:\Mastery` directory using PowerShell:

```
- name: Create a directory using PowerShell
  ansible.windows.win_shell: New-Item -Path C:\Mastery
  -ItemType Directory
```

We could even revert to the traditional cmd shell for this task:

```
- name: Create a directory using cmd.exe
  ansible.windows.win_shell: mkdir C:\MasteryCMD
  args:
    executable: cmd
```

With these pointers, it should be possible to create the desired functionality in just about any Windows environment.

That concludes our look at Windows automation with Ansible – as long as you remember to use the correct Windows native modules, you will be able to apply the rest of this book to Windows hosts just as easily as you would any given Linux host.

## Summary

Ansible handles Windows hosts as effectively as Linux (and other Unix) ones. In this chapter, we covered both how to run Ansible from a Windows host, and how to integrate Windows hosts with Ansible for automation, including the authentication mechanisms, encryption, and even the basics of Windows-specific playbooks.

You have learned that Ansible can run from a recent build of Windows that supports WSL, and how to achieve this. You have also learned how to set up Windows hosts for Ansible control and how to secure this with Kerberos authentication and encryption. You also learned how to set up and use the new and experimental support for SSH communication by Ansible with Windows hosts. Finally, you learned the basics of authoring Windows playbooks, including finding the correct modules for use with Windows hosts, escaping special characters, creating directories and copy files for the host, installing packages, and even running raw shell commands on the Windows host with Ansible. This is a sound foundation on which you will be able to build out the Windows playbooks needed to manage your own estate of Windows hosts.

In the next chapter, we will cover the effective management of Ansible in the enterprise with AWX.

## Questions

1. Ansible can communicate with Windows hosts using:
  - a) SSH
  - b) WinRM
  - c) Both of the above
2. Ansible can reliably be run from Windows:
  - a) Natively
  - b) Using Python for Windows
  - c) Through Cygwin
  - d) Through WSL or WSL2
3. The `ansible.builtin.file` module can be used to manipulate files on both Linux and Windows hosts:
  - a) True
  - b) False
4. Windows machines can have Ansible automation run on them with no initial setup:
  - a) True
  - b) False

5. The package manager for Windows is called:
  - a) Bournville
  - b) Cadbury
  - c) Chocolatey
  - d) RPM
6. Ansible modules for Windows run their commands by default using:
  - a) PowerShell
  - b) cmd.exe
  - c) Bash for Windows
  - d) WSL
  - e) Cygwin
7. You can run Windows commands directly even if a module with the functionality you need does not exist:
  - a) True
  - b) False
8. When manipulating files and directories on Windows with Ansible, you should:
  - a) Use \ for Windows path references, and / for files on the Linux host
  - b) Use / for all paths
9. Special characters in Windows filenames should be escaped with:
  - a) \
  - b) ^
  - c) "
  - d) /
10. Your Ansible playbooks must be changed depending on whether you are using WinRM or SSH communication:
  - a) True
  - b) False



# 5

# Infrastructure Management for Enterprises with AWX

It is clear that Ansible is an incredibly powerful and versatile automation tool, lending itself well to managing an entire estate of servers and network devices. Mundane, repetitive tasks can be made repeatable and straightforward, saving a great deal of time! Obviously, this is of great benefit in a corporate environment. However, this power comes at a price. If everyone has their own copy of Ansible on their own machines, how do you know who ran what playbook, and when? How do you ensure that all playbooks are correctly stored and version-controlled? Furthermore, how do you prevent the proliferation of superuser-level access credentials across your organization, while benefiting from the power of Ansible?

The answer to these questions comes in the form of AWX, an open-source enterprise management system for Ansible. AWX is the open-source, upstream version of the commercial Ansible Tower software available from Red Hat, and it offers virtually the same features and benefits, but without the support or product release cycle that Red Hat offers. AWX is a powerful, feature-rich product that includes not only a GUI to make it easy for non-Ansible users to run playbooks, but also a complete API for integration into larger workflows and CI/CD pipelines.

In this chapter, we will give you a solid foundation for installing and using AWX, specifically covering the following topics:

- Getting AWX up and running
- Integrating AWX with your first playbook
- Going beyond the basics

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running Ansible 4.3 or newer. Almost any flavor of Linux should do; for those interested in specifics, all the code presented in this chapter was tested on Ubuntu Server 20.04 LTS unless stated otherwise, and on Ansible 4.3. The example code that accompanies this chapter can be downloaded from GitHub at this URL: <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter05>.

Check out the following video to see the Code in Action video from Packt:  
<https://bit.ly/3ndx73Q>

## Getting AWX up and running

Before we get stuck into installing AWX, it is worth briefly exploring what AWX is, and what it isn't. AWX is a tool to be employed alongside Ansible. It does not duplicate or replicate, in any way, the features of Ansible. Indeed, when Ansible playbooks are run from AWX, the `ansible-playbook` executable is being called behind the scenes. AWX should be considered a complementary tool that adds the following benefits, on which many enterprises depend:

- Rich **role-based access control (RBAC)**
- Integration with centralized login services (for example, LDAP or AD)
- Secure credential management

- Auditability
- Accountability
- Lower barrier to entry for new operators
- Improved management of playbook version control
- Fully featured API

Most of the AWX code runs in a set of Linux containers. However, the standard installation method has changed since the last edition of the book, and it is now preferred that you deploy AWX on Kubernetes. If you are already proficient at Kubernetes, you may wish to try and deploy this in your own environment, as AWX should run on Red Hat's OpenShift, the open-source OKD, and any one of the many other existing flavors of Kubernetes.

If, however, you are not proficient in Kubernetes, or you are looking for some pointers on how to get started, then we will walk you through a complete installation of AWX from scratch in this part of the chapter. We will base this on the excellent `microk8s` distribution, which you can get up and running on a single node on Ubuntu Server with just one command!

Before we get started, one final note. Although Kubernetes is the preferred installation platform now, at the time of writing there is still an installation method available for Docker hosts. However, the maintainers of the AWX project note that this is targeted at development and test environments only, and has no official published release. As such, we will not cover this in this chapter. If you want to learn more, however, you can read the installation instructions at the following link: <https://github.com/ansible/awx/blob/devel/tools/docker-compose/README.md>.

With that covered, let's get started on our `microk8s`-based deployment. The installation process outlined here assumes you are starting from an unmodified Ubuntu Server 20.04 installation.

First of all, let's install `microk8s` itself, using the `snap` available with Ubuntu:

```
sudo snap install microk8s --classic
```

The only other step required is to add your user account to the `microk8s` group so that you can run the remaining commands in this section without needing `sudo` privileges:

```
sudo gpasswd -a $USER microk8s
```

You will need to log out and back in again for the change in group membership to apply to your account. Once you have done this, let's get started on preparing microk8s for our AWX deployment. We will need the storage, dns, and ingress add-ons for our deployment, so let's go ahead and enable them using the following command:

```
for i in storage dns ingress; do microk8s enable $i; done
```

Now we're ready to install AWX Operator, which is in turn used to manage the rest of the installation. Installing this is as simple as running the following:

```
microk8s kubectl apply -f https://raw.githubusercontent.com/
ansible/awx-operator/devel/deploy/awx-operator.yaml
```

The command will return immediately while the installation continues in the background. You can check on the status of the installation with the following command:

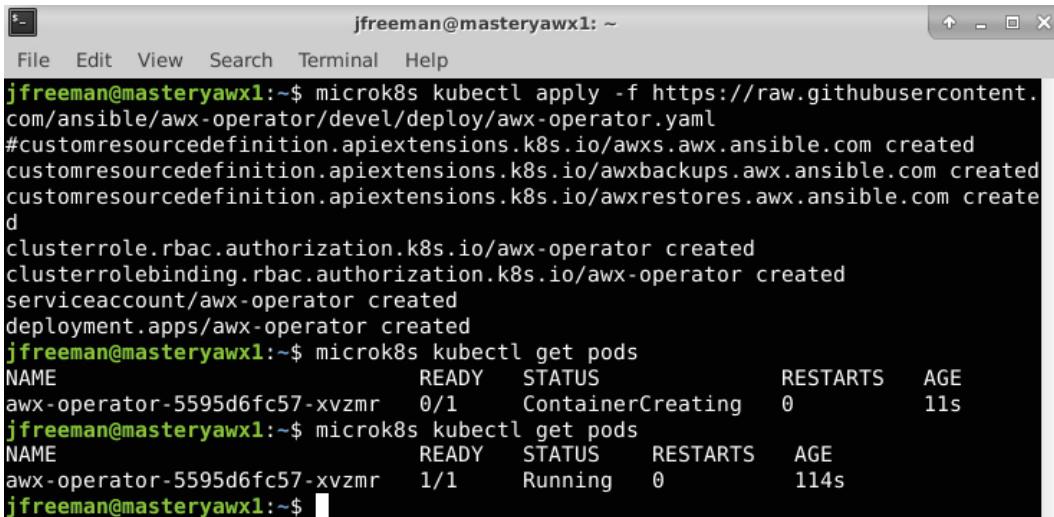
```
microk8s kubectl get pods
```

The STATUS field should say Running for the AWX Operator deployment once it is completed.

#### Important note

The previous command will clone the latest development release of AWX Operator. If you want to clone one of the releases, browse the *Releases* section of the repository, available at the following link, and check out your desired version: <https://github.com/ansible/awx-operator/releases>.

The screenshot in *Figure 5.1* shows the output following the successful deployment of AWX Operator:



```
jfreeman@masteryawx1:~$ microk8s kubectl apply -f https://raw.githubusercontent.com/ansible/awx-operator/devel/deploy/awx-operator.yaml
#customresourcedefinition.apiextensions.k8s.io/awxs.awx.ansible.com created
customresourcedefinition.apiextensions.k8s.io/awxbackups.awx.ansible.com created
customresourcedefinition.apiextensions.k8s.io/awxrestores.awx.ansible.com created
clusterrole.rbac.authorization.k8s.io/awx-operator created
clusterrolebinding.rbac.authorization.k8s.io/awx-operator created
serviceaccount/awx-operator created
deployment.apps/awx-operator created
jfreeman@masteryawx1:~$ microk8s kubectl get pods
NAME           READY   STATUS      RESTARTS   AGE
awx-operator-5595d6fc57-xvzmr   0/1     ContainerCreating   0          11s
jfreeman@masteryawx1:~$ microk8s kubectl get pods
NAME           READY   STATUS      RESTARTS   AGE
awx-operator-5595d6fc57-xvzmr   1/1     Running    0          114s
jfreeman@masteryawx1:~$
```

Figure 5.1 – The microk8s pod status following the successful deployment of AWX Operator

Next, we'll create a simple self-signed certificate for our AWX deployment. If you have your own certificate authority, you are of course welcome to generate your own certificate with the appropriate signing for your environment. If you are generating a self-signed certificate using the command that follows, be sure to replace awx.example.org with the hostname you have assigned to your AWX server:

```
openssl req -x509 -nodes -days 3650 -newkey rsa:2048 -keyout
awx.key -out awx.crt -subj "/CN=awx.example.org/O=mastery"
-addext "subjectAltName = DNS:awx.example.org"
```

We will create a secret (an object that contains a small amount of sensitive data) in Kubernetes containing our newly generated certificate:

```
microk8s kubectl create secret tls awx-secret-ssl --namespace
default --key awx.key --cert awx.crt
```

With this done, it's time to think about storage. AWX is designed to source its playbooks from source control repositories such as Git, and as such, the default installation does not provide easy access to local playbook files. However, for the purposes of creating a working example in this book that everyone can follow, we will create a persistent volume to store local playbooks. Create a YAML file called `my-awx-storage.yml`, containing the following:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: awx-pvc
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: microk8s-hostpath
  resources:
    requests:
      storage: 1Gi
```

Run the following command to create this storage using the YAML file we just created:

```
microk8s kubectl create -f my-awx-storage.yml
```

Now it's time to deploy AWX itself. To do this, we must create another YAML file that describes the deployment. We'll call this one `my-awx.yml`, and for our example, it should contain the following:

```
apiVersion: awx.ansible.com/v1beta1
kind: AWX
metadata:
  name: awx
spec:
  tower_ingress_type: Ingress
  tower_ingress_tls_secret: awx-secret-ssl
  tower_hostname: awx.example.org
  tower_projects_existing_claim: awx-pvc
  tower_projects_persistence: true
```

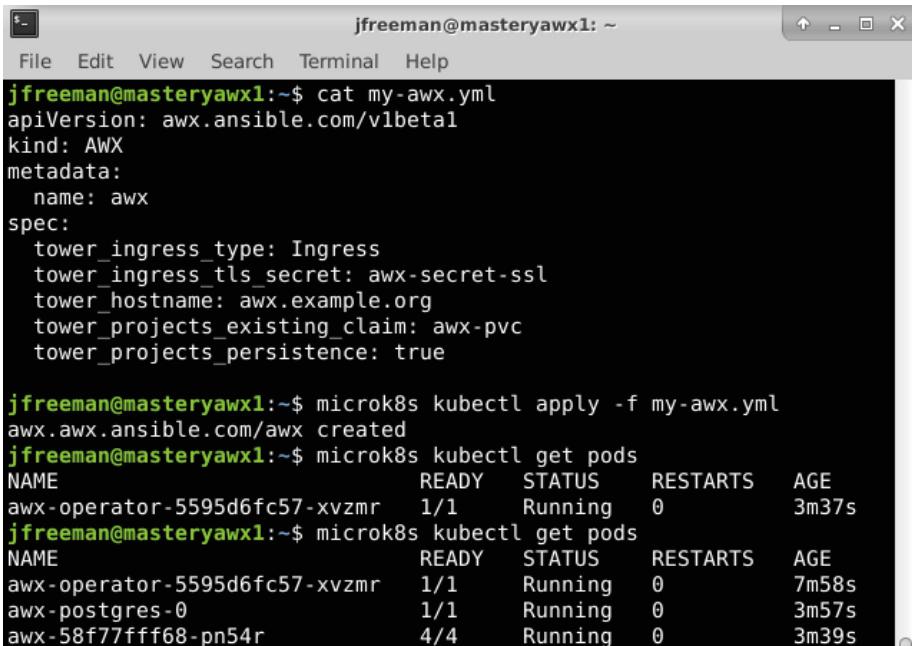
Deploy AWX using this file with the following command:

```
microk8s kubectl apply -f my-awx.yml
```

The deployment will take a few minutes, especially the first time you run it, as container images have to be downloaded in the background. You can check on the status with the following command:

```
microk8s kubectl get pods
```

When the deployment is complete, all pods should show STATUS as Running, as shown in *Figure 5.2*:



A screenshot of a terminal window titled "jfreeman@masteryawx1: ~". The window shows the following command-line session:

```
jfreeman@masteryawx1:~$ cat my-awx.yml
apiVersion: awx.ansible.com/v1beta1
kind: AWX
metadata:
  name: awx
spec:
  tower_ingress_type: Ingress
  tower_ingress_tls_secret: awx-secret-ssl
  tower_hostname: awx.example.org
  tower_projects_existing_claim: awx-pvc
  tower_projects_persistence: true

jfreeman@masteryawx1:~$ microk8s kubectl apply -f my-awx.yml
awx.awx.ansible.com/awx created
jfreeman@masteryawx1:~$ microk8s kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
awx-operator-5595d6fc57-xvzmr  1/1     Running   0          3m37s
jfreeman@masteryawx1:~$ microk8s kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
awx-operator-5595d6fc57-xvzmr  1/1     Running   0          7m58s
awx-postgres-0           1/1     Running   0          3m57s
awx-58f77ffff68-pn54r    4/4     Running   0          3m39s
```

Figure 5.2 – Kubernetes pod status after a successful AWX deployment

Of course, deploying AWX is only of limited use if we are unable to access it. We will use the ingress add-on of Microk8s to create an ingress router so that we can access our AWX deployment at our chosen hostname (`awx.example.org` in this example), over the standard HTTPS port. Create another YAML file, this time called `my-awx-ingress.yml`. It should contain the following:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
```

```
name: awx-ingress
annotations:
  nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  tls:
    - hosts:
        - awx.example.org
      secretName: awx-secret-ssl
  rules:
    - host: awx.example.org
      http:
        paths:
          - backend:
              service:
                name: awx-service
                port:
                  number: 80
              path: /
            pathType: Prefix
```

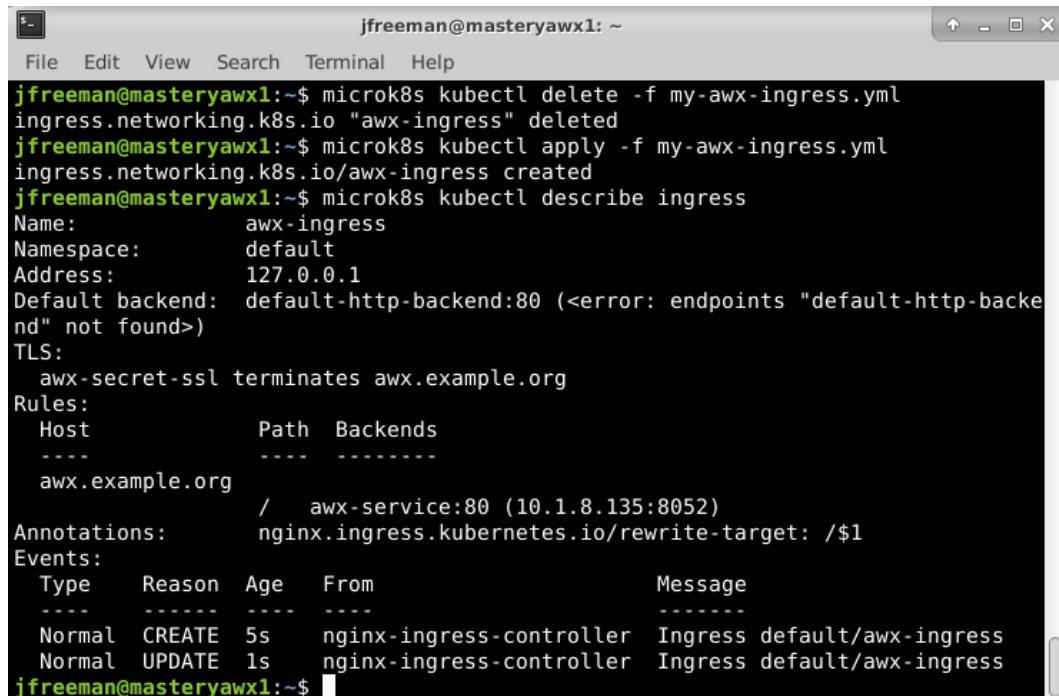
Deploy and then check this ingress definition with the following commands:

```
microk8s kubectl apply -f my-awx-ingress.yml
microk8s kubectl describe ingress
```

If you don't see an event with the Reason value set to CREATE, you may have to delete and then redeploy the ingress definition as follows:

```
microk8s kubectl delete -f my-awx-ingress.yml
microk8s kubectl apply -f my-awx-ingress.yml
```

A successful deployment of the ingress rule should look like that shown in the following figure:



The screenshot shows a terminal window titled "jfreeman@masteryawx1: ~". The user runs three commands: "microk8s kubectl delete -f my-awx-ingress.yml", "microk8s kubectl apply -f my-awx-ingress.yml", and "microk8s kubectl describe ingress". The output shows the ingress resource has been created with name "awx-ingress" in the "default" namespace, address "127.0.0.1", default backend "default-http-backend:80" (not found), and TLS termination for "awx.example.org" pointing to "awx-service:80". Annotations include "nginx.ingress.kubernetes.io/rewrite-target: /\$1". Events show two entries: "Normal CREATE 5s nginx-ingress-controller Ingress default/awx-ingress" and "Normal UPDATE 1s nginx-ingress-controller Ingress default/awx-ingress".

```
jfreeman@masteryawx1:~$ microk8s kubectl delete -f my-awx-ingress.yml
ingress.networking.k8s.io "awx-ingress" deleted
jfreeman@masteryawx1:~$ microk8s kubectl apply -f my-awx-ingress.yml
ingress.networking.k8s.io/awx-ingress created
jfreeman@masteryawx1:~$ microk8s kubectl describe ingress
Name:           awx-ingress
Namespace:      default
Address:        127.0.0.1
Default backend: default-http-backend:80 (<error: endpoints "default-http-backend" not found>
TLS:
  awx-secret-ssl terminates awx.example.org
Rules:
  Host          Path  Backends
  ----          ----  -----
  awx.example.org    /    awx-service:80 (10.1.8.135:8052)
Annotations:   nginx.ingress.kubernetes.io/rewrite-target: /$1
Events:
  Type  Reason  Age    From            Message
  ----  -----  --    --   -----
  Normal  CREATE  5s    nginx-ingress-controller  Ingress default/awx-ingress
  Normal  UPDATE  1s    nginx-ingress-controller  Ingress default/awx-ingress
jfreeman@masteryawx1:~$
```

Figure 5.3 – A successful deployment of the ingress configuration for AWX

The default username for logging into AWX is `admin`. However, the password is randomly generated and stored in a secret within Kubernetes. To retrieve this so you can log in for the first time, run the following command:

```
microk8s kubectl get secret awx-admin-password -o jsonpath='{.data.password}' | base64 --decode
```

Congratulations! You should now be able to log into your AWX deployment by pointing a web browser at the hostname you chose earlier. In this example, it would be `https://awx.example.org`.

On the first run of AWX, many operations, such as building the database schema, are performed in the background. As such, it might initially appear that the GUI is not responding. If your pod statuses look healthy, simply wait, and in a few minutes you will see the login screen appear, as shown in the following figure:

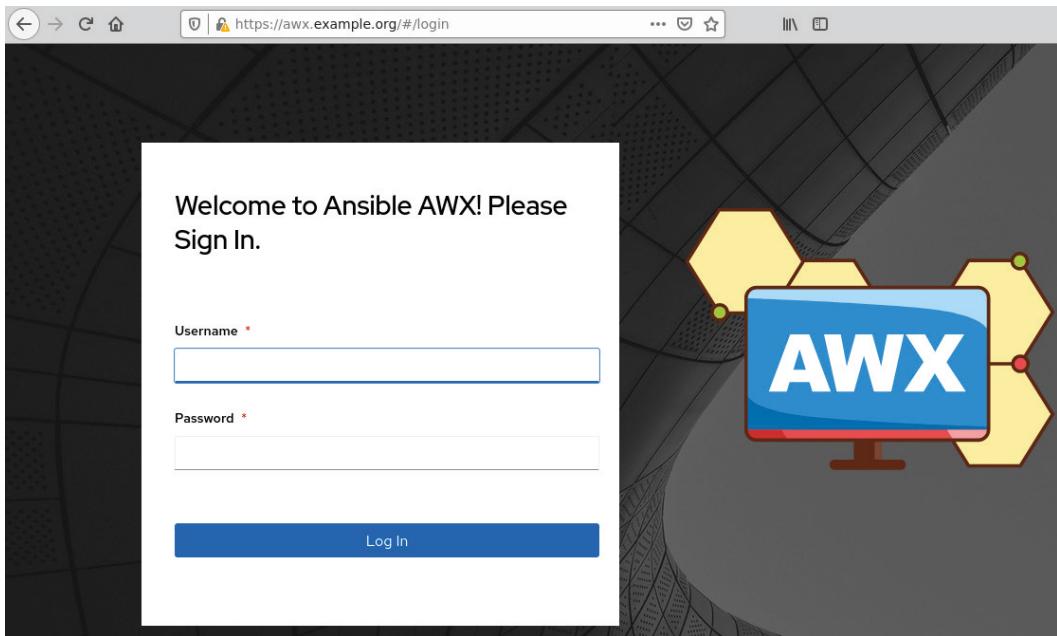


Figure 5.4 – Accessing the login screen of AWX after deployment

When you first log into AWX, you will be presented with a dashboard screen and a menu bar down the left-hand side. It is through this menu bar that we will explore AWX and perform our first configuration work. Equally, it is worth noting that when AWX is first installed, some example content is populated to help you get up to speed quicker. Feel free to explore the demo content, as the examples are different to those given in this book.

Before we complete this section, consider the persistent volume we created earlier for storing our local playbooks. How do we get access to that? When using a simple single-node deployment of `microk8s` as we have used here, you can execute a few commands to query the environment and find out where the files should go.

First off, retrieve the name of your `hostpath-provisioner` pod. It should look a little something like `hostpath-provisioner-5c65fbdb4f-jcq8b`, and can be retrieved using the following command:

```
microk8s kubectl get pods -A | awk '/hostpath/ {print $2}'
```

Having established this unique name, run the following command to discover the local directory where files are being stored for your pods. Be sure to replace the unique hostpath-provisioner name with the one from your system:

```
microk8s kubectl describe -n kube-system pod/hostpath-
provisioner-5c65fbdb4f-jcq8b | awk '/PV_DIR/ {print $2}'
```

Finally, retrieve the unique name of your persistent volume claim for your AWX playbooks, using the following command:

```
microk8s kubectl describe pvc/awx-pvc | awk '/Volume:/ {print
$2}'
```

Your final path will be an amalgamation of these results, including namespace (default in this example), and your PVC name (defined as awx-pvc in the my-awx-storage.yml file earlier). Thus, on my demo system, my local playbooks should be placed under the following directory:

```
/var/snap/microk8s/common/default-storage/default-awx-pvc-pvc-
52ea2e69-f3c7-4dd0-abcb-2a1370ca3ac6/
```

We will put some simple example playbooks into this directory later in the chapter, so it's worth locating it now and making a note of it so that you can access it easily for the later examples.

With AWX up and running on Microk8s, in the next section we will look at getting our first playbook integrated and running with AWX.

## Integrating AWX with your first playbook

There is a basic four-stage process involved in getting a playbook to run from AWX. Once you understand this, it paves the way for more advanced usage and fuller integration in an enterprise environment. In this part of the chapter, we will master these four stages in order to get to the point where we can run our first simple playbook, and this will give us the building blocks to move forward with AWX in confidence. The four stages are as follows:

1. Define a project.
2. Define an inventory.
3. Define credentials.
4. Define a template.

The first three stages can be performed in any order, but the template mentioned in the final stage pulls together the three previously created facets. Therefore, it must be defined last. Also, note that there does not need to be a one-to-one relationship between these items. Several templates can be created from one project. This is also the case for inventories and credentials.

Before we get started, we need a simple playbook that we can use in our examples as we go through this part of the chapter. On the AWX host, locate the local AWX persistent volume folder (this is described in the previous section if you are running AWX on Microk8s). I will show examples from my demo system in the following commands, but your system will have its own unique IDs. Make sure you adjust the paths for your system – copying and pasting the ones from mine will almost certainly not work!

Every locally hosted project must have its own subdirectory within the persistent volume, so let's create one here:

```
cd /var/snap/microk8s/common/default-storage/default-awx-pvc-
pvc-64aee7f5-a65d-493d-bdc1-2c33f7da8a4e
mkdir /var/lib/awx/projects/mastery
```

Now place the following example code into this folder, as `example.yaml`:

```
---
- name: AWX example playbook
  hosts: all
  gather_facts: false

  tasks:
    - name: Create temporary directory
      ansible.builtin.file:
        path: /tmp/mastery
        state: directory

    - name: Create a file with example text
      ansible.builtin.lineinfile:
        path: /tmp/mastery/mastery.txt
        line: 'Created with Ansible Mastery!'
        create: yes
```

With this done, we can proceed to defining a project.

## Defining a project

A project, in AWX terms, is simply a collection of Ansible playbooks grouped together. These collections of playbooks are often retrieved from a **source control management (SCM)** system. Indeed, this is the recommended way to host Ansible playbooks in an enterprise. Using an SCM means that everyone is working from the same version of code, and all changes are tracked. These are the elements that are vital in an enterprise environment.

With regards to the grouping of playbooks, there is no right or wrong way to organize projects, so this is very much up to the teams involved. Put simply, one project links to one repository, and so where it makes sense for multiple playbooks to live in one repository, it would make sense for them to live in one project within AWX. This is not a requirement, however – you can have just one playbook per project if it suits your needs best!

As previously discussed, it is also possible to store Ansible playbooks locally. This is useful when testing or when starting out, and we will utilize this capability in our example here, as it ensures everyone reading this book can complete the examples with ease.

Log into the AWX interface using the `admin` account and click on the **Projects** link on the left-hand menu bar. Then click on the **Add** button near the top right of the window. This creates a new blank project for us.

For now, we do not need to worry about all the fields (we'll discuss these in detail later). However, we do need to configure the following:

Field Name	Value	Notes
<b>Name</b>	Mastery Examples	A unique name to distinguish the project from the others.
<b>SCM Type</b>	Manual	Refers to the source of the playbook code – note the other options available in the drop-down list. Manual refers to playbooks on the persistent volume we created.
<b>Playbook Directory</b>	mastery	This is the name of the directory we defined earlier in the chapter and put our <code>example.yml</code> file in.

The end result should look something like the following figure:

The screenshot shows the AWX interface for creating a new project. At the top, there's a navigation bar with icons for projects, a search bar, and user authentication (admin). Below the header, the title 'Create New Project' is displayed. The form fields are as follows:

- Name \***: Mastery Examples
- Description**: (empty field)
- Organization \***: Default
- Default Execution Environment**: (empty field)
- Source Control Credential Type \***: Manual
- Type Details** section:
  - Project Base Path**: /var/lib/awx/projects
  - Playbook Directory** \*: mastery
- Buttons**: Save (blue button) and Cancel.

Figure 5.5 – Creating your first project in AWX using our local playbook directory

Click the **Save** button to store your edits. That's it – you have defined your first project in AWX! From here, we can define an inventory.

## Defining an inventory

Inventories in AWX work exactly the same as the inventories that we worked with in *Chapter 1, The System Architecture and Design of Ansible*, where we referenced them using the command line. They can be static or dynamic, can consist of groups and/or individual hosts, and can have variables defined on a global per group or per-host basis – we are now simply defining them through a user interface.

Click on the **Inventories** item on the left-hand menu bar. As with projects, we want to define something new, so click on the **Add** button near the top right of the window. A drop-down list will appear. Select **Add Inventory** from this list.

When the **Create new inventory** screen appears, enter a name for the inventory (for example, **Mastery Demo**), and then click the **Save** button.

**Important note**

You must save your blank inventory before you can start defining hosts or groups.

When this is completed, you should have a screen that looks something like that shown in the following figure:

Inventories

## Create new inventory

Name \*      Description      Organization \*

Mastery Demo

Insights Credential      Instance Groups

Variables ?      YAML      JSON

1 ---

Save      Cancel

Figure 5.6 – Creating a new empty inventory in AWX

Once you have saved the new inventory, note the tabs along the top of the inventories sub-pane – **Details**, **Access**, **Groups**, **Hosts**, **Sources**, and **Jobs**. You will find tabs like these on almost every pane in the AWX user interface – we also saw them after we defined our first project earlier in this chapter (we just didn't need to use them at that stage).

Keeping our example simple, we will define one host in a group to run our example playbook against. Click on the **Groups** tab, and then click on the **Add** button to add a new inventory group. Give the group a name and click **Save**, as shown in the following figure:

The screenshot shows the 'Create new group' dialog in the AWX interface. At the top, there's a breadcrumb navigation: Inventories > Mastery Demo > Groups. Below that is the title 'Create new group'. The main form has two fields: 'Name \*' with the value 'MasteryGroup' and 'Description' which is empty. Below these is a 'Variables' section with tabs for 'Variables', 'YAML' (which is selected), and 'JSON'. Under 'Variables', there's a table with one row labeled '1' and three columns with the value '---'. At the bottom of the dialog are 'Save' and 'Cancel' buttons.

Figure 5.7 – Creating a new inventory group in AWX

Now click on the **Hosts** tab, and then click on the **Add** button and select **Add new host** from the drop-down menu. Enter the IP address of your AWX host into the **Name** field (or the FQDN if you have set up DNS resolution). You can also add a description to the host if you wish, then click **Save**. The end result should look something like the following figure:

Inventories > Mastery Demo > Groups > Mastery Group > Hosts



## Create new host

Name \*

10.0.50.25

Description

AWX Mastery Host

Variables

YAML JSON

1	---
2	

Save Cancel

Figure 5.8 – Creating a new host in the Mastery Group group of the Mastery Demo inventory

### Important note

The **Variables** box seen on most of the inventory screens expects variables to be defined in YAML or JSON format, and not the INI format we used on the command line. Where earlier we had defined variables such as `ansible_ssh_user=james`, we would now enter `ansible_ssh_user:james` if the YAML mode is selected.

Well done! You've just created your first inventory in AWX. If we were to create this inventory on the command line, it would look like this:

```
[MasteryGroup]
10.0.50.25
```

It might be simple, but it paves the way for us to run our first playbook. Next, let's look at the concept of credentials in AWX.

## Defining credentials

One of the ways in which AWX lends itself to an enterprise is the secure storage of credentials. Ansible, given its nature and typical use cases, is often given the *keys to the kingdom* in the form of SSH keys or passwords that have root or other administrative-level privileges. Even if encrypted in a vault, the user running the playbook will have the encryption password and hence can obtain the credentials. Obviously, having many people with uncontrolled access to administrator credentials may not be desirable. Luckily for us, AWX solves this issue.

Let's take a simple example. Suppose my test host (the one that we defined the inventory for previously) has a `root` password of `Mastery123!`. How do we store this securely?

First of all, navigate to the **Credentials** menu item, and then click the **Add** button (as we have done previously) to create something new. Give the credential an appropriate name (for example, `Mastery Login`), and then click on the **Credential Type** dropdown to expand the list of available credential types (you can even create your own if you can't see the ones you need in here!).

You will see that there are many different credential types that AWX can store. For a machine login, such as ours, we want to select the Machine type. Once the credential type is set, you will see that the screen changes and fields appropriate to creating a machine credential have appeared. We could define the login based on the SSH key and various other parameters, but in our simple example, we will simply set the username and password to the appropriate values, as shown in the following figure:

Credentials

### Create New Credential

Name \*

Mastery Login

Description

Organization

Credential Type \*

Machine

Type Details

Username

root

Password

Mastery123!

Prompt on launch

SSH Private Key

Signed SSH Certificate

Figure 5.9 – Adding a new machine credential in AWX

Now, save the credential. If you now go back to edit the credential, you will note that the password disappears and is replaced by the string ENCRYPTED. It is now impossible to retrieve the password (or SSH key, or other sensitive data) through the AWX user interface directly. You will notice that you can replace the existing value (by clicking on the curly arrow to the left of the now grayed-out password field), but cannot see it. The only way to get the credential would be to get both connectivity to the backend database and the encryption key for the database that was used at the time of installation. This means even someone performing a SELECT operation on the database itself won't be able to see the key, as database rows containing sensitive data are all encrypted with a key that is autogenerated at install time. While this clearly has massive security benefits for an organization, it must also be pointed out that the loss of the backend database, or the encryption key associated with it, would result in a complete loss of the AWX configuration. As a result, it is important (as with any infrastructure deployment) to back up your AWX deployment and associated secrets, in case you need to recover from a potential disaster situation.

Nonetheless, AWX has protected your sensitive access data in a manner not totally dissimilar to Ansible Vault. Of course, Ansible Vault remains a command-line tool and, although vault data can be used in playbooks in AWX exactly as it can when Ansible is used on the command line, vault creation and modification remains a command line-only activity. With our credential in place, let's proceed to the final step necessary to run our first ever playbook from AWX – defining a template.

## Defining a template

A job template – to give it its full name – is a way of pulling together all the previously created configuration items, along with any other required parameters, to run a given playbook against an inventory. Think of it as defining how you would run `ansible-playbook` if you were on the command line.

Let's dive right in and create our template by carrying out the following steps:

1. Click on **Templates** in the left-hand menu.
2. Click on the **Add** button to create a new template.
3. Select **Add Job Template** from the drop-down list.

4. As a minimum to run our first job, you will need to define the following fields on the **Create New Job Template** screen:

Field name	Values	Notes
<b>Name</b>	Mastery Template	A unique name for the job template to identify it.
<b>Job Type</b>	Run	The default here is Run. This is exactly what we want to do. We could also select <b>Check</b> . This runs the playbook using the defined parameters, without making any changes on the inventory hosts. This is ideal for testing, but has limitations – see <i>Chapter 10, Extending Ansible</i> , for more details.
<b>Inventory</b>	Mastery Demo	Click on the magnifying glass icon in this field, and then select the inventory you created earlier in this process.
<b>Project</b>	Mastery Examples	Click on the magnifying glass icon in this field, and select the project we created earlier in this chapter, containing our example playbook.
<b>Playbook</b>	example.yaml	Once the <b>Project</b> field is populated, the <b>Playbook</b> drop-down menu is automatically populated with a list of all files found in the <b>Project</b> source with the *.yaml or *.yml extension. Note that if your project was linked to an SCM, this list will be blank until AWX first synchronizes with the SCM.
<b>Credential</b>	Mastery Login	Click on the magnifying glass icon in this field and select the credential we created earlier.

This should result in a screen that looks something like that shown in the following figure:

The screenshot shows the 'Create New Job Template' interface. The 'Name' field is populated with 'Mastery Template'. The 'Job Type' dropdown is set to 'Run'. Under 'Inventory', 'Mastery Demo' is selected. In the 'Project' section, 'Mastery Examples' is chosen. The 'Execution Environment' field is empty. The 'Playbook' dropdown contains 'example.yml'. The 'Credentials' section shows 'SSH: Mastery Login' selected. The 'Labels' section is empty. The 'Variables' section is expanded, showing a YAML key-value pair: '1: ---' and '2:'. There are tabs for 'YAML' and 'JSON', with 'YAML' currently selected.

Figure 5.10 – Creating a new template in AWX

With all the fields populated, as in the previous screenshot, click on the **Save** button. Congratulations! You are now ready to run your first playbook from AWX. To do so, navigate back to the list of templates and click on the small rocketship icon to the right of our newly created template. Immediately upon doing so, you will see the job execute and will see the output from `ansible-playbook` that we are familiar with from the command line, as shown in the following figure:

The screenshot shows the AWX interface for a job named 'Mastery Template'. The 'Output' tab is selected. At the top, there are tabs for 'Back to Jobs', 'Details', and 'Output'. Below the tabs, the job name 'Mastery Template' is displayed along with metrics: Plays 1, Tasks 2, Hosts 1, and Elapsed 00:00:07. There are also download and delete icons. A dropdown menu for 'Stdout' is open, showing the log output. The log output is as follows:

```
4 TASK [Create temporary directory] *****
5 [DEPRECATION WARNING]: Distribution Ubuntu 20.04 on host 10.0.50.25 should use
6 /usr/bin/python3, but is using /usr/bin/python for backward compatibility with
7 prior Ansible releases. A future Ansible release will default to using the
8 discovered platform python for this host. See https://docs.ansible.com/ansible/
9 2.11/reference_appendices/interpreter_discovery.html for more information. This
10 feature will be removed in version 2.12. Deprecation warnings can be disabled
11 by setting deprecation_warnings=False in ansible.cfg.
12 changed: [10.0.50.25]
13
14 TASK [Create a file with example text] *****
15 changed: [10.0.50.25]
16
17 PLAY RECAP *****
18 10.0.50.25 : ok=2    changed=2    unreachable=0    failed=0    skipped=0    rescued=0
   ignored=0
```

Figure 5.11 – The output from our first playbook template run in AWX

On this screen, you can see the raw output from `ansible-playbook`. You can access the **Jobs** screen any time by clicking on the **Jobs** menu item on the menu bar, and browsing all jobs that have been run. This is excellent for auditing the various activities that AWX has been orchestrating, especially in a large multi-user environment.

At the top of the **Jobs** screen, you can see the **Details** tab, where all the fundamental parameters we defined earlier are listed, such as **Project** and **Template**. Also displayed is useful information for auditing purposes, such as information regarding the user the job was launched by, and times that the job started and finished. A screenshot of this is shown in the following figure:

The screenshot shows the 'Details' tab of a job run in AWX. The job is named 'Mastery Template'. The 'Details' tab is selected, showing the following information:

Parameter	Value	Parameter	Value
Status	Successful	Started	5/4/2021, 9:15:14 PM
Finished	5/4/2021, 9:15:22 PM	Job Template	Mastery Template
Job Type	Playbook Run	Launched By	admin
Inventory	Mastery Demo	Project	Mastery Examples
Playbook	example.yml	Verbosity	0 (Normal)
Execution Environment	AWX EE 0.2.0	Container Group	tower
Job Slice	0/1		
Credentials	SSH: Mastery Login		
Created	5/4/2021, 9:15:13 PM by admin	Last Modified	5/4/2021, 9:15:14 PM

Below the table, there are two sections for 'Variables' and 'Artifacts', each with tabs for 'YAML' and 'JSON'. The 'Variables' section shows an empty JSON object: {}.

Figure 5.12 – The Details tab from our playbook template run

While AWX is capable of much more, these fundamental stages are central to most of the tasks you will want to perform in AWX. Therefore, gaining an understanding of their usage and sequence is essential in learning how to use AWX. Now that we have the fundamentals under our belts, in the next section we will take a look at some of the more advanced things you can do with AWX.

## Going beyond the basics

We have now covered the basics necessary to run your first playbook from AWX – the basics required for most Ansible automation within this environment. Of course, we can't possibly cover all the advanced features AWX has to offer in a single chapter. In this section, we will therefore highlight a few of the more advanced facets to explore if you wish to learn more about AWX.

### Role-based access control (RBAC)

So far, we have only looked at using AWX from the perspective of the built-in `admin` user. Of course, one of AWX's enterprise-level features is RBAC. This is achieved by the use of **users** and **teams**. A team is basically a group of users, and users can be a member of one or more teams.

Both users and teams can be created manually in the AWX user interface, or through integration with an external directory service, such as LDAP or Active Directory. In the case of directory integration, teams would most likely be mapped to groups within the directory, though rich configuration allows for administrators to define the exact nature of this behavior.

The RBAC's within AWX are rich. For example, a given user can be granted the `Admin` role within one team, and either the `Member` or `Read` roles in another.

User accounts themselves can be set up as system administrators, normal users, or system auditors.

In addition to this, as we stepped through the basic setup part of this chapter, you will have noticed the tabs on just about every page of the AWX user interface. Among these, there is almost always a tab called **Permissions**, which allows true fine-grained access control to be achieved.

For example, a given user of the **Normal User** type could be given the `Admin` role within their assigned team. However, they can then be assigned the `READ` role on a given project, and this more specific privilege supersedes the less specific `Admin` role set at the **Team** level. So, when they log in, they can see the project in question but can't change it or execute any tasks – for example, an update from the SCM.

**Important note**

As a general rule of thumb, more specific privileges supersede less specific ones. So, those at a project level will take precedence over those at a team or user level. Note that, for items where no permission is specified via either a user or their team, that person will not even see that item when logged into the user interface. The only exception to these rules are system administrators, who can see everything and perform any action. Assign this type to user accounts sparingly!

There is a great deal to explore when it comes to RBAC. Once you get the hang of it, it is easy to create secure and tightly locked-down deployments of AWX where everyone has just the right amount of access.

## Organizations

AWX contains a top-level configuration item called an **organization**. This is a collection of **inventories**, **projects**, **job templates**, and **teams** (these, in turn, are a grouping of **users**). Hence, if you have two distinct parts of an enterprise that have entirely different requirements but still require the use of AWX, they can share a single AWX instance without the need for overlapping configuration in the user interface by virtue of organizations.

While users of the system administrator type have access to all organizations, normal users will only see their associated organizations and configuration. These are a really powerful way of segregating access to the different parts of an enterprise deployment of AWX.

By way of example, when we created our inventory earlier in the chapter, you will have noticed that we ignored the **Organization** field (this was set to default – the only organization that exists on a new AWX install). If we were to create a new organization called `Mastery`, then anyone who was not a member of this organization would be unable to see this inventory, regardless of the permissions or privileges they have (the exception to this being the **System administrator** user type, which can see everything).

## Scheduling

Some AWX configuration items, such as projects (which may need to update from an SCM) or job templates (which perform a specific task), may need to be run on a regular basis. Having a powerful tool such as AWX, but then requiring operators to log in regularly to perform routine tasks, would be pointless. Therefore, AWX has built-in scheduling.

On the definition page for any project or template, simply look for the **Schedules** tab, and you then have a rich range of scheduling options available to you – *Figure 5.13* shows an example of the creation of a daily schedule, running every day from the 7th to the 11th of May 2021 at 1 pm in the London time zone. Note that this schedule is being created against the **Mastery Template** job template that we created earlier, and so will automatically run this playbook template on the defined schedule:

Templates > Mastery Template > Schedules

### Create New Schedule

Name \*  
Daily run

Description

Start date/time \*  
2021-05-07T13:00:00

Local time zone \*  
Europe/London

Run frequency \*  
Day

**Frequency Details**

Run every \*  
1 day

End \*

- Never
- After number of occurrences
- On date

Occurrences \*  
5

Save Cancel

Figure 5.13 – Creating a daily schedule to run the **Mastery Template** job template created earlier

Note the variety of options available to you for scheduling. To help you ensure that the schedule suits your requirements, a detailed breakdown of the schedule is shown when you save the new schedule. When you have schedules running unattended, along with multiple users logging into a system such as AWX, it is vital that you can maintain oversight of what is going on. Thankfully, AWX has rich features to allow auditing of the events that occur, and we will take a look at these in the next section.

## Auditing

One of the risks of running Ansible on the command line is that once a particular task has been run, its output is lost forever. It is, of course, possible to turn on logging for Ansible. However, in an enterprise, this would need to be enforced, which would be difficult with lots of operators having root access to a given Ansible machine, be it their own laptop or a server elsewhere. Thankfully, as we saw in our earlier example, AWX stores not only the details of who ran what tasks and when but also stores all the output from the `ansible-playbook` runs. In this way, compliance and auditability are achieved for enterprises wishing to use Ansible.

Simply navigate to the **Jobs** menu item, and a list of all previously run jobs (that the user has permission to see) will be shown. It is even possible to repeat previously completed jobs directly from this screen simply by clicking on the rocketship icon next to the job in question. Note that this immediately launches the job with the same parameters it was launched with last time, so be sure that clicking it is what you want to do!

*Figure 5.14* shows the job history for our demo AWX instance being used for the book:

Jobs						③
	Name	Status	Type	Start Time	Finish Time	Actions
»	<a href="#">6 – Mastery Template</a>	<span>Successful</span>	Playbook Run	5/6/2021, 11:54:00 AM	5/6/2021, 11:54:10 AM	↗
»	<a href="#">5 – Mastery Template</a>	<span>Successful</span>	Playbook Run	5/4/2021, 9:15:14 PM	5/4/2021, 9:15:22 PM	↗
»	<a href="#">4 – Mastery Template</a>	<span>Successful</span>	Playbook Run	5/4/2021, 5:29:01 PM	5/4/2021, 5:29:06 PM	↗
»	<a href="#">3 – ping</a>	<span>Successful</span>	Command	5/4/2021, 5:28:48 PM	5/4/2021, 5:28:53 PM	↗
»	<a href="#">2 – Mastery Template</a>	<span>Successful</span>	Playbook Run	5/4/2021, 5:27:30 PM	5/4/2021, 5:27:35 PM	↗
»	<a href="#">1 – ping</a>	<span>Failed</span>	Command	5/4/2021, 5:01:14 PM	5/4/2021, 5:01:23 PM	↗

Figure 5.14 – The job history pane of the AWX instance being used for the book

Clicking on the numbered entry in the **Name** column takes you to the **Output** and **Details** tab panes that we saw in *Figure 5.11* and *Figure 5.12*, but of course, relevant to the specific job run you clicked on. While you can clean up the job history, the jobs remain there for you to examine until you delete them. Also note the two grayed-out buttons at the top of *Figure 5.14*. Using these, you can cancel running jobs (useful if for any reason they get stuck or fail) and also delete multiple entries from the job history. This is great for cleaning up once you have finished your auditing.

Of course, with playbooks, there is no one-size-fits-all solution, and sometimes we need operators to be able to input unique data at the time of running playbooks. AWX provides a feature for exactly this purpose called surveys, and we will look at this in the next section.

## Surveys

Sometimes, when launching a job template, it is not possible (or desirable) to define all information upfront. While it is perfectly possible to define parameters using variables in the AWX user interface, this is not always desirable, or indeed user friendly, as the variables must be specified in valid JSON or YAML syntax. In addition, users who have only been granted the `Read` role on a template will not be able to edit that template definition – this includes the variables! However, there might be a valid reason for them to set a variable, even though they shouldn't be editing the template itself.

Surveys provide the answer to this, and on any job template you have created, you will find a tab at the top marked **Survey**. A survey is essentially a questionnaire (hence the name!) defined by an administrator that asks for input in a user-friendly manner and where simple user input validation is performed. Once validated, the entered values are stored in Ansible variables, just as they would be if they had been defined in YAML or JSON format.

For example, if we wanted to capture the `http_port` variable value for a job template when it is run, we could create a survey question, as shown in *Figure 5.15*:

Templates > Mastery Template > Survey

### Add Question

[Back to Templates](#) Details Access Notifications Schedules Jobs Survey

<b>Question *</b>	<b>Description</b>		<b>Answer variable name *</b> <a href="#">?</a>
Please enter the HTTP port number to use	'TP port value - integer between 1 and 65535		http_port
<b>Answer type *</b> <a href="#">?</a>	<input checked="" type="checkbox"/> Required		
Integer			
<b>Minimum</b>	<b>Maximum</b>	<b>Default answer</b>	
1	65535	80	

**Save** **Cancel**

Figure 5.15 – Creating a survey question to capture a valid HTTP port number into a variable  
Once you have created all your questions, note that you need to turn surveys on for your job template, as shown in *Figure 5.16*, otherwise the questions will not appear when it is run:

Templates > Mastery Template

### Survey

[Back to Templates](#) Details Access Notifications Schedules Jobs Survey

On **Add** **Delete**

- Please enter the HTTP port number to use \*

Type integer Default 80

Figure 5.16 – Turning on surveys for a job template

Now, when the playbook is run, the user is prompted to enter a value, and AWX ensures it is an integer in the specified range. A sensible default is also defined. Let's now move forward to looking at a more advanced way of using job templates in AWX, called workflows.

## Workflow templates

Playbook runs, especially from AWX, can be complex. For example, it might be desirable to update a project from an SCM system and any dynamic inventories first. We might then run a job template to roll out some updated code. If this fails, however, it would almost certainly be desirable to roll back any changes that were made (or take other remedial action). When you click on the now-familiar **Add** button to add a new template, you will see two options in the drop-down menu – **Job template** (we have already used this), and **Workflow template**.

Once all the required fields are filled in for the new workflow template and it is saved, you will automatically enter **Workflow Visualizer** (to get back to this in the future, simply access your workflow template through the GUI in the normal manner, and then click on the **Visualizer** tab). The workflow visualizer builds up a flow, from left to right, of tasks for AWX to perform. For example, the following screenshot shows a workflow where our demo project is initially synchronized with its SCM.

If that step succeeds (denoted by the green link to the next block), the demo job template is run. If that in turn succeeds, then the mastery template is run. If any of the preceding steps fail, then the workflow stops there (though an **On Failure** action can be defined at any stage). Based on this simple building block premise and the ability to perform subsequent actions in the event of success, failure, or always, will enable you to build large-scale operational flows within AWX. This will all be achieved without having to build up huge monolithic playbooks. *Figure 5.17* shows our simple workflow in the visualizer:

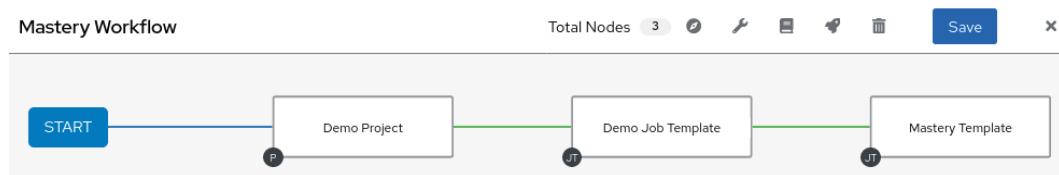


Figure 5.17 – The workflow visualizer in AWX

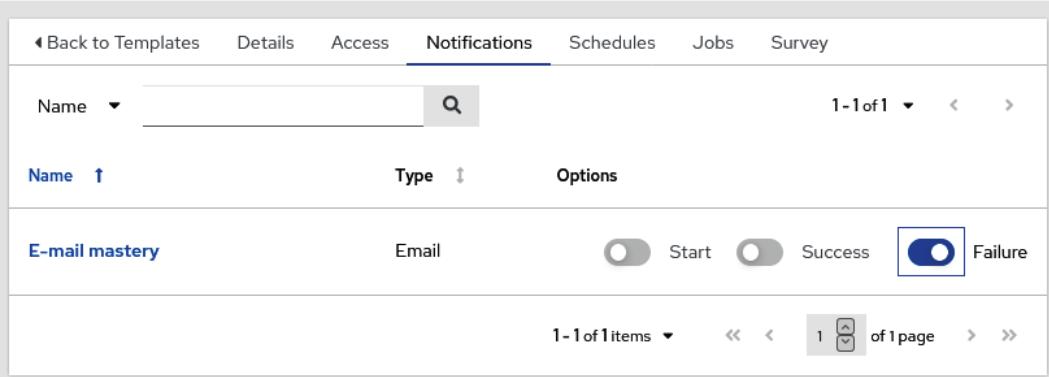
Using this tool, we can powerfully build up multi-step workflows, taking intelligent action after each stage, depending on whether it succeeded or not.

Everything we have discussed so far is great if you are interacting directly with the AWX GUI. However, what happens if you have set up unattended actions to run, but wish to be notified about their results (especially if they fail)? Equally, how can you notify a team if someone runs a potentially service-impacting change? You'll find the answers to these questions in the next section.

## Notifications

As you examined the AWX user interface, you will have noticed that most screens have a tab called **Notifications**. AWX has the ability to integrate with many popular communication platforms, such as Slack, IRC, Pagerduty, and even good old-fashioned email (this list is not exhaustive). Once the configuration for a given platform is defined through the user interface, notifications can then be sent when specific events occur. These events will vary according to the item you are wishing to generate notifications from. For example, with job templates, you can choose to be notified when the job starts, when it succeeds, and/or when it fails (and any combination of those events). You can generate different notification types for different events. For example, you could notify a Slack channel of a template being started, but email your ticketing system if the template fails to automatically generate a ticket to prompt further investigation.

For example, *Figure 5.18* shows our previously configured Mastery Template set up to email a given recipient list in the event that its execution fails. On start, and on success, no notification is given (though this can be turned on, of course):



The screenshot shows the AWX interface for managing a 'Mastery Template'. The top navigation bar includes 'Templates > Mastery Template' and a gear icon. Below this, the 'Notifications' tab is selected, indicated by a blue underline. The main content area displays a table of notifications. The table has columns for 'Name', 'Type', and 'Options'. One row is visible, showing 'E-mail mastery' under 'Name', 'Email' under 'Type', and three toggle switches under 'Options': 'Start' (off), 'Success' (off), and 'Failure' (on, highlighted with a blue border). Navigation controls at the bottom show '1-1 of 1 items' and a single-page indicator.

Figure 5.18 – Setting up email notifications for failed runs of Mastery Template

All notifications defined in AWX appear in the **Notifications** tab. However, they do not have to be added once defined. It is simply up to the user to turn the **Start**, **Success**, and **Failure** notifications on or off for each notification service.

There is one more way to interact with AWX without using the GUI. This, of course, is through the API, which we'll look at in the final part of this chapter.

## Using the API

Throughout this chapter of the book, we have looked at all AWX operations using the GUI, as this is probably the easiest and most visual way to explain their functions and usage. However, one of the key features of AWX for any enterprise is the API, which is a complete feature that enables us to perform all of the operations completed here (and more) without having to touch the UI.

This is an incredibly powerful tool, especially with regards to integration into larger workflows. For example, you could hook AWX into your CI/CD pipeline using the API, and upon a successful build of your code, you could trigger an AWX job to deploy a test environment to run it in (and even deploy the code to that environment). Similarly, you can automatically create job templates, inventory items, and all other aspects of the configuration through the API.

The API itself is browsable, and you can access it by adding `/api` or `/api/v2` to the URL of your AWX server (for version 1 and version 2 of the API respectively).

Although normally you would integrate these into a larger application or workflow, it is easy to demonstrate the API usage with `curl`. For example, suppose we want to retrieve a list of the inventories defined in our AWX server. We can do that with a command like the following:

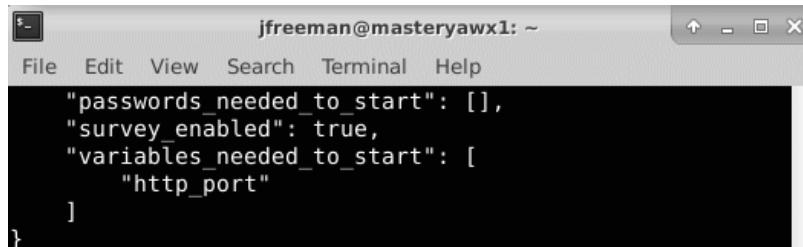
```
curl -k -s --user admin:adminpassword -X GET https://awx.  
example.org/api/v2/inventories/ | python -m json.tool
```

Naturally, you will need to substitute your credentials into the `--user` parameter and the correct FQDN for your AWX server into the URL in the command. Once done, this command will retrieve details of all the inventories defined in AWX in JSON format – you don't need to pipe this through Python's `json.tool` tool – it just makes the output more readable for a human!

Similarly, we could launch our mastery example template through the API. All configuration elements of AWX have a unique numeric ID associated with them that we must use to access them. Thus, for example, let's retrieve the list of job templates from AWX using the API:

```
curl -k -s --user admin:adminpassword -X GET https://awx.  
example.org/api/v2/job_templates/ | python -m json.tool
```

Looking through the JSON output, I can see that our Mastery Template has an id of 12 on my system. Also, because I set up a survey on this template for one of the earlier examples in this chapter, the JSON output is telling me that I need to specify some variables before the template can be launched. There are a number of items in the output of the GET query that might need to be set before a playbook can be launched, so it is worth reviewing them carefully before putting your API POST together. *Figure 5.19* shows the output from the API GET call, displaying the variables that must be set before the template can be launched:



A screenshot of a terminal window titled "jfreeman@masteryawx1: ~". The window contains a command-line interface with a menu bar (File, Edit, View, Search, Terminal, Help) and a title bar. The main area displays the following JSON code:

```
"passwords_needed_to_start": [],
"survey_enabled": true,
"variables_needed_to_start": [
    "http_port"
]
}
```

Figure 5.19 – Partial output from the API GET call on job template 12

This variable data can be specified using the extra\_vars data field in the API, so we can craft an API call like the following to launch the job:

```
curl -k -s --user admin:adminpassword -X POST -H 'Content-Type:application/json' https://awx.example.org/api/v2/job_templates/12/launch/ --data '{"extra_vars": "{\"http_port\": 80}"}' | python -m json.tool
```

The output from this command will include use details such as the job ID so that we can query the job run if we wish to. In my example, the job ID returned was 10, so I can query the status of this job (including whether it was successful or not) with the following:

```
curl -k -s --user admin:adminpassword -X GET https://awx.example.org/api/v2/jobs/10/ | python -m json.tool
```

You can even retrieve the output of the ansible-playbook command from the job run, using an API call like the following:

```
curl -k -s --user admin:adminpassword -X GET https://awx.example.org/api/v2/jobs/10/stdout/
```

Although you are unlikely to be driving the API using `curl` in a production environment, it is hoped that these simple, repeatable examples will help to get you started on your journey of AWX integration using the API.

There is even a CLI available for AWX that can be installed through Python's `pip` packaging system. This CLI uses a naming and command structure that is consistent with the HTTP-based API we have discussed in this section, and given the similarity, this is therefore left as an optional exercise. However, to get you started, the official documentation for the AWX CLI can be found here:

<https://docs.ansible.com/ansible-tower/latest/html/towercli/index.html>

Although the documentation mentions Ansible Tower, it is just as valid when used with the open-source AWX software.

## Summary

That concludes our whistle-stop tour of AWX. In this chapter, we showed that AWX is straightforward to install and configure once you know the core four-step process involved. We also showed how to build on this process with features such as surveys, notifications, and workflows.

You learned that AWX is straightforward to install (in fact, it installs with Ansible!), and how to add SSL encryption to it. You then gained an understanding of how the platform works, and how to go from a fresh install to building out projects, inventories, credentials, and templates to run Ansible jobs. You learned that there are many additional features that build on this. These were covered in the final part of this chapter in order to help you build a robust enterprise management system for Ansible.

In the next chapter, we will return to the Ansible language and look at the benefits of the Jinja2 templating system.

## Questions

1. AWX runs either in standalone Docker containers or Kubernetes.
  - a) True
  - b) False

2. AWX provides which of the following to enterprises looking to manage their automation processes?
  - a) A web UI
  - b) A feature-complete API
  - c) Source control integration
  - d) All of the above
3. AWX directly supports the secure management of credentials for automation.
  - a) True
  - b) False
4. AWX provides a graphical development environment for creating and testing Ansible playbooks.
  - a) True
  - b) False
5. AWX can schedule unattended jobs to run.
  - a) True
  - b) False
6. In AWX, the pre-configured parameter set for an `ansible-playbook` run is known as what?
  - a) Job Configuration
  - b) Ansible Template
  - c) Job Template
  - d) Ansible Run
7. AWX can have its configuration divided between different parts of a business through the creation of which of the following?
  - a) Teams
  - b) Organizations
  - c) Deploying a second AWX server
  - d) Groups

8. In AWX, it is possible to tell which of the following?
  - a) When a playbook was run
  - b) Who ran the playbook
  - c) What parameters were passed to the playbook
  - d) All of the above
9. User-friendly variable definition in AWX is provided via which feature?
  - a) Forms
  - b) e-Forms
  - c) extra vars
  - d) Surveys
10. Projects in AWX are made up of what?
  - a) Logical teams of users
  - b) Logical folders of playbooks
  - c) A task management system
  - d) Logical collections of roles



# Section 2: Writing and Troubleshooting Ansible Playbooks

In this section, you will gain a solid understanding of how to write robust, versatile playbooks, suitable for use in a wide variety of use cases and environments.

The following chapters are included in this section:

- *Chapter 6, Unlocking the Power of Jinja2 Templates*
- *Chapter 7, Controlling Task Conditions*
- *Chapter 8, Composing Reusable Ansible Content with Roles*
- *Chapter 9, Troubleshooting Ansible*
- *Chapter 10, Extending Ansible*



# 6

# Unlocking the Power of Jinja2 Templates

Manipulating configuration files by hand is a tedious and error-prone task. Equally, performing pattern matching to make changes to existing files is risky, and ensuring that the patterns are reliable and accurate can be time-consuming. Whether you are using Ansible to define configuration file content, perform variable substitution in tasks, evaluate conditional statements, or beyond, templating comes into play with nearly every Ansible playbook. In fact, given the importance of this task, it could be said that templating is the lifeblood of Ansible.

The templating engine employed by Ansible is Jinja2, which is a modern and designer-friendly templating language for Python. Jinja2 deserves its own book; however, in this chapter, we will cover some of the more common usage patterns of Jinja2 templating in Ansible to demonstrate the power it can bring to your playbooks. In this chapter, we will cover the following topics:

- Control structures
- Data manipulation
- Comparing values

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running Ansible 4.3 or newer. Almost any flavor of Linux should do; for those interested in specifics, all the code presented in this chapter was tested on Ubuntu Server 20.04 LTS unless stated otherwise, and on Ansible 4.3. The example code that accompanies this chapter can be downloaded from GitHub at <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter06>.

Check out the following video to view the Code in Action: <https://bit.ly/3lZHTM1>

## Control structures

In Jinja2, a control structure refers to the statements in a template that control the flow of the engine parsing the template. These structures include conditionals, loops, and macros. Within Jinja2 (assuming the defaults are in use), a control structure will appear inside blocks of { % ... % }. These opening and closing blocks alert the Jinja2 parser that a control statement has been provided instead of a normal string or variable name.

## Conditionals

A conditional within a template creates a decision path. The engine will consider the conditional and choose from two or more potential blocks of code. There is always a minimum of two: a path if the conditional is met (evaluated as `true`), and either an explicitly defined `else` path if the conditional is not met (evaluated as `false`) or, alternatively, an implied `else` path consisting of an empty block.

The statement for a conditional is the `if` statement. This statement works in the same way as it does in Python. An `if` statement can be combined with one or more optional `elif` statements and an optional final `else`, and, unlike Python, it requires an explicit `endif`. The following example shows a config file template snippet that combines both regular variable replacement and an `if else` structure:

```
setting = {{ setting }}
{% if feature.enabled %}
feature = True
{% else %}
feature = False
{% endif %}
another_setting = {{ another_setting }}
```

In this example, we check the `feature.enabled` variable to see whether it exists and that it has not been set to `False`. If this is `True`, then the `feature = True` text is used; otherwise, the `feature = False` text is used. Outside of this control block, the parser performs a normal variable substitution for the variables inside the curly braces. Multiple paths can be defined by using an `elif` statement, which presents the parser with another test to perform should the previous tests equate to `False`.

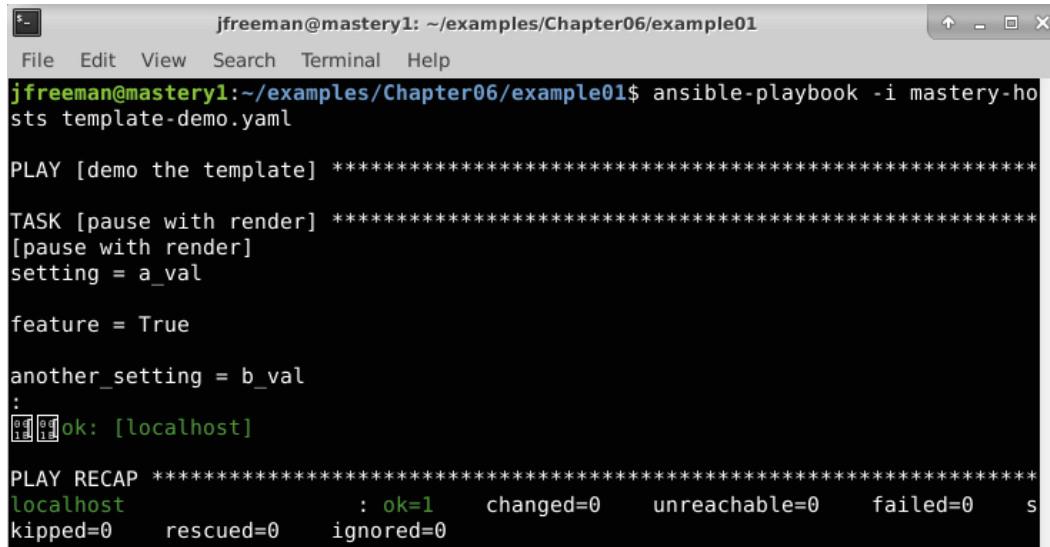
To demonstrate the rendering of the template and variable substitution, we'll save the example template as `demo.j2`. Then, we'll create a playbook, named `template-demo.yaml`, that defines the variables in use and then uses a `template` lookup as part of a `ansible.builtin.pause` task to display the rendered template on the screen:

```
---
- name: demo the template
  hosts: localhost
  gather_facts: false
  vars:
    setting: a_val
    feature:
      enabled: true
    another_setting: b_val
  tasks:
    - name: pause with render
      ansible.builtin.pause:
        prompt: "{{ lookup('template', 'demo.j2') }}"
```

Executing this playbook will display the rendered template on the screen while waiting for the input. You can use the following command to execute it:

```
ansible-playbook -i mastery-hosts template-demo.yaml
```

Simply press *Enter* to run the playbook, as shown in *Figure 6.1*:



The screenshot shows a terminal window titled 'jfreeman@mastery1: ~/examples/Chapter06/example01'. The command 'ansible-playbook -i mastery-hosts template-demo.yaml' is run. The output shows a single task being executed on localhost, which prints 'ok: [localhost]'. The final recap summary indicates 1 ok, 0 changed, 0 unreachable, 0 failed, 0 skipped, 0 rescued, and 0 ignored.

```
jfreeman@mastery1:~/examples/Chapter06/example01$ ansible-playbook -i mastery-hosts template-demo.yaml

PLAY [demo the template] ****
TASK [pause with render] ****
[pause with render]
setting = a_val

feature = True

another_setting = b_val
:
ok: [localhost]

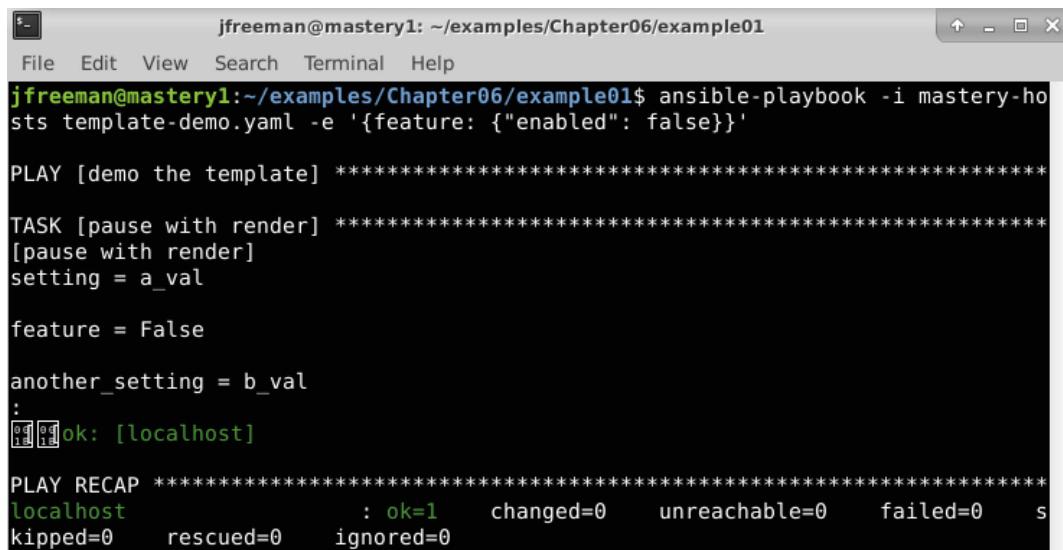
PLAY RECAP ****
localhost : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
```

Figure 6.1 – Rendering a simple template with conditionals using Ansible

Remembering the variable order of precedence for Ansible that we discussed in *Chapter 1, The System Architecture and Design of Ansible 3*, we can override the value of `feature.enabled` to `False`. We can do this by using the `--extra-vars` (or `-e`) parameter when running the playbook; this is because extra variables have higher priority than playbook-defined variables. You can achieve this by running the playbook again, but this time, using the following command:

```
ansible-playbook -i mastery-hosts template-demo.yaml -e
'{feature: {"enabled": false}}'
```

In this instance, the output should be slightly different, as shown in *Figure 6.2*:



```
jfreeman@mastery1: ~/examples/Chapter06/example01
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter06/example01$ ansible-playbook -i mastery-hosts template-demo.yaml -e '{feature: {"enabled": false}}'

PLAY [demo the template] ****
TASK [pause with render] ****
[pause with render]
setting = a_val

feature = False

another_setting = b_val
:
[0][0] ok: [localhost]
[1][1]

PLAY RECAP ****
localhost : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
```

Figure 6.2 – Rendering a simple template with conditionals using Ansible while overriding a variable value

As you can see from these simple tests, Jinja2 provides a very simple yet powerful way of defining data through conditionals in a template.

## Inline conditionals

Note that `if` statements can be used inside inline expressions. This can be useful in some scenarios where additional new lines are not desired. Let's construct a scenario where we need to define an API as either `cinder` or `cinderv2`, as shown in the following code:

```
API = cinder{{ 'v2' if api.v2 else '' }}
```

This example assumes that `api.v2` is defined as Boolean True or False.

Inline `if` expressions follow the syntax of `<do something> if <conditional is true> else <do something else>`. In an inline `if` expression, there is an implied `else`; however, that implied `else` is meant to be evaluated as an undefined object, which will normally create an error. We can protect against this by defining an explicit `else`, which renders a zero-length string.

Let's modify our playbook to demonstrate an inline conditional. This time, we'll use the debug module to render the simple template, as follows:

```
---  
- name: demo the template  
  hosts: localhost  
  gather_facts: false  
  vars:  
    api:  
      v2: true  
  tasks:  
    - name: pause with render  
      ansible.builtin.debug:  
        msg: "API = cinder{{ 'v2' if api.v2 else '' }}"
```

Notice that, this time, we are not defining an external template file; the template is actually in line with the Ansible tasks. Execute the playbook using the following command:

```
ansible-playbook -i mastery-hosts template-demo-v2.yaml
```

The output should look similar to the one shown in *Figure 6.3*:



A screenshot of a terminal window titled 'jfreeman@mastery1: ~/examples/Chapter06/example02'. The window shows the command 'ansible-playbook -i mastery-hosts template-demo-v2.yaml' being run. The output displays the playbook execution details, including tasks, results, and recap.

```
jfreeman@mastery1:~/examples/Chapter06/example02$ ansible-playbook -i mastery-hosts template-demo-v2.yaml  
  
PLAY [demo the template] ****  
  
TASK [pause with render] ****  
ok: [localhost] => {  
  "msg": "API = cinderv2"  
}  
  
PLAY RECAP ****  
localhost : ok=1    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 6.3 – Running a playbook with an inline template

Now, just as we did in our earlier example, we'll use Ansible's extra variables to change the value of `api.v2` to `false` to see the effect this has on the rendering of the inline template. Execute the playbook again using the following command:

```
ansible-playbook -i mastery-hosts template-demo-v2.yaml -e
'{"api": {"v2": false}}'
```

This time, the output should look similar to the one shown in *Figure 6.4*. Pay attention to how the rendered string has changed:

```
jfreeman@mastery1:~/examples/Chapter06/example02$ ansible-playbook -i mastery-hosts template-demo-v2.yaml -e '{"api": {"v2": false}}'

PLAY [demo the template] ****
TASK [pause with render] ****
ok: [localhost] => {
    "msg": "API = cinder"
}

PLAY RECAP ****
localhost                  : ok=1      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 6.4 – Running a playbook with an inline template while changing the behavior with extra variables

In this way, we can create very concise and powerful code that defines values based on an Ansible variable, just as we have demonstrated here.

## Loops

A loop allows you to build dynamically created sections in template files. It is useful when you know you need to operate on an unknown number of items in the same way. To start a loop control structure, we use the `for` statement. Let's demonstrate a simple way to loop over a list of directories where a fictional service might find data:

```
# data dirs
{% for dir in data_dirs -%}
data_dir = {{ dir }}
{% endfor -%}
```

**Tip**

By default, the { % } blocks print an empty line when the template is rendered. This might not be desirable in our output, but luckily, we can trim this by ending the block with -%} instead. Please refer to the official Jinja2 documentation at <https://jinja.palletsprojects.com/en/3.0.x/templates/#whitespace-control> for more details.

In this example, we will get one `data_dir =` line per item within the `data_dirs` variable, assuming that `data_dirs` is a list with at least one item in it. If the variable is not a list (or another iterable type) or is not defined, an error will be generated. If the variable is an iterable type but is empty, then no lines will be generated. Jinja2 can handle this scenario and also allows substituting in a line when no items are found in the variable via an `else` statement. In the following example, let's assume that `data_dirs` is an empty list:

```
# data dirs
{% for dir in data_dirs -%}
data_dir = {{ dir }}
{% else -%}
# no data dirs found
{% endfor -%}
```

We can test this by modifying our playbook and template file again. We'll create a template file called `demo-for.j2` with the template content listed earlier. Additionally, we will create a subtle variation on the playbook we used in our first example on conditionals to render the template and then pause for user input. The playbook file with the following code should be named `template-demo-for.yaml`:

```
- name: demo the template
hosts: localhost
gather_facts: false
vars:
  data_dirs: []
tasks:
  - name: pause with render
    ansible.builtin.pause:
      prompt: "{{ lookup('template', 'demo-for.j2') }}"
```

Once you have created these two files, you can then run the playbook using the following command:

```
ansible-playbook -i mastery-hosts template-demo-for.yaml
```

Running our playbook will render the template and produce an output that is similar to the one shown in *Figure 6.5*:

```
jfreeman@mastery1:~/examples/Chapter06/example03$ ansible-playbook -i mastery-hosts template-demo-for.yaml

PLAY [demo the template] ****
TASK [pause with render] ****
[pause with render]
# data dirs
# no data dirs found

:
ok: [localhost]

PLAY RECAP ****
localhost          : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0      rescued=0   ignored=0
```

Figure 6.5 – Rendering a template with a for loop in Ansible

As you can see, the `else` statement in the `for` loop handled the empty `data_dirs` list gracefully, which is exactly what we want in a playbook run.

## Filtering loop items

Loops can be combined with conditionals, too. Within the loop structure, an `if` statement can be used to check a condition using the current loop item as part of the conditional. Let's extend our example and prevent a user of this template from accidentally using `/` as a `data_dir` (any actions performed on the root directory of a filesystem can be dangerous, especially if they're performed recursively):

```
# data dirs
{% for dir in data_dirs -%}
{% if dir != "/" -%}
data_dir = {{ dir }}
{% endif -%}
{% else -%}
# no data dirs found
{% endfor -%}
```

The preceding example successfully filters out any `data_dirs` item that is `/`, but it requires far more typing than should be necessary. Jinja2 provides a convenient way that allows you to filter loop items easily as part of the `for` statement. Let's repeat the previous example using this convenience:

```
# data dirs
{% for dir in data_dirs if dir != "/" -%}
data_dir = {{ dir }}
{% else -%}
# no data dirs found
{% endfor -%}
```

So, not only does this structure require less typing, but it also correctly counts the loops, which we'll learn about in the next section.

## Loop indexing

Loop counting is provided for free, yielding an index of the current iteration of the loop. As variables, they can be accessed in a few different ways. The following table outlines the ways they can be referenced:

Variable	Description
<code>loop.index</code>	The current iteration of the loop (1 indexed)
<code>loop.index0</code>	The current iteration of the loop (0 indexed)
<code>loop.revindex</code>	The number of iterations until the end of the loop (1 indexed)
<code>loop.revindex0</code>	The number of iterations until the end of the loop (0 indexed)
<code>loop.first</code>	Boolean True if it is the first iteration
<code>loop.last</code>	Boolean True if it is the last iteration
<code>loop.length</code>	The number of items in the sequence

Having information related to the position inside the loop can help with the logic around what content to render. Considering our previous examples, rather than rendering multiple lines of `data_dir` to express each data directory, instead, we could provide a single line with comma-separated values. Without having access to loop iteration data, this would be difficult. However, by using this data, it can be straightforward. For the sake of simplicity, this example assumes a trailing comma after the last item is allowed, and that any white space (newlines) between items is also allowed:

```
# data dirs
{% for dir in data_dirs if dir != "/" -%}
  {% if loop.first -%}
```

```

data_dir = {{ dir }},
    {% else -%}
        {{ dir }},
    {% endif -%}
    {% else -%}
# no data dirs found
    {% endfor -%}

```

The preceding example made use of the `loop.first` variable to determine whether it needed to render the `data_dir =` part, or if it just needed to render the appropriately spaced padded directory. By using a filter in the `for` statement, we get a correct value for `loop.first`, even if the first item in `data_dirs` is an undesired `/`.

#### Important note

Take a look at the indentation on the first `else` statement – why did we do that? The answer is all to do with white space control in Jinja2 when it is rendered. Put simply, if you don't indent the control statement (for example, an `if` or `else` statement) that precedes the template content you wish to render, the template content will have all the white space to the left trimmed out; therefore, our subsequent directory entries would not be indented at all. Indentation is vitally important in some files (including YAML and Python!), and so, this is a small but vitally important nuance.

To test this, we'll create a new template file, named `demo-for.j2`, with the contents listed earlier. Additionally, we'll modify `template-demo-for.yaml` to define some `data_dirs`, including one of the `/`, which should be filtered out:

```

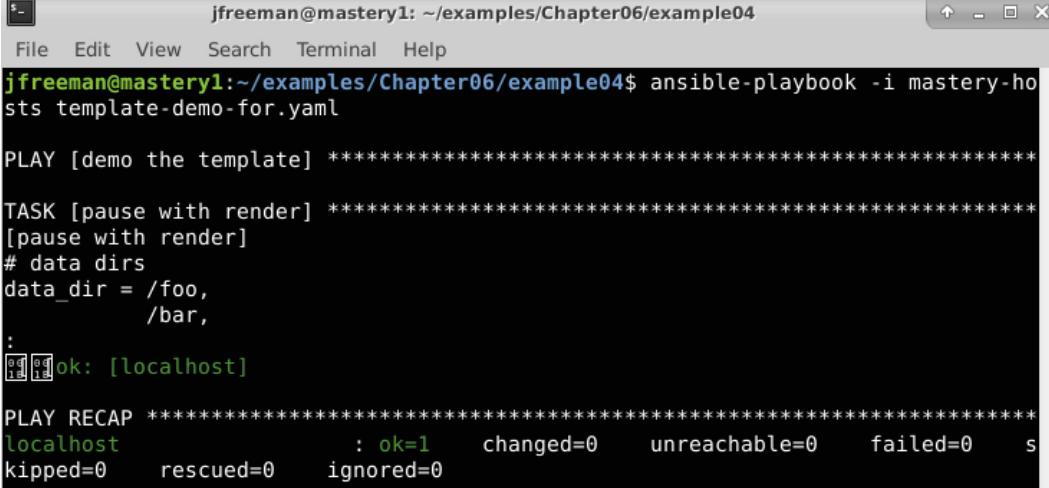
---
- name: demo the template
  hosts: localhost
  gather_facts: false
  vars:
    data_dirs: [ '/', '/foo', '/bar' ]
  tasks:
    - name: pause with render
      ansible.builtin.pause:
        prompt: "{{ lookup('template', 'demo-for.j2') }}"

```

Now, we can execute the playbook using the following command:

```
ansible-playbook -i mastery-hosts template-demo-for.yaml
```

When it runs, we should see our rendered content, as shown in *Figure 6.6*:



```
jfreeman@mastery1: ~/examples/Chapter06/example04$ ansible-playbook -i mastery-hosts template-demo-for.yaml

PLAY [demo the template] ****
TASK [pause with render] ****
[pause with render]
# data dirs
data_dir = /foo,
            /bar,
:
[0]: [0]: ok: [localhost]

PLAY RECAP ****
localhost                  : ok=1      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 6.6 – Rendering a template using a for loop in Ansible while making use of loop indexing

If, in the preceding example, trailing commas were not allowed, we could utilize inline if statements to determine whether we're done with the loop and render the commas correctly. You can view this in the following enhancement to the preceding template code:

```
# data dirs.

{% for dir in data_dirs if dir != "/" -%}
{% if loop.first -%}
data_dir = {{ dir }}{{ ',' if not loop.last else '' }}
            {% else -%}
            {{ dir }}{{ ',' if not loop.last else '' }}
{% endif -%}
{% else -%}
# no data dirs found
{% endfor -%}
```

Using inline `if` statements allow us to construct a template that will only render a comma if there are more items in the loop that passed our initial filter. Once more, we'll update `demo-for.j2` with the earlier content and execute the playbook using the following command:

```
ansible-playbook -i mastery-hosts template-demo-for.yaml
```

The output of the rendered template should look similar to the one shown in *Figure 6.7*:

```
jfreeman@mastery1:~/examples/Chapter06/example05$ ansible-playbook -i mastery-hosts template-demo-for.yaml

PLAY [demo the template] ****
TASK [pause with render] ****
[pause with render]
# data dirs.
data_dir = /foo,
           /bar
;
[1]: ok: [localhost]

PLAY RECAP ****
localhost          : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
```

Figure 6.7 – Rendering a template with a for loop in Ansible, making extended use of loop indexing  
The output is pretty much the same as before. However, this time, our template evaluates whether to place a comma after each value of `dir` in the loop using the inline `if` statement, removing the stray comma at the end of the final value.

## Macros

An astute reader will have noticed that, in the previous example, we had some repeated code. Repeating code is the enemy of any developer, and thankfully, Jinja2 has a way to help! A macro is like a function in a regular programming language: it's a way to define a reusable idiom. A macro is defined inside a `{% macro ... %} ... {% endmacro %}` block. It has a name and can take zero or more arguments. Code within a macro does not inherit the namespace of the block that is calling the macro, so all arguments must be explicitly passed in. Macros are called within curly brace blocks by name, and with zero or more arguments passed in via parentheses. Let's create a simple macro named `comma` to take the place of our repeating code:

```
{% macro comma(loop) -%}
{{ ',' if not loop.last else '' }}{%- endmacro %}
```

```
{%- endmacro -%}
# data dirs.
{% for dir in data_dirs if dir != "/" -%}
{% if loop.first -%}
data_dir = {{ dir }}{{ comma(loop) }}
{% else -%}
{{ dir }}{{ comma(loop) }}
{% endif -%}
{% else -%}
# no data dirs found
{% endfor -%}
```

Calling `comma` and passing it in the `loop` object allows the macro to examine the loop and decide whether a comma should be omitted or not.

## Macro variables

Macros have access to any positional or keyword argument passed along when calling the macro. Positional arguments are arguments that are assigned to variables based on the order they are provided, while keyword arguments are unordered and explicitly assign data to variable names. Keyword arguments can also have a default value if they aren't defined when the macro is called. There are three additional special variables that are available:

- `varargs`: This is a holding place for additional unexpected positional arguments that are passed along to the macro. These positional argument values will make up the `varargs` list.
- `kwargs`: This is the same as `varargs`; however, instead of holding extra positional argument values, it will hold a hash of extra keyword arguments and their associated values.
- `caller`: This can be used to call back to a higher-level macro that might have called this macro (yes, macros can call other macros).

In addition to these three special variables, there are a number of variables that expose internal details regarding the macro itself. These are a bit complicated, but we'll walk through their usage one by one. First, let's take a look at a short description of each variable:

- `name`: This is the name of the macro itself.
- `arguments`: This is a tuple of the names of the arguments that the macro accepts.

- `defaults`: This is a tuple of the default values.
- `catch_kwargs`: This is a Boolean that will be defined as `true` if the macro accesses (and, thus, accepts) the `kwargs` variable.
- `catch_varargs`: This is a Boolean that will be defined as `true` if the macro accesses (and, thus, accepts) the `varargs` variable.
- `caller`: This is a Boolean that will be defined as `true` if the macro accesses the `caller` variable (and, thus, could be called from another macro).

Similar to a class in Python, these variables need to be referenced via the name of the macro itself. Attempting to access these macros without prepending the name will result in undefined variables. Now, let's walk through and demonstrate the usage of each of them.

## name

The `name` variable is actually very simple. It simply provides a way to access the name of the macro as a variable, perhaps for further manipulation or usage. The following template includes a macro that references the name of the macro to render it in the output:

```
{% macro test() -%}
{{ test.name }}
{%- endmacro -%}
{{ test() }}
```

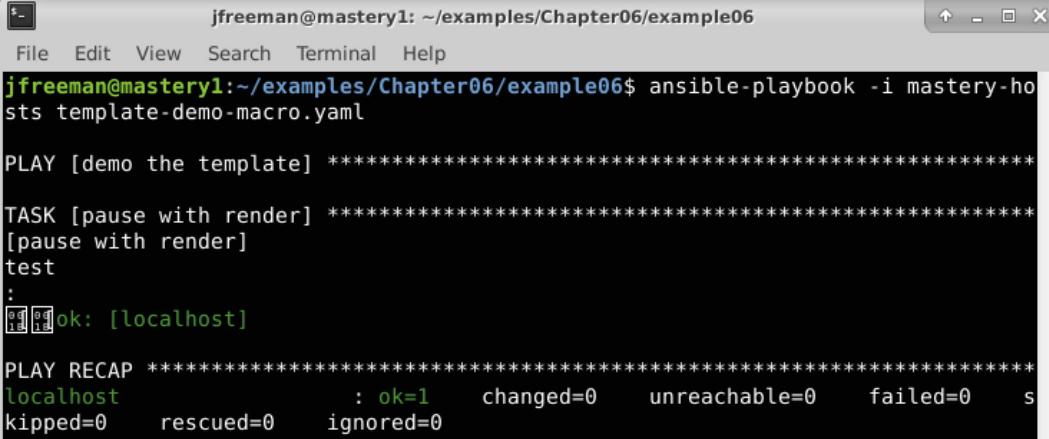
Let's say we were to create `demo-macro.j2` with this template and the following `template-demo-macro.yaml` playbook:

```
---
- name: demo the template
  hosts: localhost
  gather_facts: false
  vars:
    data_dirs: ['/','/foo','/bar']
  tasks:
    - name: pause with render
      ansible.builtin.pause:
        prompt: "{{ lookup('template', 'demo-macro.j2') }}"
```

We will run this playbook using the following command:

```
ansible-playbook -i mastery-hosts template-demo-macro.yaml
```

When you run the playbook, your output should look similar to the one shown in *Figure 6.8*:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter06/example06". The user runs the command "ansible-playbook -i mastery-hosts template-demo-macro.yaml". The output shows the playbook execution process:

```
jfreeman@mastery1:~/examples/Chapter06/example06$ ansible-playbook -i mastery-hosts template-demo-macro.yaml

PLAY [demo the template] ****
TASK [pause with render] ****
[pause with render]
test
:
[0]:ok: [localhost]

PLAY RECAP ****
localhost : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
```

Figure 6.8 – Rendering a template employing the name macro variable

As you see from this test run, our template is simply rendered with the macro name and nothing else, just as expected.

## arguments

The `arguments` variable is a tuple of the arguments that the macro accepts. Note that these are the explicitly defined arguments, not the special `kwargs` or `varargs`. Our previous example would have rendered an empty tuple, `()`, so let's modify it to get something else:

```
{% macro test(var_a='a string') -%}
{{ test.arguments }}
{%- endmacro -%}
{{ test() }}
```

Running the same playbook as before, to render this template in the same manner, should yield the output shown in *Figure 6.9*:

Figure 6.9 – Running a playbook to render a Jinja2 template that prints its macro arguments

In this example, we can clearly see that our template is rendered with the name of the arguments that the macro accepts (and not their values).

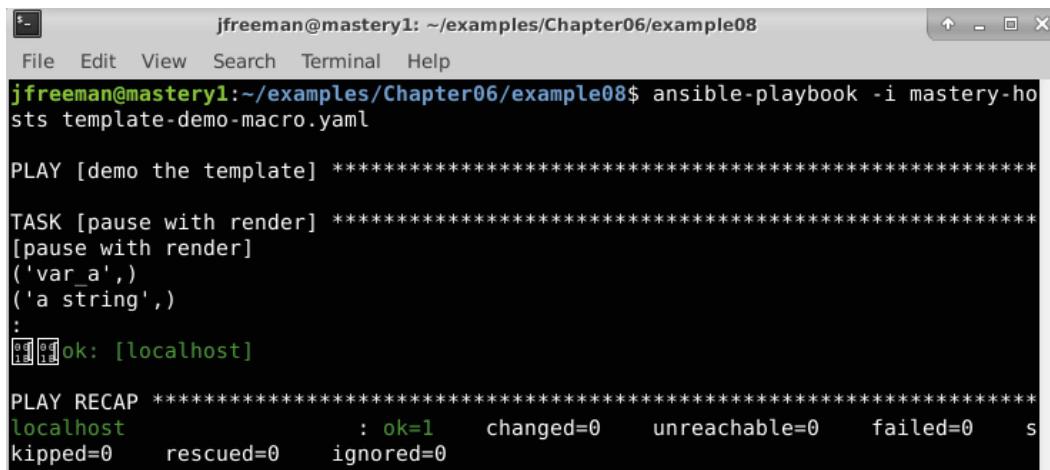
## Defaults

The `defaults` variable is a tuple of the default values for any keyword arguments that the macro explicitly accepts. Although still present in the documentation for Jinja2 (at the time of writing, an issue is open to correct the documentation), this variable was removed from all versions of Jinja2 that are newer than version 2.8.1. If you need to access this variable, you will need to downgrade your Jinja2 Python module to 2.8.1.

For those working with an older version of Jinja2, we can demonstrate this variable as follows; let's change our macro to display the default values as well as the arguments:

```
{% macro test(var_a='a string') -%}
{{ test.arguments }}
{{ test.defaults }}
{%- endmacro -%}
{{ test() }}
```

We can run our existing test playbook just as we have done before, but now using the newly updated template. If your version of Jinja2 supports the `defaults` variable, the output should look similar to the one shown in *Figure 6.10*:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter06/example08". The window shows the command "ansible-playbook -i mastery-hosts template-demo-macro.yaml" being run. The output shows a single task being executed on the localhost, which successfully renders the template. The final PLAY RECAP shows 1 task completed with 1 ok result.

```
jfreeman@mastery1:~/examples/Chapter06/example08$ ansible-playbook -i mastery-hosts template-demo-macro.yaml

PLAY [demo the template] ****
TASK [pause with render] ****
[pause with render]
('var_a',)
('a string',)
:
ok: [localhost]

PLAY RECAP ****
localhost : ok=1    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 6.10 – Rendering a Jinja2 template with the defaults and name macro variables

Here, we can see that the template is rendered with both the names and the default values of the arguments that are accepted by the macro.

### catch\_kwargs

This variable is only defined if the macro itself accesses the `kwargs` variable to catch any extra keyword arguments that might have been passed along. If defined, it will be set to `true`. Without accessing the `kwargs` variable, any extra keyword arguments in a call to the macro will result in an error when rendering the template. Likewise, accessing `catch_kwargs` without also accessing `kwargs` will result in an undefined error. Let's modify our example template again so that we can pass along additional `kwargs` variables:

```
{% macro test() -%}
{{ kwargs }}
{{ test.catch_kwargs }}
{%- endmacro -%}
{{ test(unexpected='surprise') }}
```

We can run our updated template through our existing rendering template again, using the same command as before. This time, the output should look similar to the one shown in *Figure 6.11*:

```
jfreeman@mastery1: ~/examples/Chapter06/example09$ ansible-playbook -i mastery-hosts template-demo-macro.yaml

PLAY [demo the template] ****
TASK [pause with render] ****
[pause with render]
{'unexpected': 'surprise'}
True
:
ok: [localhost]

PLAY RECAP ****
localhost          : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
```

Figure 6.11 – Rendering a template with the catch\_kwargs variable

As you can see from this output, the template does not give an error when an unexpected variable is passed to it and, instead, enables us to access the unexpected value(s) that were passed.

### catch\_varargs

Much like `catch_kwargs`, this variable only exists (and gets set to `true`) if the macro accesses the `varargs` variable. Modifying our example once more, we can see this in action:

```
{% macro test() -%}
{{ varargs }}
{{ test.catch_varargs }}
{%- endmacro -%}
{{ test('surprise') }}
```

The template's rendered result should look similar to the one shown in *Figure 6.12*:

```
jfreeman@mastery1:~/examples/Chapter06/example10$ ansible-playbook -i mastery-hosts template-demo-macro.yaml

PLAY [demo the template] ****
TASK [pause with render] ****
[pause with render]
('surprise',)
True
:
ok: [localhost]

PLAY RECAP ****
localhost                  : ok=1      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 6.12 – Rendering a template that makes use of the varargs and catch\_varargs macro variables

Again, we can see that we were able to catch and render the unexpected value that was passed to the macro rather than returning an error on render, which would have happened if we hadn't used `catch_varargs`.

## caller

The `caller` variable requires a bit more explanation. A macro can call out to another macro. This can be useful if the same chunk of the template will be used multiple times, but part of the internal data changes more than what could easily be passed as a macro parameter. The `caller` variable isn't exactly a variable; it's more of a reference back to the call to get the contents of that calling macro.

Let's update our template to demonstrate its usage:

```
{% macro test() -%}
The text from the caller follows: {{ caller() }}
{%- endmacro -%}
{% call test() -%}
This is text inside the call
{% endcall -%}
```

The rendered result should be similar to the one shown in *Figure 6.13*:

```
jfreeman@mastery1: ~/examples/Chapter06/example11$ ansible-playbook -i mastery-hosts template-demo-macro.yaml

PLAY [demo the template] ****
TASK [pause with render] ****
[pause with render]
The text from the caller follows:
This is text inside the call
:
[1]: ok: [localhost]

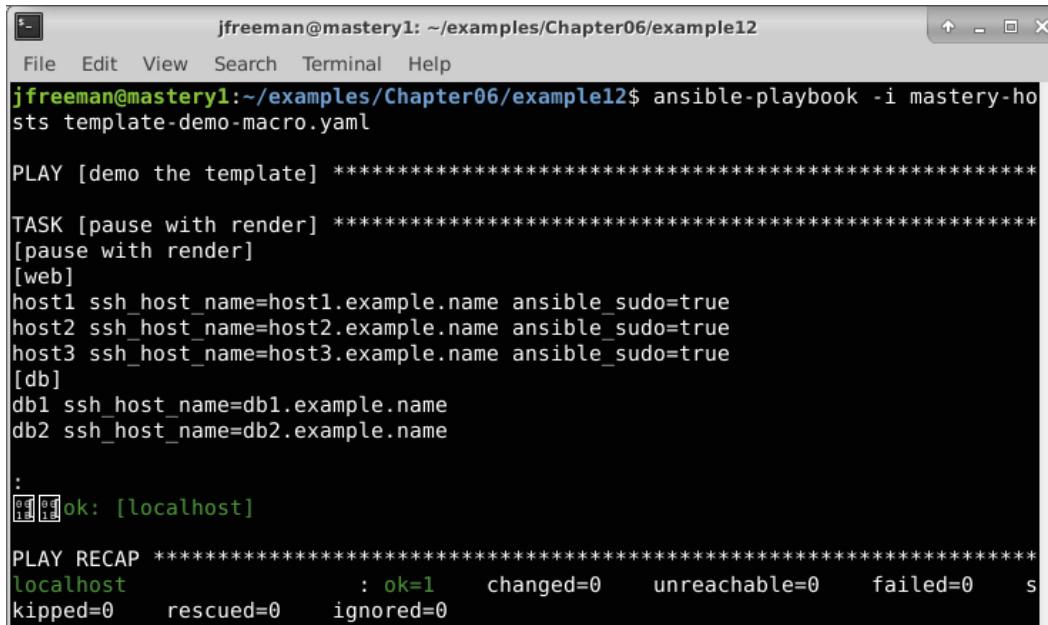
PLAY RECAP ****
localhost : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0   rescued=0   ignored=0
```

Figure 6.13 – Rendering a template that makes use of the caller variable

A call to a macro can still pass arguments to that macro; any combination of arguments or keyword arguments can be passed. If the macro utilizes `varargs` or `kwargs`, then more of those can be passed along as well. Additionally, a macro can pass arguments back to the caller, too! To demonstrate this, let's create a bigger example. This time, our example will generate a file that's suitable for an Ansible inventory:

```
{% macro test(group, hosts) -%}
[{{ group }}]
{%- for host in hosts -%}
{{ host }} {{ caller(host) }}
{%- endfor -%}
{%- endmacro -%}
{%- call(host) test('web', ['host1', 'host2', 'host3']) -%}
ssh_host_name={{ host }}.example.name ansible_sudo=true
{%- endcall -%}
{%- call(host) test('db', ['db1', 'db2']) -%}
ssh_host_name={{ host }}.example.name
{%- endcall -%}
```

Once rendered using our test playbook, the result should be as shown in *Figure 6.14*:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter06/example12". The command run is "ansible-playbook -i mastery-hosts template-demo-macro.yaml". The output shows a play for "demo the template" with a task for "pause with render". It lists hosts under two groups: [web] (host1, host2, host3) and [db] (db1, db2). A colon follows the [db] group. Below the hosts, there is a line starting with ":" followed by "ok: [localhost]". The final line is "PLAY RECAP" with statistics: localhost : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0.

```
jfreeman@mastery1: ~/examples/Chapter06/example12$ ansible-playbook -i mastery-hosts template-demo-macro.yaml

PLAY [demo the template] ****
TASK [pause with render] ****
[pause with render]
[web]
host1 ssh_host_name=host1.example.name ansible_sudo=true
host2 ssh_host_name=host2.example.name ansible_sudo=true
host3 ssh_host_name=host3.example.name ansible_sudo=true
[db]
db1 ssh_host_name=db1.example.name
db2 ssh_host_name=db2.example.name

:
ok: [localhost]

PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=0    s
kiped=0      rescued=0    ignored=0
```

Figure 6.14 – A more advanced example of a template rendered using the caller variable

We called the `test` macro twice, once for each group we wanted to define. Each group had a subtly different set of host variables to apply, and those were defined in the call itself. We saved typing by having the macro call back to the caller, passing along the `host` variable from the current loop.

Control blocks provide programming power inside templates, allowing template authors to make their templates more efficient. The efficiency isn't necessarily in the initial draft of the template; instead, efficiency really comes into play when a small change to a repeating value is needed. Now that we've looked, in detail, at building control structures within Jinja2, in the next section, we'll move on to look at ways that this powerful templating language can help us with another common automation requirement: data manipulation.

# Data manipulation

While control structures influence the flow of template processing, another tool exists that can help you to modify the contents of a variable. This tool is called a filter. Filters are the same as small functions, or methods, that can be run on the variable. Some filters operate without arguments, some take optional arguments, and some require arguments. Filters can be chained together as well, where the result of one filter action is fed into the next filter and then the next. Jinja2 comes with many built-in filters, and Ansible extends these with many custom filters that are available to you when using Jinja2 within templates, tasks, or any other place Ansible allows templating.

## Syntax

A filter is applied to a variable by way of the pipe symbol, |, followed by the name of the filter, and then any arguments for the filter inside parentheses. There can be a space between the variable name and the pipe symbol, as well as a space between the pipe symbol and the filter name. For example, if we wanted to apply the `lower` filter (which makes all the characters lowercase) to the `my_word` variable, we would use the following syntax:

```
{ { my_word | lower } }
```

Because the `lower` filter does not take any arguments, it is not necessary to attach an empty parentheses set to it. However, if we use a different filter that requires arguments, this all changes. Let's use the `replace` filter, which allows us to replace all occurrences of a substring with another substring. In this example, we want to replace all occurrences of the `no` substring with `yes` in the `answers` variable:

```
{ { answers | replace('no', 'yes') } }
```

Applying multiple filters is accomplished by simply adding more pipe symbols and more filter names. Let's combine both `replace` and `lower` to demonstrate the syntax – the filters are applied in the sequence listed. In the following example, first, we replace all instances of the `no` substring with `yes`, and then convert the entire resulting string into lowercase:

```
{ { answers | replace('no', 'yes') | lower } }
```

As we are doing a case-sensitive string replacement, you might choose to perform the lowercase conversion first, as this means you won't miss any instances of the word no regardless of the case – assuming, of course, this is the behavior you want! The code for this latter example would simply be as follows:

```
{ { answers | lower | replace('no', 'yes') } }
```

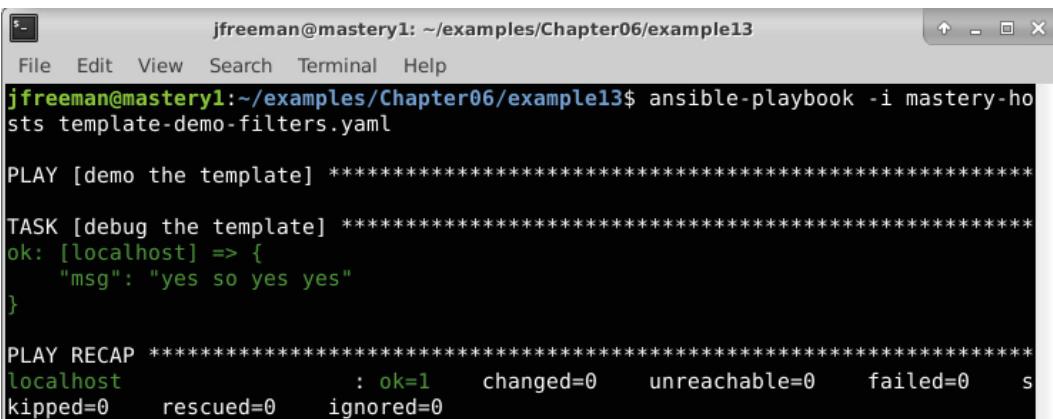
We can easily demonstrate this with a simple play that uses the debug command to render the line:

```
- name: demo the template
  hosts: localhost
  gather_facts: false
  vars:
    answers: "no so YES no"
  tasks:
    - name: debug the template
      ansible.builtin.debug:
        msg: "{ { answers | replace('no', 'yes') | lower } }"
```

Now, we can execute the playbook using the following command:

```
ansible-playbook -i mastery-hosts template-demo-filters.yaml
```

All instances of the word no within our answers variable that have been declared in the code will be replaced with the word yes. Additionally, all characters will be converted into lowercase. The output should be similar to the one shown in *Figure 6.15*:



The screenshot shows a terminal window titled 'jfreeman@mastery1: ~/examples/Chapter06/example13'. The terminal displays the command 'ansible-playbook -i mastery-hosts template-demo-filters.yaml'. The output shows the playbook execution details:

```
jfreeman@mastery1:~/examples/Chapter06/example13$ ansible-playbook -i mastery-hosts template-demo-filters.yaml
PLAY [demo the template] ****
TASK [debug the template] ****
ok: [localhost] => {
    "msg": "yes so yes yes"
}
PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0      rescued=0    ignored=0
```

Figure 6.15 – Demonstrating the use of chained filters in a simple Ansible playbook

Here, we can see that the playbook runs as expected and combines the two filters to operate on our test string, just as requested. Of course, these are just two of the filters available. In the next section, let's proceed to look at some of the more useful filters included with Jinja2.

## Useful built-in filters

A full list of the filters that are built into Jinja2 can be found in the Jinja2 documentation. At the time of writing this book, there are 50 built-in filters. We will take a look at some of the more commonly used filters next.

### Tip

If you want to look at the list of all available filters, the Jinja2 documentation for the current version (which was available at the time of writing) can be found at <https://jinja.palletsprojects.com/en/3.0.x/templates/#builtin-filters>.

### default

The `default` filter is a way to provide a default value for an otherwise undefined variable, which, in turn, prevents Ansible from generating an error. It is shorthand for a complex `if` statement, which checks whether a variable is defined before trying to use it with an `else` clause to provide a different value. Let's look at two examples that render the same thing. One uses the `if/else` structure, while the other uses the `default` filter:

```
{% if some_variable is defined -%}
{{ some_variable }}
{%- else -%}
default_value
{%- endif -%}
{{ some_variable | default('default_value') }}
```

The rendered result of each of these examples is the same; however, the example using the `default` filter is much quicker to write and easier to read.

While `default` is very useful, proceed with caution if you are using the same variable in multiple locations. Changing a default value can become a hassle, and it might be more efficient to define the variable with a default at the play or role level.

## length

The `length` filter will return the length of a sequence or a hash. In earlier editions of this book, we referenced a variable called `count`, which is an alias of `length` and accomplishes the same thing. This filter can be useful for performing any sort of math around the size of a set of hosts, or any other scenario where the count of some set needs to be known. Let's create an example where we set a `max_threads` configuration entry to match the count of the hosts in the play:

```
max_threads: {{ play_hosts | count }}
```

This provides us with a nice, concise way of getting the number of hosts contained within the `play_hosts` variable and assigning the answer to the `max_threads` variable.

## random

The `random` filter is used to make a random selection from a sequence. Let's use this filter to delegate a task to a random selection from the `db_servers` group:

```
name: backup the database
shell: mysqldump -u root nova > /data/nova.backup.sql
delegate_to: "{{ groups['db_servers'] | random }}"
run_once: true
```

Here, we can easily delegate this task to a single member of the `db_servers` group, which is picked at random using our filter.

## round

The `round` filter exists to round a number up or down. This can be useful if you need to perform floating-point math, and then turn the result into a rounded integer. The `round` filter takes optional arguments to define a precision (with a default of 0) and a rounding method. The possible rounding methods are `common` (which rounds up or down and is the default), `ceil` (which always rounds up), and `floor` (which always rounds down). In this example, we'll chain two filters together to round a math result to zero precision, and then turn that into an integer:

```
{{ math_result | round | int }}
```

Therefore, if the `math_result` variable was set to `3.4`, the output of the previous filter chain would be `3`.

## Useful Ansible provided custom filters

While there are many filters provided with Jinja2, Ansible includes some additional filters that playbook authors might find particularly useful. We'll highlight these next.

### Tip

These custom filters in Ansible change often between releases. They are worth reviewing, especially if you make heavy use of them. A full list of the custom Ansible filters is available at [https://docs.ansible.com/ansible/latest/user\\_guide/playbooks\\_filters.html](https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html).

## Filters related to task status

Ansible tracks task data for each task. This data is used to determine whether a task has failed, resulted in a change, or was skipped altogether. Playbook authors can register the results of a task, and in previous versions of playbooks, they would have used filters to check the tasks' status. As of Ansible 2.9, this has been removed completely. So, if you have any legacy playbooks from earlier Ansible versions, you might need to update them accordingly.

Before the release of Ansible 2.7, you would have used a conditional with a filter like this:

```
when: derp | success
```

You should now use the new syntax, which is shown in the following snippet. Note that the code in the following code block performs the same function:

```
when: derp is success
```

Let's view this in action in the following code:

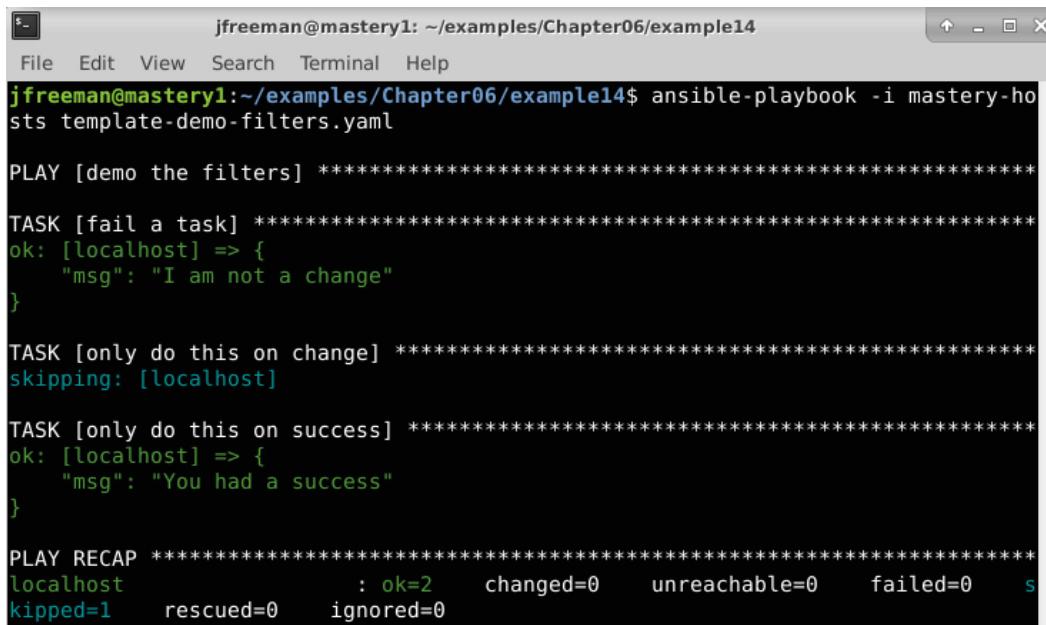
```
---
- name: demo the filters
  hosts: localhost
  gather_facts: false
  tasks:
    - name: fail a task
      ansible.builtin.debug:
        msg: "I am not a change"
      register: derp
    - name: only do this on change
```

```
ansible.builtin.debug:  
    msg: "You had a change"  
when: derp is changed  
- name: only do this on success  
  ansible.builtin.debug:  
    msg: "You had a success"  
when: derp is success
```

You can run this playbook using the following command:

```
ansible-playbook -i mastery-hosts template-demo-filters.yaml
```

The output is shown in *Figure 6.16*:



A screenshot of a terminal window titled 'jfreeman@mastery1: ~/examples/Chapter06/example14'. The terminal shows the execution of an Ansible playbook named 'template-demo-filters.yaml'. The output indicates two tasks were run: one for 'change' which failed, and one for 'success' which succeeded. The final recap shows 2 tasks completed successfully.

```
jfreeman@mastery1:~/examples/Chapter06/example14$ ansible-playbook -i mastery-hosts template-demo-filters.yaml  
  
PLAY [demo the filters] *****  
  
TASK [fail a task] *****  
ok: [localhost] => {  
    "msg": "I am not a change"  
}  
  
TASK [only do this on change] *****  
skipping: [localhost]  
  
TASK [only do this on success] *****  
ok: [localhost] => {  
    "msg": "You had a success"  
}  
  
PLAY RECAP *****  
localhost                  : ok=2      changed=0      unreachable=0      failed=0      s  
kiped=1      rescued=0      ignored=0
```

Figure 6.16 – Running an Ansible playbook with a conditional based on task status

As you can see, the `ansible.builtin.debug` statement resulted in `success`. So, we skipped the task to be run on a `change` and executed the one to be run on `success`.

## shuffle

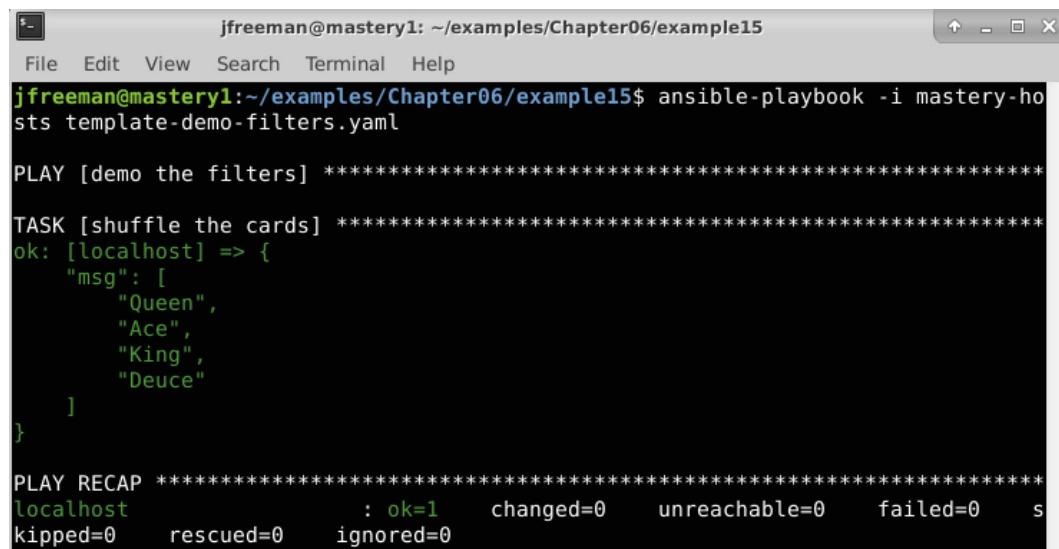
Similar to the `random` filter, the `shuffle` filter can be used to produce randomized results. Unlike the `random` filter, which selects one random choice from a list, the `shuffle` filter will shuffle the items in a sequence and return the full sequence:

```
---
- name: demo the filters
  hosts: localhost
  gather_facts: false
  tasks:
    - name: shuffle the cards
      ansible.builtin.debug:
        msg: "{{ ['Ace', 'Queen', 'King', 'Deuce'] | shuffle
}}"
```

Run this playbook using the following command:

```
ansible-playbook -i mastery-hosts template-demo-filters.yaml
```

The output is shown in *Figure 6.17*:



```
jfreeman@mastery1: ~/examples/Chapter06/example15
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter06/example15$ ansible-playbook -i mastery-hosts template-demo-filters.yaml

PLAY [demo the filters] ****
TASK [shuffle the cards] ****
ok: [localhost] => {
  "msg": [
    "Queen",
    "Ace",
    "King",
    "Deuce"
  ]
}

PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

Figure 6.17 – Running a playbook that makes use of the `shuffle` filter

As expected, the whole list returned but with the order shuffled. If you run the playbook repeatedly, you will see a different order of the returned list on each run. Try this for yourself!

## Filters dealing with pathnames

Configuration management and orchestration frequently refer to pathnames, but often, only part of the path is desired. For example, we might need the full path to a file but not the filename itself. Or, perhaps we just need to extract the filename from a full path, ignoring the directories preceding it. Ansible provides a few filters to help with precisely these tasks, and we will examine them in the following sections.

### basename

Let's say we have a requirement to work with just the filename from a full path. Of course, we could perform some complex pattern matching to do this. However, often, this results in code that is not easy to read and can be difficult to maintain. Luckily, Ansible provides a filter specifically for extracting the filename from a full path, as we will demonstrate next. In this example, we will use the basename filter to extract the filename from a full path:

```
---
- name: demo the filters
  hosts: localhost
  gather_facts: false
  tasks:
    - name: demo basename
      ansible.builtin.debug:
        msg: "{{ '/var/log/nova/nova-api.log' | basename }}"
```

Run this playbook using the following command:

```
ansible-playbook -i mastery-hosts template-demo-filters.yaml
```

The output is shown in *Figure 6.18*:



```
jfreeman@mastery1: ~/examples/Chapter06/example16
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter06/example16$ ansible-playbook -i mastery-hosts template-demo-filters.yaml

PLAY [demo the filters] ****
TASK [demo basename] ****
ok: [localhost] => {
    "msg": "nova-api.log"
}

PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0      rescued=0    ignored=0
```

Figure 6.18 – Running a playbook that makes use of the basename filter

Here, you can see that just the filename was returned from the full path, as desired.

## dirname

The inverse of basename is dirname. Instead of returning the final part of a path, dirname will return everything else (except the filename, which is the final part of the full path). Let's change our previous play to use dirname and rerun it using the same command. The output should now look similar to the one shown in *Figure 6.19*:



```
jfreeman@mastery1: ~/examples/Chapter06/example17
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter06/example17$ ansible-playbook -i mastery-hosts template-demo-filters.yaml

PLAY [demo the filters] ****
TASK [demo dirname] ****
ok: [localhost] => {
    "msg": "/var/log/nova"
}

PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0      rescued=0    ignored=0
```

Figure 6.19 – Running a playbook using the dirname filter

Now, we have just the path of our variable, which could be extremely useful elsewhere in our playbook.

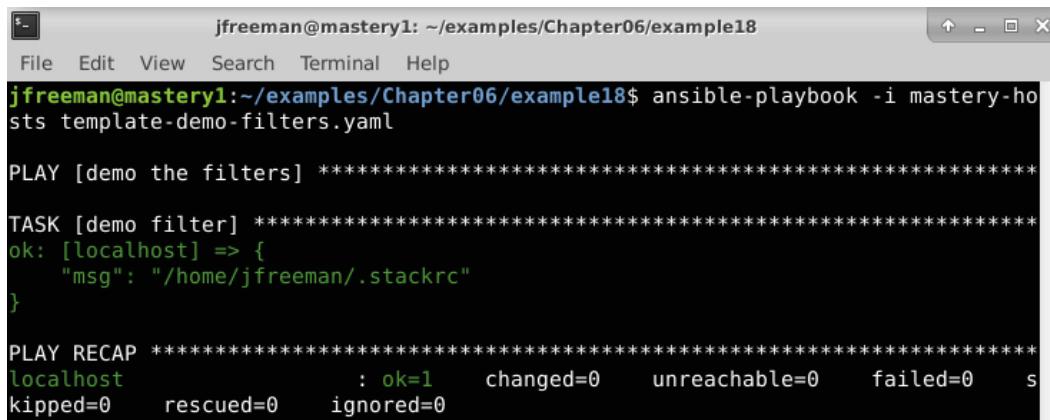
## expanduser

Often, paths to various things are supplied with a user shortcut, such as `~/.stackrc`. However, some tasks might require the full path to the file. Rather than the complicated command and register calls, the `expanduser` filter provides a way to expand the path to the full definition. In this example, the username is `jfreeman`:

```
---
```

```
- name: demo the filters
  hosts: localhost
  gather_facts: false
  tasks:
    - name: demo filter
      ansible.builtin.debug:
        msg: "{{ ' ~/.stackrc' | expanduser }}"
```

You can run this playbook with the same command as before, and the output should look similar to the one shown in *Figure 6.20*:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter06/example18". The terminal contains the following output:

```
jfreeman@mastery1:~/examples/Chapter06/example18$ ansible-playbook -i mastery-hosts template-demo-filters.yaml

PLAY [demo the filters] ****
TASK [demo filter] ****
ok: [localhost] => {
    "msg": "/home/jfreeman/.stackrc"
}

PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 6.20 – Running a playbook using the `expanduser` filter

Here, we have successfully expanded the path, which could be useful for creating configuration files or performing other file operations that might need an absolute pathname rather than a relative pathname.

## Base64 encoding

When reading content from remote hosts, such as with the `ansible.builtin.slurp` module (which is used to read file content from remote hosts into a variable), the content will be Base64 encoded. To decode such content, Ansible provides a `b64decode` filter. Similarly, if running a task that requires Base64-encoded input, regular strings can be encoded using the `b64encode` filter.

Let's use Ansible to create a test file, called `/tmp/derp`, which will contain a test string. Then, we'll use the `ansible.builtin.slurp` module to obtain the file contents and decode them using the aforementioned filter:

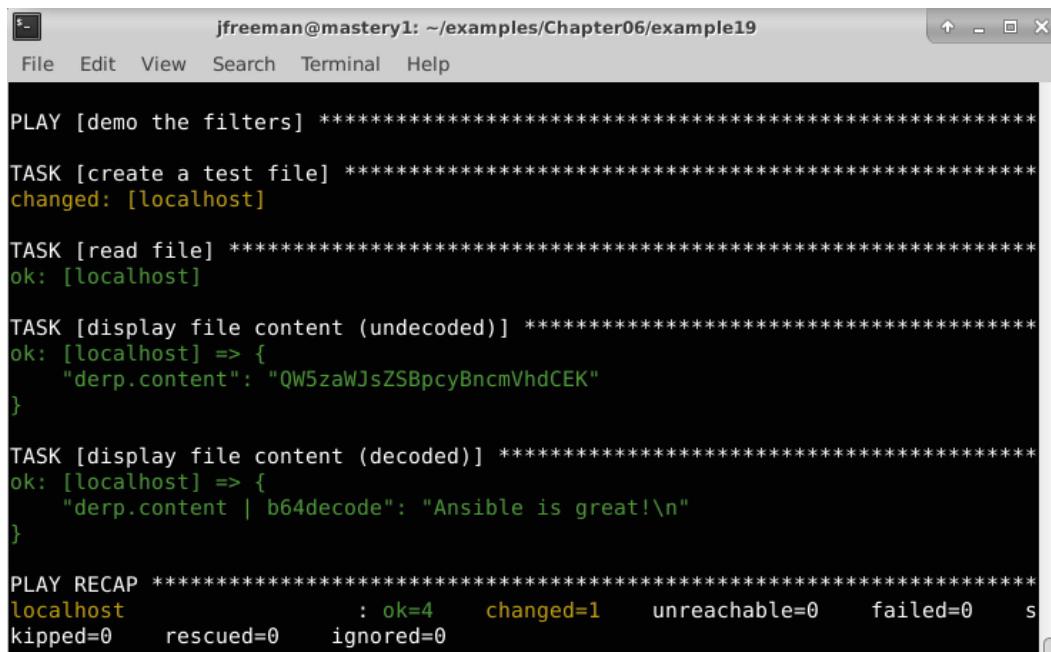
```
---
```

```
- name: demo the filters
  hosts: localhost
  gather_facts: false
  tasks:
    - name: create a test file
      ansible.builtin.lineinfile:
        path: /tmp/derp
        line: "Ansible is great!"
        state: present
        create: yes
    - name: read file
      ansible.builtin.slurp:
        src: /tmp/derp
        register: derp
    - name: display file content (undecoded)
      ansible.builtin.debug:
        var: derp.content
    - name: display file content (decoded)
      ansible.builtin.debug:
        var: derp.content | b64decode
```

If you are working with the example code that accompanies this book, run the playbook using the following command:

```
ansible-playbook -i mastery-hosts template-demo-filters.yaml
```

The output is shown in *Figure 6.21*:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter06/example19". The window shows the execution of an Ansible playbook named "example19". The output is as follows:

```
PLAY [demo the filters] *****
TASK [create a test file] *****
changed: [localhost]

TASK [read file] *****
ok: [localhost]

TASK [display file content (undecoded)] *****
ok: [localhost] => {
    "derp.content": "QW5zaWJsZSBpcyBncmVhdCEK"
}

TASK [display file content (decoded)] *****
ok: [localhost] => {
    "derp.content | b64decode": "Ansible is great!\n"
}

PLAY RECAP *****
localhost : ok=4      changed=1      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 6.21 – Running a playbook featuring the b64decode filter

Here, we successfully read the small file we created into a variable. Additionally, we can see the variable contents in Base64-encoded form (remember that this encoding was performed by the `ansible.builtin.slurp` module). We can then decode it using a filter to view the original file contents.

## Searching for content

It is relatively common in Ansible to search a string for a substring. In particular, the common administrator task of running a command and grepping the output for a particular key piece of data is a reoccurring construct in many playbooks. While it's possible to replicate this with a shell task to execute a command, pipe the output into `grep`, and use careful handling of `failed_when` to catch the `grep` exit codes, a far better strategy is to use a command task, `register` the output, and then utilize the Ansible-provided **Regular Expression (regex)** filters in later conditionals.

Let's take a look at two examples: one using `ansible.builtin.shell`, the pipe, and the `grep` method, and another using the `search` test:

```
- name: check database version
  ansible.builtin.shell: neutron-manage current | grep juno
  register: neutron_db_ver
  failed_when: false
- name: upgrade db
  ansible.builtin.command: neutron-manage db_sync
  when: neutron_db_ver is failed
```

The preceding example works by forcing Ansible to always see the task as successful but assumes that if the exit code from the shell is nonzero, then the `juno` string was not found in the output of the `neutron-manage` command. This construct is functional but complex to read, and it could mask real errors from the command. Let's try again using the `search` test.

As we previously mentioned, regarding the task status, using `search` on a string in Ansible is considered a test and is deprecated. Although it might read slightly odd, in order to be compliant with Ansible 2.9 and later versions, we must use the `is` keyword in place of the pipe when using `search` in this context:

```
- name: check database version
  ansible.builtin.command: neutron-manage current
  register: neutron_db_ver
- name: upgrade db
  ansible.builtin.command: neutron-manage db_sync
  when: not neutron_db_ver.stdout is search('juno')
```

Here, we are requesting to run the task named `upgrade db` when `neutron_db_ver.stdout` does not contain the `juno` string. Once you get used to the concept of `when: not ... is`, you can see that this version is much cleaner to follow and does not mask errors from the first task.

The `search` filter searches a string and will return `True` if the substring is found anywhere inside the input string. However, if an exact complete match is desired instead, the `match` filter can be used. Full Python regex syntax can be utilized inside the `search/match` string.

## Omitting undefined arguments

The `omit` variable requires a bit of explanation. Sometimes, when iterating over a hash of data to construct task arguments, it might be necessary to only provide some arguments for some of the items in the hash. Even though Jinja2 supports inline `if` statements to conditionally render parts of a line, this does not work well in an Ansible task. Traditionally, playbook authors would create multiple tasks, one for each set of potential arguments passed in, and use conditionals to sort the loop members between each task set. A recently added magic variable named `omit` solves this problem when used in conjunction with the `default` filter. The `omit` variable will remove the argument the variable was used with altogether.

To illustrate how this works, let's consider a scenario where we need to install a set of Python packages with `ansible.builtin.pip`. Some of the packages have a specific version, while others do not. These packages are in a list of hashes named `pips`. Each hash has a `name` key and, potentially, a `ver` key. Our first example utilizes two different tasks to complete the installation:

```
- name: install pips with versions
  ansible.builtin.pip: "name={{ item.name }} version={{ item.
    ver }}"
  loop: "{{ pips }}"
  when: item.ver is defined
- name: install pips without versions
  ansible.builtin.pip: "name={{ item.name }}"
  loop: "{{ pips }}"
  when: item.ver is undefined
```

This construct works, but the loop is iterated twice, and some of the iterations will be skipped in each task. The following example collapses the two tasks into one, and utilizes the `omit` variable:

```
- name: install pips
  ansible.builtin.pip: "name={{ item.name }} version={{ item.
    ver | default(omit) }}"
  loop: "{{ pips }}"
```

This example is shorter, cleaner, and doesn't generate additional skipped tasks.

## Python object methods

Jinja2 is a Python-based template engine, and so Python object methods are available within templates. Object methods are methods, or functions, that are directly accessible by the variable object (typically, a `string`, `list`, `int`, or `float`). A good way to think about this is as follows: if you were writing Python code and could write the variable, then a period, and then a method call, you would have access to do the same in Jinja2. Within Ansible, only methods that return modified content or a Boolean are typically used. Let's explore some common object methods that might be useful in Ansible.

### String methods

String methods can be used to return new strings, return a list of strings that have been modified in some way, or test a string for various conditions and return a Boolean. Some useful methods are as follows:

- `endswith`: This determines whether the string ends with a substring.
- `startswith`: This is the same as `endswith` but from the start.
- `split`: This splits the string on characters (the default is space) into a list of substrings.
- `rsplit`: This is the same as `split`, but it starts from the end of the string and works backward.
- `splitlines`: This splits the string at newlines into a list of substrings.
- `upper`: This returns a copy of the string all in uppercase.
- `lower`: This returns a copy of the string all in lowercase.
- `capitalize`: This returns a copy of the string with just the first character in uppercase.

We can create a simple playbook that will utilize some of these methods in a single task:

```
---
- name: demo the filters
  hosts: localhost
  gather_facts: false

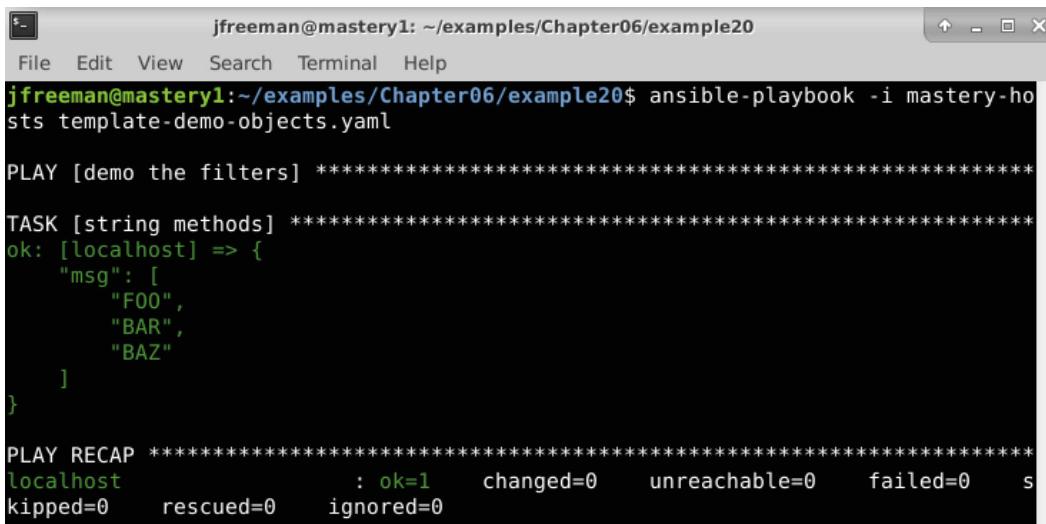
  tasks:
    - name: string methods
```

```
ansible.builtin.debug:  
  msg: "{{ 'foo bar baz'.upper().split() }}"
```

If you are using the example code that accompanies this book, run this playbook using the following command:

```
ansible-playbook -i mastery-hosts template-demo-objects.yaml
```

The output will look similar to the one shown in *Figure 6.22*:



A screenshot of a terminal window titled 'jfreeman@mastery1: ~/examples/Chapter06/example20'. The window shows the command 'ansible-playbook -i mastery-hosts template-demo-objects.yaml' being run. The output displays the execution of a task that uses Python string methods ('upper()' and 'split()') on a list of strings ('FOO', 'BAR', 'BAZ'). The task is successful ('ok: [localhost] => {'). The final PLAY RECAP shows 1 host with 0 failures.

```
jfreeman@mastery1:~/examples/Chapter06/example20$ ansible-playbook -i mastery-hosts template-demo-objects.yaml  
  
PLAY [demo the filters] *****  
  
TASK [string methods] *****  
ok: [localhost] => {  
  "msg": [  
    "FOO",  
    "BAR",  
    "BAZ"  
  ]  
}  
  
PLAY RECAP *****  
localhost : ok=1    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 6.22 – Running a playbook that makes use of the Python string object methods

As these are object methods, we need to access them using dot notation rather than with a filter via | .

## List methods

Most of the methods Ansible provides relating to lists perform modifications on the list itself. However, there are two list methods that are useful when working with lists, especially when loops are involved. These two functions are `index` and `count`, and their functionality is described as follows:

- `index`: This returns the first index position of a provided value.
- `count`: This counts the items in the list.

These functions can be incredibly useful when iterating through a list in a loop, as it allows positional logic to be performed and appropriate actions to be taken, given our position in the list as we work through it. This is common in other programming languages, and fortunately, Ansible also provides this.

## The int and float methods

Most `int` and `float` methods are not useful for Ansible. Sometimes, our variables are not exactly in the format we want them in. However, instead of defining more and more variables that slightly modify the same content, we can make use of Jinja2 filters to carry out the manipulation for us in the various places that require that modification. This allows us to stay efficient with the definition of our data, preventing numerous duplicate variables and tasks that might have to be changed later.

## Comparing values

Comparisons are used in many places with Ansible. Task conditionals are comparisons. Jinja2 control structures, such as `if/elif/else` blocks, `for` loops, and macros, often use comparisons; some filters use comparisons as well. To master Ansible's usage of Jinja2, it is important to understand what comparisons are available.

## Comparisons

Like most languages, Jinja2 comes equipped with the standard set of comparison expressions you would expect, which will render a Boolean `true` or `false`.

The expressions in Jinja2 are as follows:

Expression	Definition
<code>==</code>	True if two objects are equal
<code>!=</code>	True if two objects are not equal
<code>&gt;</code>	True if the left-hand side is greater than the right-hand side
<code>&lt;</code>	True if the left-hand side is less than the right-hand side
<code>&gt;=</code>	True if the left-hand side is greater than or equal to the right-hand side
<code>&lt;=</code>	True if the left-hand side is less than or equal to the right-hand side

If you have written comparison operations in almost any other programming language (usually in the form of an `if` statement), these should all seem very familiar. Jinja2 maintains this functionality in templates, allowing for the same powerful comparison operations you would expect in conditional logic from any good programming language.

## Logic

Sometimes, performing a single comparison operation on its own is not enough – perhaps we might want to perform an action if two comparisons evaluate to `true` at the same time. Alternatively, we might want to perform an operation only if a comparison is not true. Logic in Jinja2 helps you to group two or more comparisons together, allowing for the formation of complex conditions from simple comparisons. Each comparison is referred to as an operand, and the logic that's used to bind these together into complex conditionals is given in the following list:

- `and`: This returns `true` if the left and the right operand are true.
- `or`: This returns `true` if the left or the right operand is true.
- `not`: This negates an operand.
- `()`: This wraps a set of operands together to form a larger operand.

To build on the definition of logical conditions in Jinja2, we can perform tests for certain variable conditions such as if a variable is defined or not. We will look at this in more detail in the next section.

## Tests

A test in Jinja2 is used to determine whether a variable matches certain well-defined criteria, and we have already come across this in this chapter in specific scenarios. The `is` operator is used to initiate a test. Tests are used wherever a Boolean result is desired, such as with `if` expressions and task conditionals. There are many built-in tests, but we'll highlight a few of the particularly useful ones, as follows:

- `defined`: This returns `true` if the variable is defined.
- `undefined`: This is the opposite of `defined`.
- `none`: This returns `true` if the variable is defined, but the value is `None`.
- `even`: This returns `true` if the number is divisible by 2.
- `odd`: This returns `true` if the number is not divisible by 2.

To test whether a value is not something, simply use `is not`.

We can create a playbook to demonstrate some of these value comparisons:

```
---
- name: demo the logic
  hosts: localhost
  gather_facts: false
  vars:
    num1: 10
    num3: 10
  tasks:
    - name: logic and comparison
      ansible.builtin.debug:
        msg: "Can you read me?"
        when: num1 >= num3 and num1 is even and num2 is not
              defined
```

If you are running the code that accompanies this book, you can execute this example playbook using the following command:

```
ansible-playbook -i mastery-hosts template-demo-comparisons.yaml
```

The output is shown in *Figure 6.23*:

```
jfreeman@mastery1: ~/examples/Chapter06/example21
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter06/example21$ ansible-playbook -i mastery-hosts template-demo-comparisons.yaml

PLAY [demo the logic] ****
TASK [logic and comparison] ****
ok: [localhost] => {
    "msg": "Can you read me?"
}

PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 6.23 – Executing a playbook containing a complex conditional

Here, we can see that our complex conditional evaluated as `true`, and so the debug task was executed.

That concludes our look at Ansible's extensive templating capabilities. We hope that this chapter has sown seeds of ideas for you on ways to efficiently automate your infrastructure.

## Summary

Jinja2 is a powerful language that is extensively used by Ansible. Not only is it used to generate file content, but it is also used to make portions of a playbook dynamic. Mastering Jinja2 is vital for creating and maintaining elegant and efficient playbooks and roles.

In this chapter, we learned how to build simple templates with Jinja2 and render them from an Ansible playbook. Additionally, we learned how to make effective use of control structures, how to manipulate data, and even how to perform comparisons and tests on variables to both control the flow of Ansible playbooks (by keeping the code lightweight and efficient) and create and manipulate data without the need for duplicate definitions or excessive numbers of variables.

In the next chapter, we will explore Ansible's capability in more depth to define what constitutes a change or failure for tasks within a play.

## Questions

1. Jinja2 conditionals can be used to render content inline with a playbook task.
  - a) True
  - b) False
2. Which of the following Jinja2 constructs will print an empty line each time it is evaluated?
  - a) { % if loop.first -%}
  - b) { % if loop.first %}
  - c) {%- if loop.first -%}
  - d) {%- if loop.first %}

3. Jinja2 macros can be used to do which of the following?
  - a) Define a sequence of keystrokes that need to be automated.
  - b) Define a function for automating spreadsheets with Ansible.
  - c) Define a function that gets called regularly from elsewhere in the template.
  - d) Macros are not used in Jinja2.
4. Which of the following is a valid expression for chaining two Jinja2 filters together to operate on an Ansible variable?
  - a) {{ value.replace('A', 'B').lower }}
  - b) {{ value | replace('A', 'B') | lower }}
  - c) value.replace('A', 'B').lower
  - d) lower(replace('A', 'B', value))
5. Jinja2 filters always have mandatory arguments.
  - a) True
  - b) False
6. Which Ansible custom filter would you use to retrieve a random entry from a list variable?
  - a) shuffle
  - b) random
  - c) select
  - d) rand
7. Ansible can extract the filename from a full path using which filter?
  - a) filename
  - b) dirname
  - c) expanduser
  - d) basename

8. Ansible provides a construct for skipping optional arguments to prevent undefined variable errors. What is it called?
  - a) skip\_var
  - b) skip
  - c) omit
  - d) prevent\_undefined
9. Complex conditionals can be constructed for Ansible tasks using which operators?
  - a) and, or, and not
  - b) and, nand, or, nor, and not
  - c) &&, | |, and !
  - d) &, |, and !
10. Which of the following task execution conditionals will allow the task to run if the previous task has been completed successfully?
  - a) previoustask | success
  - b) previoustask = success
  - c) previoustask == success
  - d) previoustask is success

# 7

# Controlling Task Conditions

Ansible is a system for running tasks on one or more hosts, and ensuring that operators understand whether changes have occurred (and indeed whether any issues were encountered). As a result, Ansible tasks result in one of four possible statuses: `ok`, `changed`, `failed`, or `skipped`. These statuses perform a number of important functions.

From the perspective of an operator running an Ansible playbook, they provide an overview of the Ansible run that has been completed—whether anything changed or not and whether there were any failures that need addressing. In addition, they determine the flow of the playbook—for example, if a task results in a `changed` status, we might want to perform a restart of the service, but otherwise leave it running. Ansible possesses all the necessary functions to achieve this.

Similarly, if a task results in a `failed` status, the default behavior of Ansible is not to attempt any further tasks on that host. Tasks can also make use of conditionals that check the status of previous tasks to control operations. As a result, these statuses, or task conditions, are central to just about everything Ansible does, and it is important to understand how to work with them and hence control the flow of a playbook to cater for cases where, for example, a failure might occur. We'll look at how exactly to handle such things in this chapter.

In this chapter, we'll explore this in detail, focusing specifically on the following topics:

- Controlling what defines a failure
- Recovering gracefully from a failure
- Controlling what defines a change
- Iterating over a set of tasks using loops

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running **Ansible 4.3** or newer. Almost any flavor of Linux should do—for those interested in specifics, all the code presented in this chapter was tested on **Ubuntu Server 20.04 LTS** unless stated otherwise, and on Ansible 4.3. The example code that accompanies this chapter can be downloaded from GitHub at this URL: <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter07>.

Check out the following video to see the Code in Action: <https://bit.ly/3AVXxME>.

## Defining a failure

Most modules that ship with Ansible have differing criteria for what constitutes an error. An error condition is highly dependent upon the module and what the module is attempting to accomplish. When a module returns an error, the host will be removed from the set of available hosts, preventing any further tasks or handlers from being executed on that host. Furthermore, the `ansible-playbook` and `ansible` executables will exit with a non-zero exit code to indicate failure. However, we are not limited by a module's opinion of what an error is. We can ignore errors or redefine an error condition.

## Ignoring errors

A task condition named `ignore_errors` is used to ignore errors. This condition is a Boolean, meaning that the value should be something Ansible understands to be `true`, such as `yes`, `on`, `true`, or `1` (string or integer).

To demonstrate how to use `ignore_errors`, let's create a playbook where we attempt to query a web server that doesn't exist. Typically, this would be an error, and if we don't define `ignore_errors`, we get the default behavior; that is, the host will be marked as failed and no further tasks will be attempted on that host. Create a new playbook called `error.yaml`, as follows, to look further at this behavior:

```
---
- name: error handling
  hosts: localhost
  gather_facts: false

  tasks:
    - name: broken website
      ansible.builtin.uri:
        url: http://notahost.nodomain
```

Run this playbook with the following command:

```
ansible-playbook -i mastery-hosts error.yaml
```

The single task in this playbook should result in an error that looks like that shown in *Figure 7.1*:

```
jfreeman@mastery1: ~/examples/Chapter07/example01
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter07/example01$ ansible-playbook -i mastery-hosts error.yaml

PLAY [error handling] ****
TASK [broken website] ****
fatal: [localhost]: FAILED! => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": false, "elapsed": 0, "msg": "Status code was -1 and not [200]: Request failed: <urlopen error [Errno -2] Name or service not known>", "redirected": false, "status": -1, "url": "http://notahost.nodomain"}
PLAY RECAP ****
localhost                  : ok=0    changed=0    unreachable=0    failed=1    skipped=0    rescued=0    ignored=0
```

Figure 7.1 – Running a playbook that deliberately induces a task error

Now, let's imagine that we didn't want Ansible to stop here, and instead we wanted it to continue. We can add the `ignore_errors` condition to our task like this:

```
- name: broken website
  ansible.builtin.uri:
    url: http://notahost.nodomain
  ignore_errors: true
```

This time, when we run the playbook using the same command as before, our error will be ignored, as shown in *Figure 7.2*:

```
jfreeman@mastery1: ~/examples/Chapter07/example02$ ansible-playbook -i mastery-hosts error.yaml

PLAY [error handling] ****
TASK [broken website] ****
fatal: [localhost]: FAILED! => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": false, "elapsed": 0, "msg": "Status code was -1 and not [200]: Request failed: <urlopen error [Errno -2] Name or service not known>", "redirected": false, "status": -1, "url": "http://notahost.nodomain"}  
...ignoring

PLAY RECAP ****
localhost                  : ok=1      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=1
```

Figure 7.2 – Running the same playbook but with the `ignore_errors` task condition added

Any further tasks for that host will still be attempted and the playbook does not register any failed hosts.

## Defining an error condition

The `ignore_errors` condition is a bit of a blunt instrument. Any error generated from the module used by the task will be ignored. Furthermore, the output, at first glance, still appears like an error and may be confusing to an operator attempting to discover a real failure. A more subtle tool is the `failed_when` condition. This condition is more like a fine scalpel, allowing a playbook author to be very specific as to what constitutes an error for a task. This condition performs a test to generate a Boolean result, much like the `when` condition. If the condition results in a Boolean `true` value, the task will be considered a failure. Otherwise, the task will be considered successful.

The `failed_when` condition is quite useful when used in combination with the `command` or `shell` module and registering the result of the execution. Many programs that are executed can have detailed non-zero exit codes that mean different things. However, these Ansible modules all consider an exit code of anything other than 0 to be a failure. Let's look at the `iscsiadm` utility. This utility can be used for many things related to iSCSI. For the sake of demonstration, we'll replace our `uri` module in `error.yaml` and attempt to discover any active `iscsi` sessions:

```
- name: query sessions
  ansible.builtin.command: /sbin/iscsiadm -m session
  register: sessions
```

Run this playbook using the same command as before; unless you happen to be on a system with active iSCSI sessions, you will see output very much like that in *Figure 7.3*:

```
jfreeman@mastery1: ~/examples/Chapter07/example03
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter07/example03$ ansible-playbook -i mastery-hots error.yaml

PLAY [error handling] ****
TASK [query sessions] ****
fatal: [localhost]: FAILED! => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "cmd": ["/sbin/iscsiadm", "-m", "session"], "delta": "0:00:00.003872", "end": "2021-05-31 16:24:49.485125", "msg": "non-zero return code", "rc": 21, "start": "2021-05-31 16:24:49.481253", "stderr": "iscsiadm: No active sessions.", "stderr_lines": ["iscsiadm: No active sessions."], "stdout": "", "stdout_lines": []}

PLAY RECAP ****
localhost                  : ok=0    changed=0    unreachable=0    failed=1    s
kipped=0      rescued=0      ignored=0
```

Figure 7.3 – Running a playbook to discover active iSCSI sessions without any failure handling

#### Important Note

The `iscsiadm` tool may not be installed by default, in which case you will get a different error from the preceding one. On our Ubuntu Server 20.04 test machine, it was installed using the following command: `sudo apt install open-iscsi`.

We can just use the `ignore_errors` condition, but that would mask other problems with `iscsi`, so instead of this, we want to instruct Ansible that an exit code of 21 is acceptable. To that end, we can make use of the registered variable to access the `rc` variable, which holds the return code. We'll make use of this in a `failed_when` statement:

```
- name: query sessions
  command: /sbin/iscsiadm -m session
  register: sessions
  failed_when: sessions.rc not in (0, 21)
```

We simply stated that any exit code other than 0 or 21 should be considered a failure. Run the playbook again, but this time with added verbosity, using the `-v` flag on your command like this:

```
ansible-playbook -i mastery-hosts error.yaml -v
```

Assuming again that you have no active iSCSI sessions, the output will look like that shown in *Figure 7.4*. Use of the `-v` flag is, of course, not mandatory, but it is helpful in this case as it shows us the exit code of the `iscsiadm` utility:

```
jfreeman@mastery1:~/examples/Chapter07/example04
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter07/example04$ ansible-playbook -i mastery-hosts error.yaml -v
No config file found; using defaults

PLAY [error handling] ****
TASK [query sessions] ****
changed: [localhost] => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "cmd": ["/sbin/iscsiadm", "-m", "session"], "delta": "0:00:00.004058", "end": "2021-05-31 16:25:11.109908", "failed_when_result": false, "msg": "non-zero return code", "rc": 21, "start": "2021-05-31 16:25:11.105850", "stderr": "iscsiadm: No active sessions.", "stderr_lines": ["iscsiadm: No active sessions."], "stdout": "", "stdout_lines": []}

PLAY RECAP ****
localhost                  : ok=1    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 7.4 – Running the same playbook but handling failures based on the command exit code

The output now shows no error, and, in fact, we see a new data key in the results—`failed_when_result`. This shows whether our `failed_when` statement rendered `true` or `false`; it was `false` in this case.

Many command-line tools do not have detailed exit codes. In fact, most typically use 0 for success and another non-zero code for all failure types. Thankfully, the `failed_when` statement is not just limited to the exit code of the application; it is a free-form Boolean statement that can access any sort of data required. Let's look at a different problem, one involving Git. We'll imagine a scenario where we want to ensure that a particular branch does not exist in a Git checkout. This task assumes a Git repository is checked out in the `/srv/app` directory. The command to delete a Git branch is `git branch -D`. Let's have a look at the following code snippet:

```
- name: delete branch bad
  ansible.builtin.command: git branch -D badfeature
  args:
    chdir: /srv/app
```

For this code to work, you will need to check a Git repository out into the preceding directory. If you don't have one to test with, you can easily create one using the following commands (just make sure you don't have anything important in `/srv/app` that could get overwritten!):

```
sudo mkdir -p /srv/app
sudo chown $USER /srv/app
cd /srv/app
git init
git commit --allow-empty -m "initial commit"
```

Once you have done this, you are ready to run the updated playbook task we detailed previously. Like before, we'll add verbosity to the output so that we can better understand the behavior of our playbook.

#### Important Note

The `ansible.builtin.command` and `ansible.builtin.shell` modules use a different format for providing module arguments. `ansible.builtin.command` itself is provided in free form, while module arguments go into an `args` hash.

Running the playbook as described should yield an error, as `git` will produce an exit code of 1, as the branch does not exist, as shown in *Figure 7.5*:

```
jfreeman@mastery1: ~/examples/Chapter07/example05
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter07/example05$ ansible-playbook -i mastery-hosts error.yaml -v
No config file found; using defaults

PLAY [error handling] ****
TASK [delete branch bad] ****
fatal: [localhost]: FAILED! => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "cmd": ["git", "branch", "-D", "badfeature"], "delta": "0:00:00.004929", "end": "2021-05-31 16:26:18.008193", "msg": "non-zero return code", "rc": 1, "start": "2021-05-31 16:26:18.003264", "stderr": "error: branch 'badfeature' not found.", "stderr_lines": ["error: branch 'badfeature' not found."], "stdout": "", "stdout_lines": []}

PLAY RECAP ****
localhost                  : ok=0    changed=0    unreachable=0    failed=1    s
kipped=0      rescued=0   ignored=0
```

Figure 7.5 – Running a `git` command in an Ansible playbook with no error handling

As you can see, the error was not handled gracefully, and the play for `localhost` has been aborted.

#### Important Note

We're using the `ansible.builtin.command` module to easily demonstrate our topic despite the existence of the `ansible.builtin.git` module. When dealing with Git repositories, the `ansible.builtin.git` module should be used instead.

Without the `failed_when` and `changed_when` conditions, we would have to create a two-step task combo to protect ourselves from errors:

```
- name: check if branch badfeature exists
  ansible.builtin.command: git branch
  args:
    chdir: /srv/app
  register: branches

- name: delete branch bad
  ansible.builtin.command: git branch -D badfeature
```

```
args:
  chdir: /srv/app
when: branches.stdout is search('badfeature')
```

In the scenario where the branch doesn't exist, running these tasks should look as in *Figure 7.6*:

```
jfreeman@mastery1:~/examples/Chapter07/example06$ ansible-playbook -i mastery-hosts error.yaml -v
No config file found; using defaults

PLAY [error handling] ****
TASK [check if branch badfeature exists] ****
changed: [localhost] => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "cmd": ["git", "branch"], "delta": "0:00:00.005296", "end": "2021-05-31 16:26:47.045404", "rc": 0, "start": "2021-05-31 16:26:47.040108", "stderr": "", "stderr_lines": [], "stdout": "* master", "stdout_lines": ["* master"]}

TASK [delete branch bad] ****
skipping: [localhost] => {"changed": false, "skip_reason": "Conditional result was False"}

PLAY RECAP ****
localhost                  : ok=1    changed=1    unreachable=0    failed=0    s
kipped=1      rescued=0    ignored=0
```

Figure 7.6 – Handling errors using two tasks in an Ansible playbook

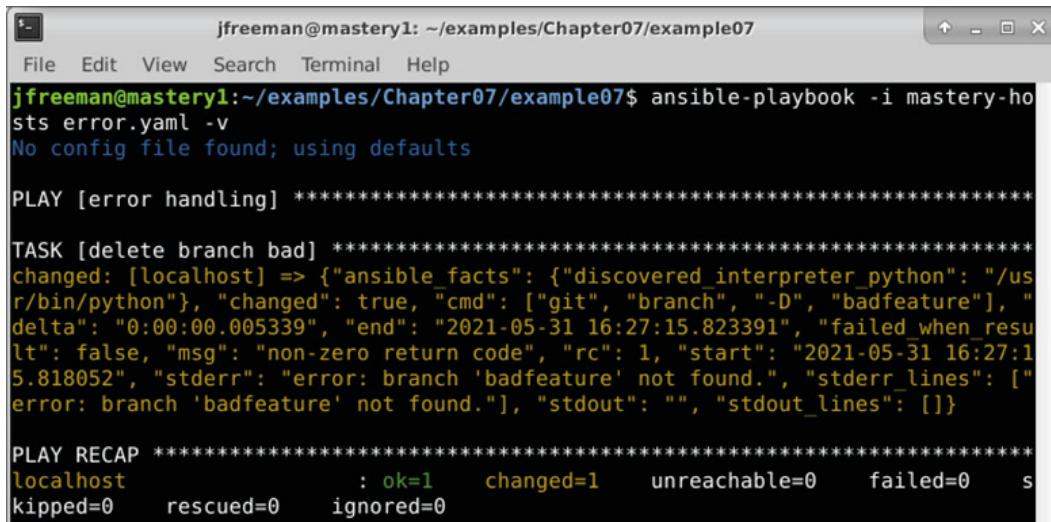
While the two-task set is functional, it is not efficient. Let's improve upon this and leverage the `failed_when` functionality to reduce the two tasks to one:

```
- name: delete branch bad
  ansible.builtin.command: git branch -D badfeature
  args:
    chdir: /srv/app
  register: gitout
  failed_when:
    - gitout.rc != 0
    - not gitout.stderr is search('branch.*not found')
```

**Important Note**

Multiple conditions that would normally be joined with and can instead be expressed as list elements. This can make playbooks easier to read and logic issues easier to find.

We check the command return code for anything other than 0 and then use the search filter to search the `stderr` value with a branch.`*not` found regex. We use Jinja2 logic to combine the two conditions, which will evaluate to an inclusive true or false option, as shown in *Figure 7.7*:



```
jfreeman@mastery1: ~/examples/Chapter07/example07
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter07/example07$ ansible-playbook -i mastery-hosts error.yaml -v
No config file found; using defaults

PLAY [error handling] ****
TASK [delete branch bad] ****
changed: [localhost] => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "cmd": ["git", "branch", "-D", "badfeature"], "delta": "0:00:00.005339", "end": "2021-05-31 16:27:15.823391", "failed_when_result": false, "msg": "non-zero return code", "rc": 1, "start": "2021-05-31 16:27:15.818052", "stderr": "error: branch 'badfeature' not found.", "stderr_lines": ["error: branch 'badfeature' not found."], "stdout": "", "stdout_lines": []}

PLAY RECAP ****
localhost                  : ok=1    changed=1    unreachable=0    failed=0    s
kipped=0      rescued=0    ignored=0
```

Figure 7.7 – Handling errors efficiently in a single task within an Ansible playbook

This demonstrates how we can redefine failure in an Ansible playbook, and gracefully handle conditions that would otherwise disrupt a play. We can also redefine what Ansible sees as a change, and we will look at this next.

## Defining a change

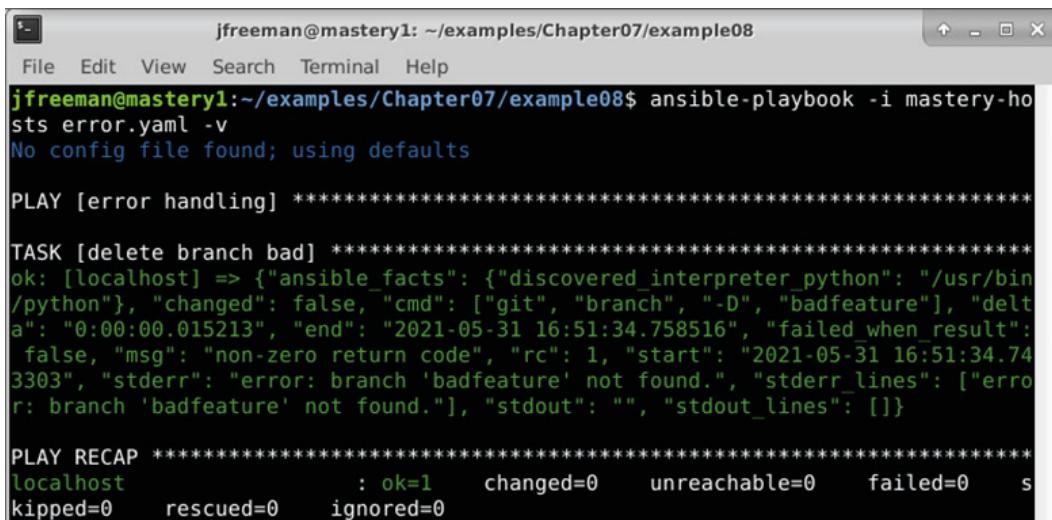
Similar to defining a task failure, it is also possible to define what constitutes a changed task result. This capability is particularly useful with the `ansible.builtin.command` family of modules (`command`, `shell`, `raw`, and `script`). Unlike most other modules, the modules of this family do not have an inherent idea of what a change may be. In fact, unless otherwise directed, these modules only result in `failed`, `changed`, or `skipped`. There is simply no way for these modules to assume a changed versus unchanged condition, as they cannot be expected to understand or interpret every possible shell command you might execute using them.

The `changed_when` condition allows a playbook author to instruct a module on how to interpret a change. Just like `failed_when`, `changed_when` performs a test to generate a Boolean result. Frequently, the tasks used with `changed_when` are commands that will exit non-zero to indicate that no work is needed to be done; so, often, authors will combine `changed_when` and `failed_when` to fine-tune the task result evaluation.

In our previous example, the `failed_when` condition caught the case where there was no work to be done but the task still showed a change. We want to register a change on the exit code 0, but not on any other exit code. Let's expand our example task to accomplish this:

```
- name: delete branch bad
  ansible.builtin.command: git branch -D badfeature
  args:
    chdir: /srv/app
  register: gitout
  failed_when:
    - gitout.rc != 0
    - not gitout.stderr is search('branch.*not found')
  changed_when: gitout.rc == 0
```

Now, if we run our task when the branch still does not exist (again adding verbosity to the output to help us see what's going on under the hood), we'll see output similar to that shown in *Figure 7.8*:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter07/example08". The window shows the command "ansible-playbook -i mastery-hosts error.yaml -v" being run. The output indicates "No config file found; using defaults". The "PLAY [error handling]" section shows a task for deleting a branch. The "ok" status is shown for the task, indicating it succeeded. The "PLAY RECAP" section at the bottom shows the host "localhost" with "ok=1", "changed=0", "unreachable=0", "failed=0", "skipped=0", "rescued=0", and "ignored=0".

```
jfreeman@mastery1:~/examples/Chapter07/example08$ ansible-playbook -i mastery-hosts error.yaml -v
No config file found; using defaults

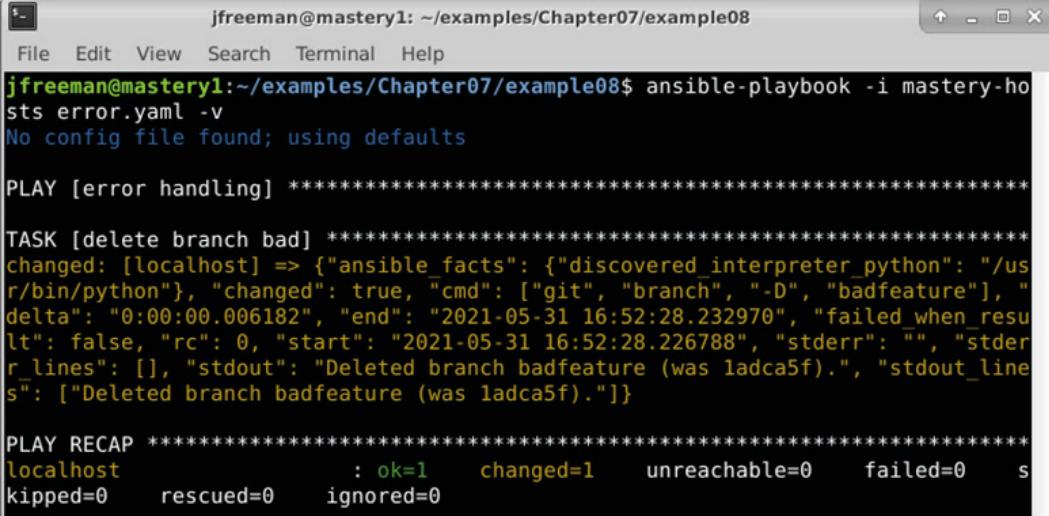
PLAY [error handling] ****
TASK [delete branch bad] ****
ok: [localhost] => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": false, "cmd": ["git", "branch", "-D", "badfeature"], "delta": "0:00:00.015213", "end": "2021-05-31 16:51:34.758516", "failed_when_result": false, "msg": "non-zero return code", "rc": 1, "start": "2021-05-31 16:51:34.743303", "stderr": "error: branch 'badfeature' not found.", "stderr_lines": ["error: branch 'badfeature' not found."], "stdout": "", "stdout_lines": []}

PLAY RECAP ****
localhost                  : ok=1      changed=0      unreachable=0      failed=0      skipped=0      rescued=0      ignored=0
```

Figure 7.8 – Extending our Git playbook with a `changed_when` task condition

Note how the changed key now has the value false.

Just for the sake of completeness, we'll change the scenario so that the branch does exist and run it again. To create the branch, simply run `git branch badfeature` from the `/srv/app` directory. Now, we can execute our playbook once again to see the output, which should now look like that shown in *Figure 7.9*:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter07/example08". The window contains the following command and its output:

```
jfreeman@mastery1:~/examples/Chapter07/example08$ ansible-playbook -i mastery-hosts error.yaml -v
No config file found; using defaults

PLAY [error handling] ****
TASK [delete branch bad] ****
changed: [localhost] => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "cmd": ["git", "branch", "-D", "badfeature"], "delta": "0:00:00.006182", "end": "2021-05-31 16:52:28.232970", "failed_when_result": false, "rc": 0, "start": "2021-05-31 16:52:28.226788", "stderr": "", "stderr_lines": [], "stdout": "Deleted branch badfeature (was 1adca5f).", "stdout_lines": ["Deleted branch badfeature (was 1adca5f)."]}

PLAY RECAP ****
localhost                  : ok=1    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 7.9 – Testing the same playbook when the badfeature branch exists in our test repository

This time, our output is different; it's registering a change, and the `stdout` data shows the branch being deleted.

## Special handling of the command family

A subset of the command family of modules (`ansible.builtin.command`, `ansible.builtin.shell`, and `ansible.builtin.script`) has a pair of special arguments that will influence whether the task work has already been done, and thus, whether or not a task will result in a change. The options are `creates` and `removes`. These two arguments expect a file path as a value. When Ansible attempts to execute a task with the `creates` or `removes` arguments, it will first check whether the referenced file path exists.

If the path exists and the `creates` argument was used, Ansible will consider that the work has already been completed and will return `ok`. Conversely, if the path does not exist and the `removes` argument is used, then Ansible will again consider the work to be complete, and it will return `ok`. Any other combination will cause the work to actually happen. The expectation is that whatever work the task is doing will result in either the creation or removal of the file that is referenced.

The convenience of `creates` and `removes` saves developers from having to do a two-task combo. Let's create a scenario where we want to run the `frobitz` script from the `files/` subdirectory of our project root. In our scenario, we know that the `frobitz` script will create a path, `/srv/whiskey/tango`. In fact, the source of `frobitz` is the following:

```
#!/bin/bash
rm -rf /srv/whiskey/tango
mkdir -p /srv/whiskey/tango
```

We don't want this script to run twice as it can be destructive to any existing data. Replacing the existing tasks in our `error.yaml` playbook, the two-task combo will look like this:

```
- name: discover tango directory
  ansible.builtin.stat: path=/srv/whiskey/tango
  register: tango

- name: run frobitz
  ansible.builtin.script: files/frobitz --initialize /srv/
  whiskey/tango
  when: not tango.stat.exists
```

Run the playbook with added verbosity as we have done throughout this chapter so far. If the /srv/whiskey/tango path already exists, the output will be as shown in *Figure 7.10*:

```
jfreeman@mastery1: ~/examples/Chapter07/example09$ ansible-playbook -v example09.yml
[WARNING]: Could not validate connection to localhost: Connection timed out.
[WARNING]: Could not validate connection to localhost: Connection timed out.

PLAY [localhost] ****
  TASK [discover tango directory] ****
  ok: [localhost] => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": false, "stat": {"atime": 1622479983.152215, "attr_flags": "e", "attributes": ["extents"], "block_size": 4096, "blocks": 8, "charset": "binary", "ctime": 1622479983.152215, "dev": 64513, "device_type": 0, "executable": true, "exists": true, "gid": 0, "gr_name": "root", "inode": 2071996, "isblk": false, "ischr": false, "isdir": true, "isfifo": false, "isgid": false, "islnk": false, "isreg": false, "issock": false, "isuid": false, "mimetype": "inode/directory", "mode": "0755", "mtime": 1622479983.152215, "nlink": 2, "path": "/srv/whiskey/tango", "pw_name": "root", "readable": true, "rgrp": true, "roth": true, "rusr": true, "size": 4096, "uid": 0, "version": "3644827379", "wgrp": false, "woth": false, "writeable": false, "wusr": true, "xgrp": true, "xoth": true, "xusr": true}}
  TASK [run frobitz] ****
  skipping: [localhost] => {"changed": false, "skip_reason": "Conditional result was False"}
  PLAY RECAP ****
  localhost : ok=1    changed=0    unreachable=0    failed=0    skipped=1   rescued=0    ignored=0

jfreeman@mastery1:~/examples/Chapter07/example09$
```

Figure 7.10 – A two-task play to conditionally run a destructive script only when necessary

If the /srv/whiskey/tango path did not exist, the `ansible.builtin.stat` module would have returned far less data, and the `exists` key would have a value of `false`. Thus, our `frobitz` script would have been run.

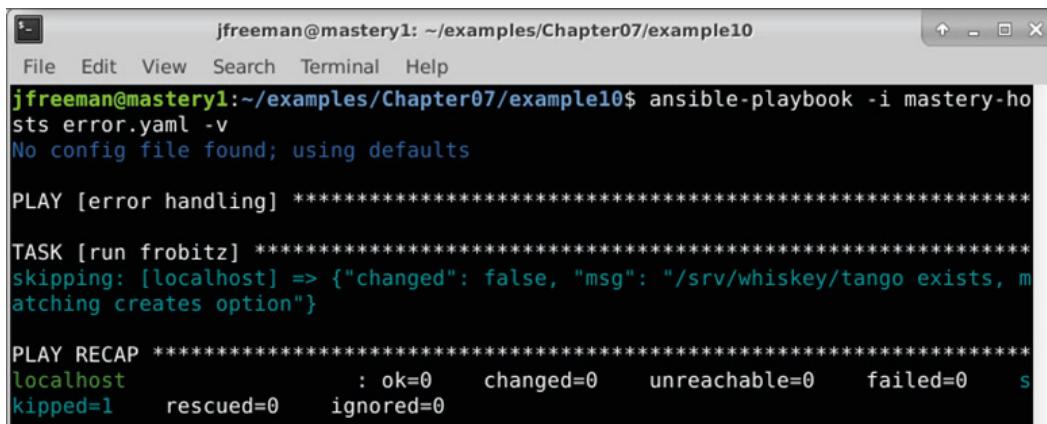
Now, we'll use `creates` to reduce this down to a single task:

```
- name: run frobitz
  ansible.builtin.script: files/frobitz
  args:
    creates: /srv/whiskey/tango
```

#### Important Note

The `ansible.builtin.script` module is actually an `action_plugin`, which will be discussed in *Chapter 10, Extending Ansible*.

This time, our output will be slightly different, as *Figure 7.11* shows:



```
jfreeman@mastery1: ~/examples/Chapter07/example10$ ansible-playbook -i mastery-hosts error.yaml -v
No config file found; using defaults

PLAY [error handling] ****
TASK [run frobitz] ****
skipping: [localhost] => {"changed": false, "msg": "/srv/whiskey/tango exists, matching creates option"}

PLAY RECAP ****
localhost                  : ok=0    changed=0    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0
```

Figure 7.11 – Making our previous playbook more efficient by combining all task conditions into one task

On this occasion, we simply skipped running the script altogether as the directory already existed before the playbook was even run. This saves time during the playbook execution and also prevents any potentially destructive actions that might occur from running a script.

#### Important Note

Making good use of `creates` and `removes` will keep your playbooks concise and efficient.

## Suppressing a change

Sometimes, it can be desirable to completely suppress changes. This is often used when executing a command in order to gather data. The command execution isn't actually changing anything; instead, it's just gathering info, like the `ansible.builtin.setup` module. Suppressing changes on such tasks can be helpful for quickly determining whether a playbook run resulted in any actual change in the fleet.

To suppress changes, simply use `false` as an argument to the `changed_when` task key. Let's extend one of our previous examples to discover the active `iscsi` sessions to suppress changes:

```
- name: discover iscsi sessions
  ansible.builtin.command: /sbin/iscsiadm -m session
  register: sessions
```

```

failed_when:
  - sessions.rc != 0
  - not sessions.stderr is
    search('No active sessions')
changed_when: false

```

Now, no matter what comes in the return data, Ansible will treat the task as `ok` rather than `changed`, as *Figure 7.12* shows:

```

jfreeman@mastery1: ~/examples/Chapter07/example11
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter07/example11$ ansible-playbook -i mastery-hosts error.yaml

PLAY [error handling] ****
TASK [discover iscsi sessions] ****
ok: [localhost]

PLAY RECAP ****
localhost : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0

```

Figure 7.12 – Suppressing changes in Ansible playbooks

Thus, there are only two possible states to this task now—`failed` and `ok`. We have actually negated the possibility of a `changed` task result. Of course, failures when running code are a part of life, and it is important that we are able to handle these gracefully in our playbooks. In the next section, we will look at how this is achieved in Ansible.

## Error recovery

While error conditions can be narrowly defined, there will be times when real errors happen. Ansible provides a method to react to true errors, a method that allows running additional tasks when an error occurs, defining specific tasks that always execute even if there was an error, or even both. This method is the **block** feature.

The blocks feature, introduced with Ansible version 2.0, provides some additional structure to related sets of play task. Blocks can group tasks together into a logical unit, which can have task controls applied to the unit (or block) as a whole. In addition, a block of tasks can have optional `rescue` and `always` sections, which execute on condition of an error and regardless of the error state, respectively. We will explore how these work in the following two sections.

## Using the rescue section

The `rescue` section of a block defines a logical unit of tasks that will be executed should an actual failure be encountered within the block. As Ansible performs the tasks within a block, executing from top to bottom as it normally does, when an actual failure is encountered, execution will jump to the first task of the `rescue` section of the block (if it exists; this section is optional). Then, tasks are performed from top to bottom until either the end of the `rescue` section is reached or another error is encountered.

After the `rescue` section completes, task execution continues with whatever comes after the block, as if there were no errors. This provides a way to gracefully handle errors, allowing cleanup tasks to be defined so that a system is not left in a completely broken state, and the rest of a play can continue. This is far cleaner than a complex set of task-registered results and task conditionals based on the error status.

To demonstrate this, let's create a new task set inside a block. This task set will have an unhandled error in it that will cause execution to switch to the `rescue` section, from where we'll perform a cleanup task.

We'll also provide a task after the block to ensure execution continues. We'll reuse the `error.yaml` playbook:

```
---
- name: error handling
  hosts: localhost
  gather_facts: false

  tasks:
    - block:
        - name: delete branch bad
          ansible.builtin.command: git branch -D badfeature
          args:
            chdir: /srv/app

        - name: this task is lost
          ansible.builtin.debug:
            msg: "I do not get seen"
```

The two tasks listed in the `block` section are executed in the order in which they are listed. Should one of them result in a `failed` result, the following code shown in the `rescue` block will be executed:

```
rescue:
  - name: cleanup task
    ansible.builtin.debug:
      msg: "I am cleaning up"

  - name: cleanup task 2
    ansible.builtin.debug:
      msg: "I am also cleaning up"
```

Finally, this task is executed regardless of the earlier tasks. Note how the lower indentation level means it gets run at the same level as the block, rather than as part of the `block` structure:

```
- name: task after block
  ansible.builtin.debug:
    msg: "Execution goes on"
```

Try executing this playbook to observe its behavior; add verbosity to the output as we have throughout this chapter to help you understand what is going on. When this play executes, the first task will result in an error, and the second task will be passed over. Execution continues with the `cleanup` tasks, and should look as in *Figure 7.13*:

```
jfreeman@mastery1: ~/examples/Chapter07/example12
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter07/example12$ ansible-playbook -i mastery-hosts error.yaml -v
No config file found; using defaults

PLAY [error handling] ****
TASK [delete branch bad] ****
fatal: [localhost]: FAILED! => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "cmd": ["git", "branch", "-D", "badfeature"], "delta": "0:00:00.008412", "end": "2021-06-02 10:22:48.387761", "msg": "non-zero return code", "rc": 1, "start": "2021-06-02 10:22:48.379349", "stderr": "error: branch 'badfeature' not found.", "stderr_lines": ["error: branch 'badfeature' not found."], "stdout": "", "stdout_lines": []}

TASK [cleanup task] ****
ok: [localhost] => {
    "msg": "I am cleaning up"
}

TASK [cleanup task 2] ****
ok: [localhost] => {
    "msg": "I am also cleaning up"
}

TASK [task after block] ****
ok: [localhost] => {
    "msg": "Execution goes on"
}

PLAY RECAP ****
localhost                  : ok=3      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=1      ignored=0
```

Figure 7.13 – Executing a playbook containing a block with a rescue section

Not only was the `rescue` section executed, but the rest of the play completed as well, and the whole `ansible-playbook` execution was considered successful in spite of the earlier task failure inside the block. Let's build on this example in the next section by looking at the `always` section of a block.

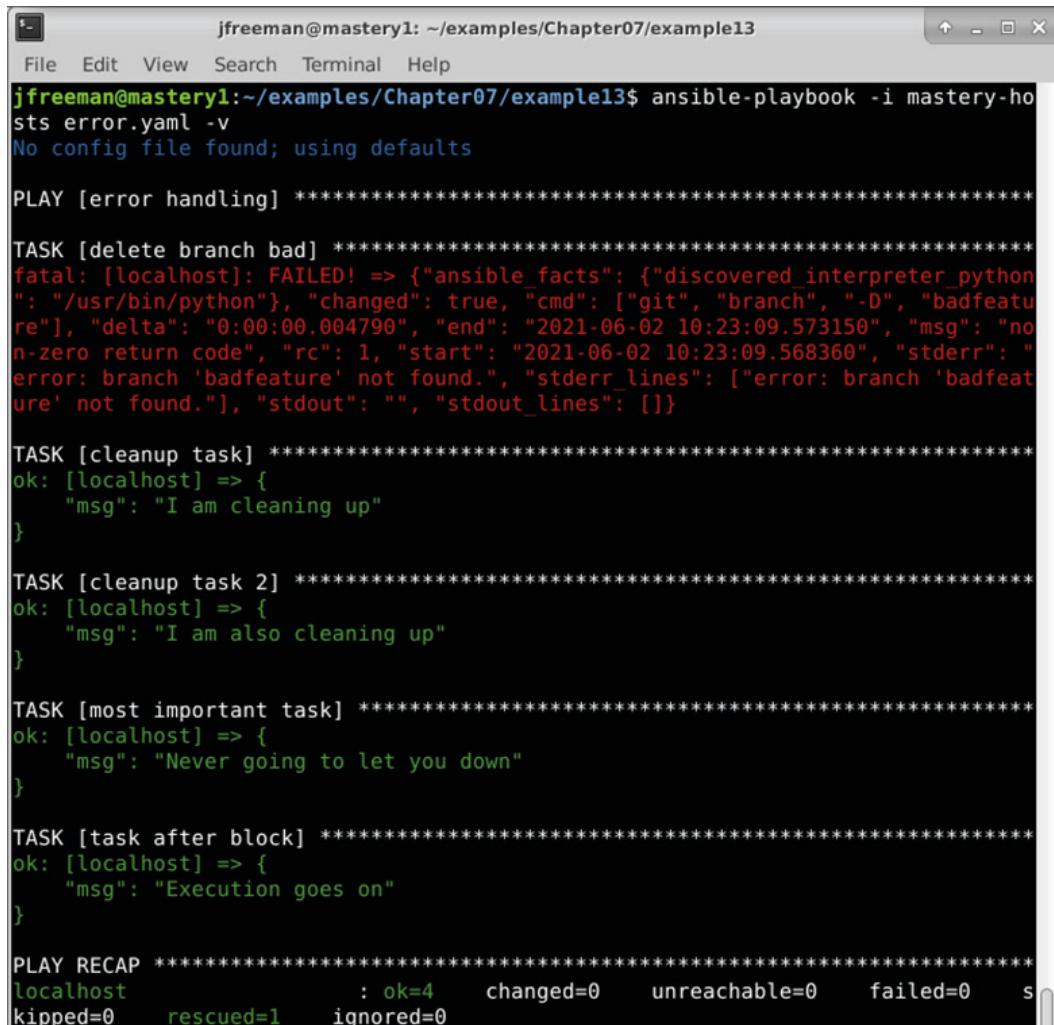
## Using the `always` section

In addition to `rescue`, we can also use another section, named `always`. This section of a block will always be executed irrespective of whether there were errors. This feature is handy for ensuring that the state of a system is always left functional, irrespective of whether a block of tasks was successful. As some tasks of a block may be skipped due to an error, and a `rescue` section is only executed when there is an error, the `always` section provides the guarantee of task execution in every instance.

Let's extend our previous example and add an `always` section to our block:

```
always:
  - name: most important task
    ansible.builtin.debug:
      msg: "Never going to let you down"
```

Rerunning our playbook as in the previous section, we see the additional task displayed, as shown in *Figure 7.14*:



```
jfreeman@mastery1:~/examples/Chapter07/example13$ ansible-playbook -i mastery-hosts error.yaml -v
No config file found; using defaults

PLAY [error handling] ****
TASK [delete branch bad] ****
fatal: [localhost]: FAILED! => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "cmd": ["git", "branch", "-D", "badfeature"], "delta": "0:00:00.004790", "end": "2021-06-02 10:23:09.573150", "msg": "non-zero return code", "rc": 1, "start": "2021-06-02 10:23:09.568360", "stderr": "error: branch 'badfeature' not found.", "stderr_lines": ["error: branch 'badfeature' not found."], "stdout": "", "stdout_lines": []}

TASK [cleanup task] ****
ok: [localhost] => {
    "msg": "I am cleaning up"
}

TASK [cleanup task 2] ****
ok: [localhost] => {
    "msg": "I am also cleaning up"
}

TASK [most important task] ****
ok: [localhost] => {
    "msg": "Never going to let you down"
}

TASK [task after block] ****
ok: [localhost] => {
    "msg": "Execution goes on"
}

PLAY RECAP ****
localhost                  : ok=4      changed=0      unreachable=0      failed=0      s
kiped=0      rescued=1      ignored=0
```

Figure 7.14 – Running an Ansible playbook containing a block with both rescue and always sections

To verify that the `always` section does indeed always execute, we can alter the play so that the Git task is considered successful using the task conditionals we developed in the earlier section, *Defining an error condition*. The first part of this modified playbook is shown in the following snippet for your reference:

```
---
- name: error handling
  hosts: localhost
  gather_facts: false

  tasks:
  - block:
    - name: delete branch bad
      ansible.builtin.command: git branch -D badfeature
      args:
        chdir: /srv/app
      register: gitout
      failed_when:
        - gitout.rc != 0
        - not gitout.stderr is search('branch.*not found')
```

Note the changed `failed_when` condition, which will enable the `git` command to run without being considered a failure. The rest of the playbook (which should, by now, have been built up in the previous examples) remains unchanged.

This time, when we execute the playbook, our `rescue` section is skipped over, our previously masked-by-error task is executed, and our `always` block is still executed, as *Figure 7.15* demonstrates:

```
jfreeman@mastery1: ~/examples/Chapter07/example14
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter07/example14$ ansible-playbook -i mastery-hosts error.yaml -v
No config file found; using defaults

PLAY [error handling] ****
TASK [delete branch bad] ****
changed: [localhost] => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "cmd": ["git", "branch", "-D", "badfeature"], "delta": "0:00:00.007357", "end": "2021-06-02 10:23:25.677735", "failed_when_result": false, "msg": "non-zero return code", "rc": 1, "start": "2021-06-02 10:23:25.670378", "stderr": "error: branch 'badfeature' not found.", "stderr_lines": ["error: branch 'badfeature' not found."], "stdout": "", "stdout_lines": []}

TASK [this task is lost] ****
ok: [localhost] => {
    "msg": "I do not get seen"
}

TASK [most important task] ****
ok: [localhost] => {
    "msg": "Never going to let you down"
}

TASK [task after block] ****
ok: [localhost] => {
    "msg": "Execution goes on"
}

PLAY RECAP ****
localhost                  : ok=4      changed=1      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 7.15 – Executing a playbook containing a block with rescue and always sections but without task errors

Note also that our previously lost task is now executed, as the failure condition for the `delete branch bad` task was changed such that it no longer fails in this play. In a similar manner, our `rescue` section is no longer needed, and all other tasks (including the `always` section) complete as expected. In the final part of our look at error recovery in Ansible, we'll see how to handle errors caused by unreliable environments.

## Handling unreliable environments

So far in this chapter, we have focused on gracefully handling errors, and changing the default behavior of Ansible with respect to changes and failures. This is all well and good for tasks, but what about if you are running Ansible in an unreliable environment? For example, poor or transient connectivity might be used to reach the managed hosts, or hosts might be down on a regular basis for some reason. The latter example might be a dynamically scaled environment that could be scaled up in times of high load and scaled back when demand is low to save on resources—hence you cannot guarantee that all hosts will be available at all times.

Luckily, a playbook keyword, `ignore_unreachable`, handles exactly these cases, and ensures that all tasks are attempted on our inventory even for hosts that get marked as unreachable during the execution of a task. This is in contrast to the default behavior where Ansible will stop processing tasks for a given host once the first error occurs. As in so many cases, this is best explained by means of an example, so let's reuse the `error.yaml` playbook to create such a case:

```
---
- name: error handling
  hosts: all
  gather_facts: false

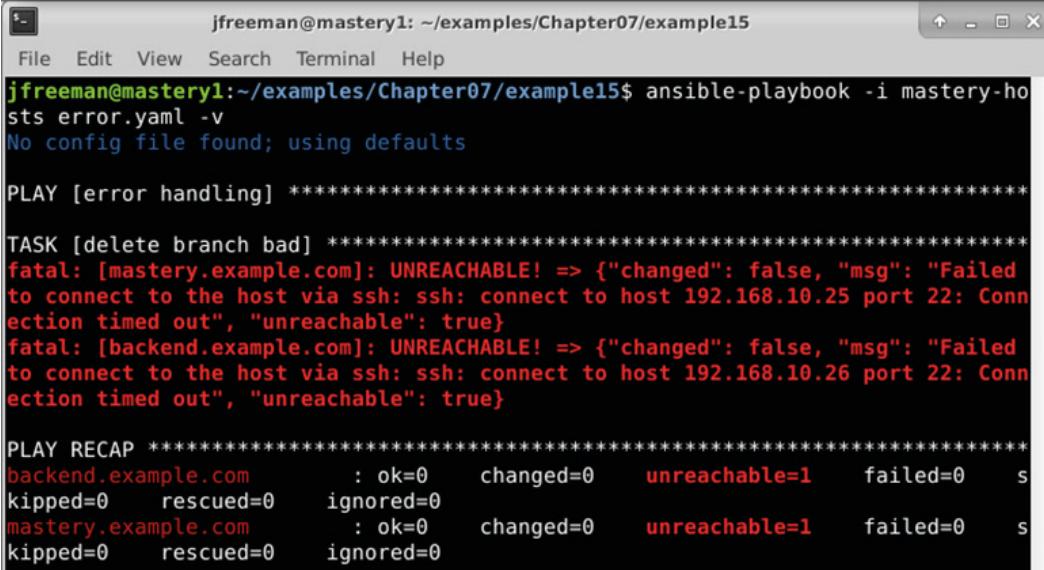
  tasks:
    - name: delete branch bad
      ansible.builtin.command: git branch -D badfeature
      args:
        chdir: /srv/app

    - name: important task
      ansible.builtin.debug:
        msg: It is important we attempt this task!
```

We are going to try to delete the `badfeature` branch from a Git repository on two remote hosts as defined in our inventory. This inventory will look a little different from the others we have used throughout this book, as we will deliberately create two fictitious hosts that are unreachable. It doesn't matter what you actually call these hosts, or what IP addresses you define, but for the example to work as described in this section, the hosts must not be reachable. My inventory file looks like this:

```
[demo]
mastery.example.com ansible_host=192.168.10.25
backend.example.com ansible_host=192.168.10.26
```

As we have deliberately created an inventory of hosts that don't exist, we know they will get marked as unreachable as soon as the first task is attempted. In spite of this, there is a second task that absolutely must be attempted if at all possible. Let's run the playbook as it is and see what happens; the output should look like that shown in *Figure 7.16*:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter07/example15". The terminal output is as follows:

```
jfreeman@mastery1:~/examples/Chapter07/example15$ ansible-playbook -i mastery-hosts error.yaml -v
No config file found; using defaults

PLAY [error handling] ****
TASK [delete branch bad] ****
fatal: [mastery.example.com]: UNREACHABLE! => {"changed": false, "msg": "Failed to connect to the host via ssh: ssh: connect to host 192.168.10.25 port 22: Connection timed out", "unreachable": true}
fatal: [backend.example.com]: UNREACHABLE! => {"changed": false, "msg": "Failed to connect to the host via ssh: ssh: connect to host 192.168.10.26 port 22: Connection timed out", "unreachable": true}

PLAY RECAP ****
backend.example.com      : ok=0    changed=0    unreachable=1    failed=0    s
kippe=0    rescued=0    ignored=0
mastery.example.com     : ok=0    changed=0    unreachable=1    failed=0    s
kippe=0    rescued=0    ignored=0
```

Figure 7.16 – Attempting a two-task play on an inventory with unreachable hosts

As you can see from the output, the task called `important_task` was never attempted—the play was aborted after the first task since the hosts were unreachable. However, let's use our newly discovered flag to change this behavior. Change the code so that it looks like the code here:

```
---  
- name: error handling
```

```

hosts: all
gather_facts: false

tasks:
- name: delete branch bad
  ansible.builtin.command: git branch -D badfeature
  args:
    chdir: /srv/app
  ignore_unreachable: true

- name: important task
  ansible.builtin.debug:
    msg: It is important we attempt this task!

```

This time, note that even though the hosts were unreachable on the first attempt, our second task is still executed, as *Figure 7.17* shows:

```

jfreeman@mastery1: ~/examples/Chapter07/example16$ ansible-playbook -i mastery-hosts error.yaml -v
No config file found; using defaults

PLAY [error handling] ****
TASK [delete branch bad] ****
fatal: [mastery.example.com]: UNREACHABLE! => {"changed": false, "msg": "Failed to connect to the host via ssh: ssh: connect to host 192.168.10.25 port 22: Connection timed out", "skip_reason": "Host mastery.example.com is unreachable", "unreachable": true}
fatal: [backend.example.com]: UNREACHABLE! => {"changed": false, "msg": "Failed to connect to the host via ssh: ssh: connect to host 192.168.10.26 port 22: Connection timed out", "skip_reason": "Host backend.example.com is unreachable", "unreachable": true}

TASK [important task] ****
ok: [mastery.example.com] => {
    "msg": "It is important we attempt this task!"
}
ok: [backend.example.com] => {
    "msg": "It is important we attempt this task!"
}

PLAY RECAP ****
backend.example.com      : ok=1    changed=0    unreachable=1    failed=0    s
kiped=1    rescued=0    ignored=0
mastery.example.com     : ok=1    changed=0    unreachable=1    failed=0    s
kiped=1    rescued=0    ignored=0

```

Figure 7.17 – Attempting the same two-task play on unreachable hosts, but this time ignoring reachability

This is useful if, like the `debug` command, it might run locally, or perhaps it is vital and should be attempted even if connectivity was down on the first attempt. So far in this chapter, you have learned about the tools Ansible provides to handle a variety of error conditions with grace. Next, we will proceed to look at controlling the flow of tasks using loops—an especially important tool for making code concise and preventing repetition.

## Iterative tasks with loops

Loops deserve a special mention in this chapter. So far, we have focused on controlling the flow of a playbook in a top-to-bottom fashion—we have changed the various conditions that might be evaluated as the playbook runs, and we have also focused on creating concise, efficient code. What happens, however, if you have a single task, but need to run it against a list of data; for example, creating several user accounts, directories, or indeed something more complex?

Looping changed in Ansible 2.5—prior to this, loops were generally created with keywords such as `with_items` and you may still see this in legacy code. Although some backward compatibility remains, it is advisable to move to the newer `loop` keyword instead.

Let's take a simple example—we need to create two directories. Create `loop.yaml` as follows:

```
---
```

```
- name: looping demo
  hosts: localhost
  gather_facts: false
  become: true
```

```

  tasks:
    - name: create a directory
      ansible.builtin.file:
        path: /srv/whiskey/alpha
        state: directory
```

```

    - name: create another directory
      ansible.builtin.file:
        path: /srv/whiskey/beta
        state: directory
```

When we run this, as expected, our two directories get created, as *Figure 7.18* shows:



```
jfreeman@mastery1: ~/examples/Chapter07/example17$ ansible-playbook -i mastery-hosts loop.yaml

PLAY [looping demo] ****
TASK [create a directory] ****
changed: [localhost]

TASK [create another directory] ****
changed: [localhost]

PLAY RECAP ****
localhost                  : ok=2    changed=2    unreachable=0    failed=0    s
kipped=0      rescued=0    ignored=0
```

Figure 7.18 – Running a simple playbook to create two directories

However, you can see this code is repetitive and inefficient. Instead, we could change it to something like this:

```
---
- name: looping demo
  hosts: localhost
  gather_facts: false
  become: true

  tasks:
    - name: create a directory
      ansible.builtin.file:
        path: "{{ item }}"
        state: directory
    loop:
      - /srv/whiskey/alpha
      - /srv/whiskey/beta
```

Note the use of the special `item` variable, which is now used to define the path from the loop items at the bottom of the task. Now, when we run this code, the output looks somewhat different, as *Figure 7.19* shows:

```
jfreeman@mastery1:~/examples/Chapter07/example18$ ansible-playbook -i mastery-hosts loop.yaml

PLAY [looping demo] ****
TASK [create a directory] ****
changed: [localhost] => (item=/srv/whiskey/alpha)
changed: [localhost] => (item=/srv/whiskey/beta)

PLAY RECAP ****
localhost : ok=1    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 7.19 – A playbook to create the same two directories, but this time using a loop for more efficient code

The two directories were still created exactly as before, but this time within a single task. This makes our playbooks much more concise and efficient. Ansible offers many more powerful looping options, including nested loops and the ability to create loops that will carry on until a given criterion is met (often referred to as `do until` loops in other languages), as opposed to a specific limited set of data.

`do until` loops are incredibly useful when waiting for a certain condition to be met. For example, if we wanted to wait until a flag file is written to the filesystem, we could use the `ansible.builtin.stat` module to query the file, register the result of the module run to a variable, and then run this in a loop until the condition that the file exists is met. The following code fragment shows exactly this—it will loop (`retries`) five times, with a 10-second delay between each retry:

```
- name: Wait until /tmp/flag exists
  ansible.builtin.stat:
    path: /tmp/flag
    register: statresult
    until: statresult.stat.exists
    retries: 5
    delay: 10
```

Nested loops can be created in one of two ways—either by iterating over nested lists or by iterating over an included tasks file. For example, let's assume we want to create two new files, each in two paths (as defined by two lists in Ansible). Our code might look like this:

```
---
- name: Nested loop example
  hosts: all
  gather_facts: no
  vars:
    paths:
      - /tmp
      - /var/tmp
    files:
      - test1
      - test2

  tasks:
    - name: Create files with nested loop
      ansible.builtin.file:
        path: "{{ item[0] }}/{{ item[1] }}"
        state: touch
      loop: "{{ paths | product(files) | list }}"
```

Here, we have used the `product` Jinja2 filter to create a nested list out of the two variable lists, which `loop` then faithfully iterates for us. Running this playbook should yield output that looks like that in *Figure 7.20*:



The terminal window shows the command `jfreeman@mastery1:~/examples/Chapter07/example19$ ansible-playbook -i mastery-hosts nestedloop.yaml` being run. The output shows the playbook executing a task to create files with a nested loop, resulting in four files being created: /tmp/test1, /tmp/test2, /var/tmp/test1, and /var/tmp/test2. The final recap summary shows 1 host with 1 ok, 1 changed, 0 unreachable, 0 failed, 0 skipped, 0 rescued, and 0 ignored.

```
jfreeman@mastery1:~/examples/Chapter07/example19$ ansible-playbook -i mastery-hosts nestedloop.yaml

PLAY [Nested loop demo 1] ****
TASK [Create files with nested loop] ****
changed: [localhost] => (item=['/tmp', 'test1'])
changed: [localhost] => (item=['/tmp', 'test2'])
changed: [localhost] => (item=['/var/tmp', 'test1'])
changed: [localhost] => (item=['/var/tmp', 'test2'])

PLAY RECAP ****
localhost                  : ok=1    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 7.20 – Running a playbook using a nested loop built with the `product` Jinja2 filter

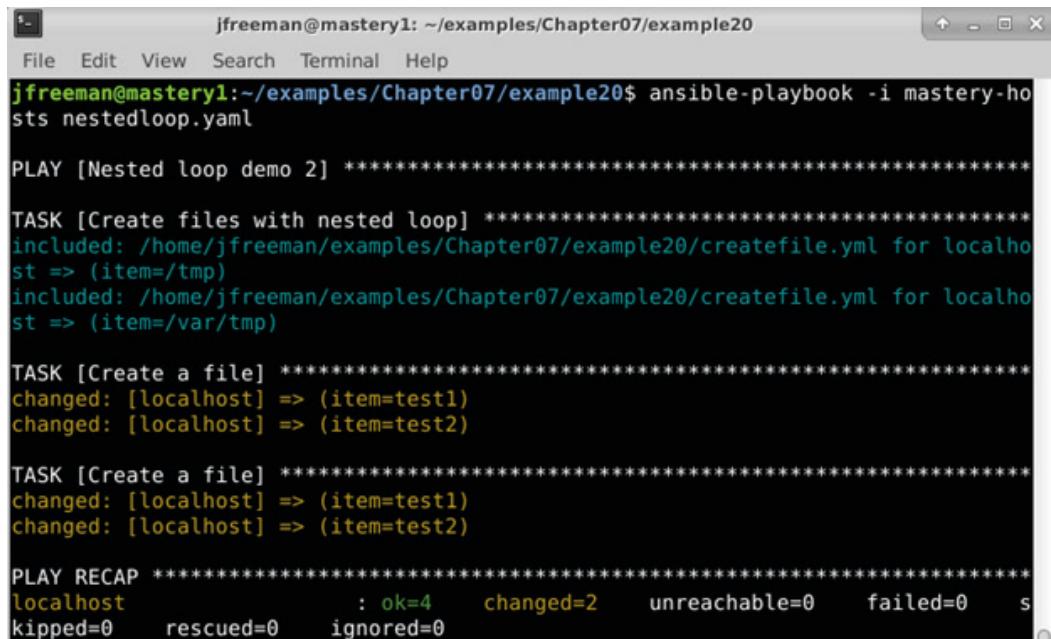
You can also create a nested loop by including an external tasks file within an outer loop and then placing an inner loop within the tasks file. Now, if you do this without doing anything further, both loops will use the `item` loop variable, which of course will clash. To prevent this from being an issue, it is necessary to use one of the special `loop_control` parameters to change the loop variable name for the outer loop. Thus, using the same header code and variables as before, we could change our original task to the following:

```
- name: Create files with nested loop
  ansible.builtin.include_tasks: createfile.yml
  loop: "{{ paths }}"
  loop_control:
    loop_var: pathname
```

The included tasks file would then look like this:

```
---
- name: Create a file
  ansible.builtin.file:
    path: "{{ pathname }}/{{ item }}"
    state: touch
  loop: "{{ files }}"
```

This code performs exactly the same function as the first nested loop example, but is a little more cumbersome as it requires an external tasks file. In addition, you will see from the screenshot in *Figure 7.21* that the way that it operates is somewhat different. This is important to factor in when you are building nested loops as this may (or may not) be what you want:

A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter07/example20". The window shows the execution of an Ansible playbook named "nestedloop.yaml". The output indicates a nested loop where two tasks are run for each item in two lists: "/tmp" and "/var/tmp". The first task creates files "test1" and "test2" in each directory. The second task creates a file "test3" in each directory. The final PLAY RECAP summary shows 4 tasks completed successfully (ok=4), 2 changed (changed=2), and 0 failed or unreachable. All other metrics (skipped, rescued, ignored) are 0.

```
jfreeman@mastery1:~/examples/Chapter07/example20$ ansible-playbook -i mastery-hosts nestedloop.yaml

PLAY [Nested loop demo 2] ****
TASK [Create files with nested loop] ****
included: /home/jfreeman/examples/Chapter07/example20/createfile.yml for localhost => (item=/tmp)
included: /home/jfreeman/examples/Chapter07/example20/createfile.yml for localhost => (item=/var/tmp)

TASK [Create a file] ****
changed: [localhost] => (item=test1)
changed: [localhost] => (item=test2)

TASK [Create a file] ****
changed: [localhost] => (item=test1)
changed: [localhost] => (item=test2)

PLAY RECAP ****
localhost                  : ok=4    changed=2    unreachable=0    failed=0    s
kipped=0      rescued=0      ignored=0
```

Figure 7.21 – Building nested loops in Ansible through an included tasks file, using the `loop_control` variable

It could be said that it's easier to read this format, though, and ultimately it is up to you to determine which you prefer for your needs, and indeed whether one is more suitable for you than the other. Full details of loop creation techniques and parameters are available in the Ansible documentation here: [https://docs.ansible.com/ansible/latest/user\\_guide/playbooks\\_loops.html](https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html).

## Summary

In this chapter, you learned that it is possible to define specifically how Ansible perceives a failure or a change when a specific task is run, how to use blocks to gracefully handle errors and perform cleanup, and how to write tight, efficient code using loops.

As a result, you should now be able to alter any given task to provide specific conditions under which Ansible will fail it or consider a change successful. This is incredibly valuable when running shell commands, as we have demonstrated in this chapter, and also serves when defining specialized use cases for existing modules. You should also now be able to organize your Ansible tasks into blocks, ensuring that if failures do occur, recovery actions can be taken that would otherwise not need to be run. Finally, you should now be able to write tight, efficient Ansible playbooks using loops, removing the need for repetitive code and lengthy, inefficient playbooks.

In the next chapter, we'll explore the use of roles for organizing tasks, files, variables, and other content.

## Questions

1. By default, Ansible will stop processing further tasks for a given host after the first failure occurs:
  - a) True
  - b) False
2. The `ansible.builtin.command` and `ansible.builtin.shell` modules' default behavior is to only ever give a task status of `changed` or `failed`:
  - a) True
  - b) False
3. You can store the results from a task using which Ansible keyword?
  - a) `store:`
  - b) `variable:`
  - c) `register:`
  - d) `save:`
4. Which of the following directives can be used to change the failure condition of a task?
  - a) `error_if:`
  - b) `failed_if:`
  - c) `error_when:`
  - d) `failed_when:`
5. You can combine multiple conditional statements in Ansible using which of the following?
  - a) `and`
  - b) `or`
  - c) The YAML list format (which works the same as a logical AND)
  - d) All of the above

6. Changes can be suppressed with which of the following?
  - a) suppress\_changed: true
  - b) changed\_when: false
  - c) changed: false
  - d) failed\_when: false
7. In a block section, all tasks are executed in order on all hosts:
  - a) Until the first error occurs
  - b) Regardless of any error condition
8. Which optional section of a block gets run only if an error occurs in the block tasks?
  - a) recover
  - b) rescue
  - c) always
  - d) on\_error
9. Tasks in the always section of a block are run:
  - a) Regardless of what happened in either the block tasks or the rescue section
  - b) Only if the rescue section did not get run
  - c) Only if no errors were encountered
  - d) When called manually by the user
10. The default name of the variable referencing the current element of a loop is:
  - a) loopvar
  - b) loopitem
  - c) item
  - d) val



# 8

# Composing Reusable Ansible Content with Roles

For many projects, a simple, single **Ansible** playbook may suffice. As time goes on and projects grow, additional playbooks and variable files are added, and task files may be split. Other projects within an organization may want to reuse some of the content, and either the projects get added to the directory tree or the desired content may get copied across multiple projects. As the complexity and size of the scenario grow, something more than a loosely organized handful of playbooks, task files, and variable files is highly desired. Creating such a hierarchy can be daunting and may explain why many Ansible implementations start off simple and only become more organized once the scattered files become unwieldy and a hassle to maintain. Making the migration can be difficult and may require rewriting significant portions of playbooks, which can further delay reorganization efforts.

In this chapter, we will cover best practices for composable, reusable, and well-organized content within Ansible. The lessons learned in this chapter will help developers design Ansible content that grows well with the project, avoiding the need for difficult redesign work later. The following is an outline of what we will cover:

- Task, handler, variable, and playbook inclusion concepts
- Roles (structures, defaults, and dependencies)
- Designing top-level playbooks to utilize roles
- Sharing roles across projects (dependencies via Galaxy; Git-like repositories)

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running **Ansible 4.3** or newer. Almost any flavor of Linux should do—for those interested in specifics, all the code presented in this chapter was tested on **Ubuntu Server 20.04 Long-Term Support (LTS)** unless stated otherwise, and on Ansible 4.3.

The example code that accompanies this chapter can be downloaded from GitHub at this link: <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter08>.

Check out the following video to see the Code in Action: <https://bit.ly/3E0mmIX>.

## Task, handler, variable, and playbook inclusion concepts

The first step to understanding how to efficiently organize an Ansible project structure is to master the concept of including files. The act of including files allows content to be defined in a topic-specific file that can be included in other files one or more times within a project. This inclusion feature supports the concept of **Don't Repeat Yourself (DRY)**.

## Including tasks

Task files are **YAML Ain't Markup Language (YAML)** files that define one or more tasks. These tasks are not directly tied to any particular play or playbook; they exist purely as a list of tasks. These files can be referenced by **playbooks** or other task files by way of the `include` operator. Now, you might expect the `include` operator to be a keyword of Ansible in its own right—however, this is not the case; it is actually a module just like `ansible.builtin.debug`. For conciseness, we will refer to it in this chapter as the `include` operator, but when we say this, your code will actually contain the **Fully Qualified Collection Name (FQCN)**—see *Chapter 2, Migrating from Earlier Ansible Versions*, which is `ansible.builtin.include`. You'll see this in action very shortly, so don't worry—this will all make sense soon! This operator takes a path to a task file, and as we learned in *Chapter 1, The System Architecture and Design of Ansible*, the path can be relative to the file referencing it.

To demonstrate how to use the `include` operator to include tasks, let's create a simple play that includes a task file with some debug tasks within it. First, let's write our playbook file, which we'll call `includer.yaml`, as follows:

```
---
- name: task inclusion
  hosts: localhost
  gather_facts: false

  tasks:
    - name: non-included task
      ansible.builtin.debug:
        msg: "I am not included"
    - ansible.builtin.include: more-tasks.yaml
```

Next, we'll create a `more-tasks.yaml` file you can see referenced in the `include` statement. This should be created in the same directory that holds `includer.yaml`. The code is illustrated in the following snippet:

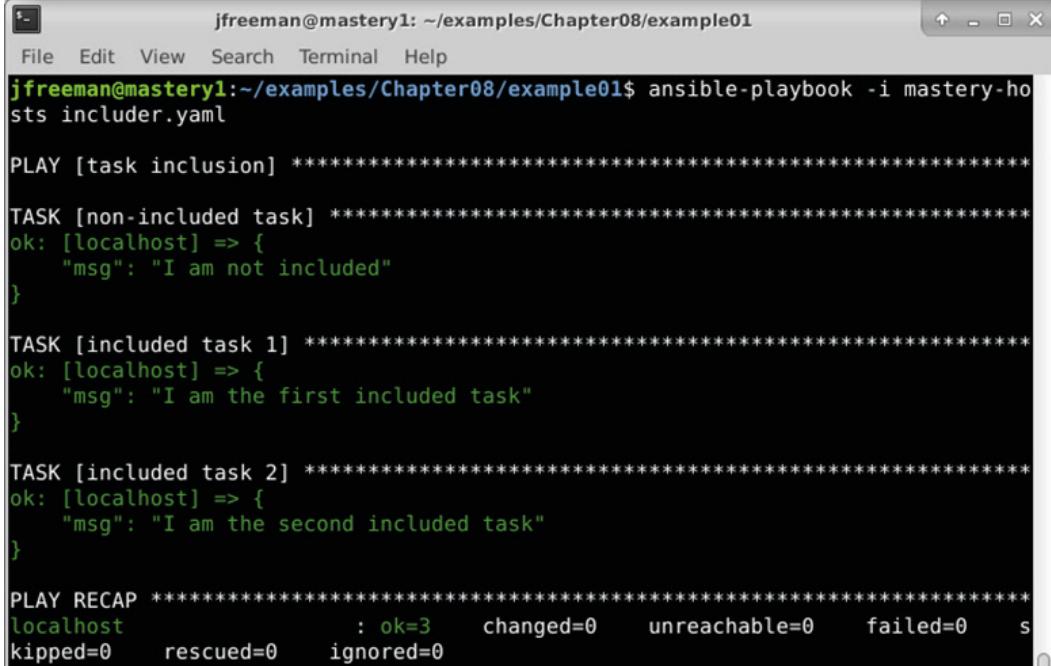
```
---
- name: included task 1
  ansible.builtin.debug:
    msg: "I am the first included task"

- name: included task 2
  ansible.builtin.debug:
    msg: "I am the second included task"
```

Now, we can execute our playbook with the following command to observe the output:

```
ansible-playbook -i mastery-hosts includer.yaml
```

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



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter08/example01". The user runs the command "ansible-playbook -i mastery-hosts includer.yaml". The output shows the execution of the playbook, including the inclusion of the "more-tasks.yaml" file, which contains two debug tasks. The first task outputs "I am the first included task" and the second outputs "I am the second included task". Finally, the PLAY RECAP summary shows 3 tasks completed successfully (ok=3) with 0 changes, 0 unreachable hosts, 0 failed hosts, 0 skipped hosts, 0 rescued hosts, and 0 ignored hosts.

```
jfreeman@mastery1: ~/examples/Chapter08/example01
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example01$ ansible-playbook -i mastery-hosts includer.yaml

PLAY [task inclusion] ****
TASK [non-included task] ****
ok: [localhost] => {
    "msg": "I am not included"
}

TASK [included task 1] ****
ok: [localhost] => {
    "msg": "I am the first included task"
}

TASK [included task 2] ****
ok: [localhost] => {
    "msg": "I am the second included task"
}

PLAY RECAP ****
localhost                  : ok=3      changed=0      unreachable=0      failed=0      s
kippe
d=0      rescued=0      ignored=0
```

Figure 8.1 – Executing an Ansible playbook that includes a separate task file

We can clearly see our tasks from the `include` file execution. Because the `include` operator was used within the play's `tasks` section, the included tasks were executed within that play. In fact, if we were to add a task to the play after the `include` operator, as illustrated in the following code snippet, we would see that the order of execution follows as if all the tasks from the included file existed at the spot the `include` operator was used:

```
tasks:
  - name: non-included task
    ansible.builtin.debug:
      msg: "I am not included"

  - ansible.builtin.include: more-tasks.yaml

  - name: after-included tasks
    ansible.builtin.debug:
      msg: "I run last"
```

If we run our modified playbook using the same command as before, we will see the task order we expect, as the following screenshot demonstrates:

```
jfreeman@mastery1: ~/examples/Chapter08/example02$ ansible-playbook -i mastery-hosts includer.yaml

PLAY [task inclusion] *****
TASK [non-included task] *****
ok: [localhost] => {
    "msg": "I am not included"
}

TASK [included task 1] *****
ok: [localhost] => {
    "msg": "I am the first included task"
}

TASK [included task 2] *****
ok: [localhost] => {
    "msg": "I am the second included task"
}

TASK [after-included tasks] *****
ok: [localhost] => {
    "msg": "I run last"
}

PLAY RECAP *****
localhost                  : ok=4    changed=0    unreachable=0    failed=0
                           skipped=0   rescued=0   ignored=0
```

Figure 8.2 – Demonstrating the order of task execution in a playbook that uses the `include` operator

By breaking these tasks into their own file, we could include them multiple times or in multiple playbooks. If we ever have to alter one of the tasks, we only have to alter a single file, no matter how many places this file gets referenced.

## Passing variable values to included tasks

Sometimes, we want to split out a set of tasks but have those tasks act slightly differently depending on the variable data. The `include` operator allows us to define and override variable data at the time of inclusion. The scope of the definition is only within the included task file (and any other files that file may itself include).

To illustrate this capability, let's create a new scenario in which we need to touch a couple of files, each in their own directory path. Instead of writing two file tasks for each file (one to create a directory and another to touch the file), we'll create a task file with each task that will use variable names in the tasks. Then, we'll include the task file twice, each time passing different data in. First, we'll do this with the `files.yaml` task file, as follows:

```
---
```

```
- name: create leading path
  ansible.builtin.file:
    path: "{{ path }}"
    state: directory
```

```
- name: touch the file
  ansible.builtin.file:
    path: "{{ path + '/' + file }}"
    state: touch
```

Next, we'll modify our `includer.yaml` playbook to include the task file we've just created, passing along variable data for the `path` and `file` variables, as follows:

```
---
- name: touch files
  hosts: localhost
  gather_facts: false

  tasks:
    - ansible.builtin.include: files.yaml
      vars:
        path: /tmp/foo
        file: herp

    - ansible.builtin.include: files.yaml
      vars:
        path: /tmp/foo
        file: derp
```

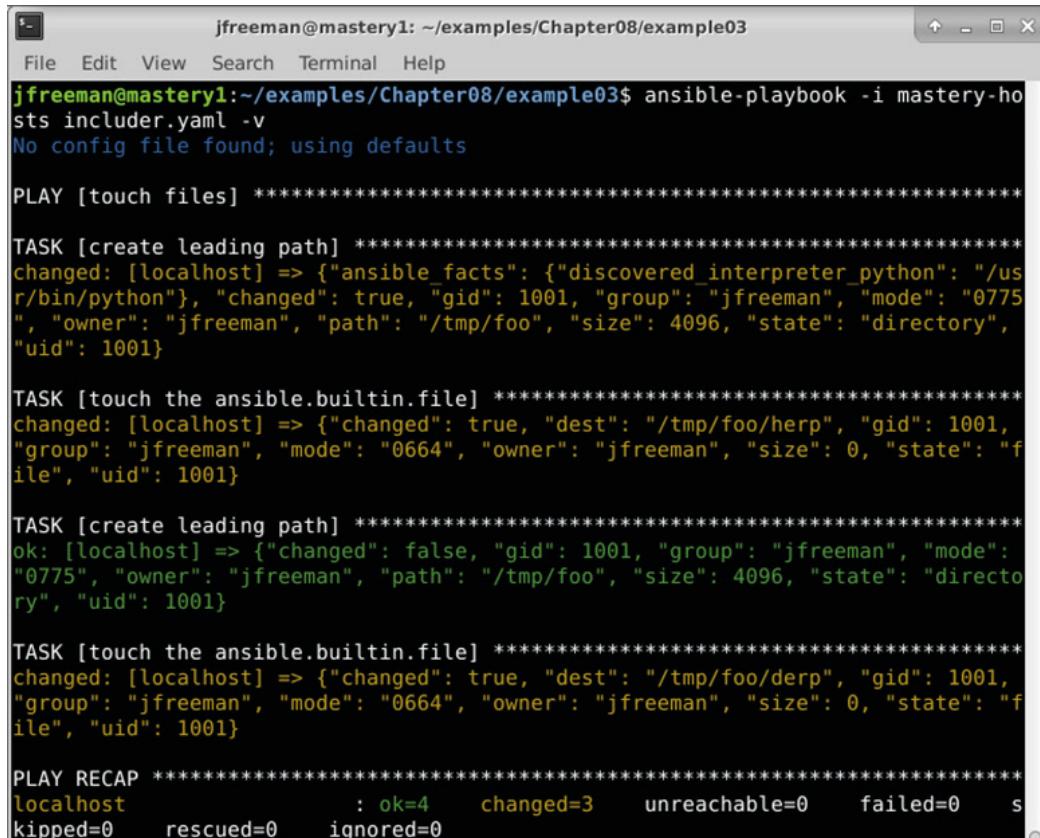
#### Important Note

Variable definitions provided when including files can either be in the inline format of `key=value` or in the illustrated YAML format of `key: value` inside a `vars` hash.

When we run this playbook, we'll see four tasks get executed: the two tasks from within the included `files.yaml` file twice. The second set should result in only one change as the path is the same for both sets, and should be created the first time the task is executed. Run the playbook with added verbosity so that we can see a little more about what's going on under the hood, by using the following command:

```
ansible-playbook -i mastery-hosts includer.yaml -v
```

The output from running this playbook should look something like this:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter08/example03". The window shows the command "ansible-playbook -i mastery-hosts includer.yaml -v" being run. The output displays the execution of a playbook with four tasks. The first task creates a directory at "/tmp/foo/herp" with owner "jfreeman", group "jfreeman", mode "0664", and size 0. The second task creates a file at "/tmp/foo/herp" with the same ownership and permissions. The third task creates a directory at "/tmp/foo/derp" with the same ownership and permissions. The fourth task creates a file at "/tmp/foo/derp" with the same ownership and permissions. The final "PLAY RECAP" summary shows 4 hosts managed, with 4 tasks successful (ok=4), 3 tasks changed, 0 unreachable, 0 failed, 0 skipped, 0 rescued, and 0 ignored.

```
jfreeman@mastery1:~/examples/Chapter08/example03$ ansible-playbook -i mastery-hosts includer.yaml -v
No config file found; using defaults

PLAY [touch files] *****

TASK [create leading path] *****
changed: [localhost] => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "gid": 1001, "group": "jfreeman", "mode": "0775", "owner": "jfreeman", "path": "/tmp/foo", "size": 4096, "state": "directory", "uid": 1001}

TASK [touch the ansible.builtin.file] *****
changed: [localhost] => {"changed": true, "dest": "/tmp/foo/herp", "gid": 1001, "group": "jfreeman", "mode": "0664", "owner": "jfreeman", "size": 0, "state": "file", "uid": 1001}

TASK [create leading path] *****
ok: [localhost] => {"changed": false, "gid": 1001, "group": "jfreeman", "mode": "0775", "owner": "jfreeman", "path": "/tmp/foo", "size": 4096, "state": "directory", "uid": 1001}

TASK [touch the ansible.builtin.file] *****
changed: [localhost] => {"changed": true, "dest": "/tmp/foo/derp", "gid": 1001, "group": "jfreeman", "mode": "0664", "owner": "jfreeman", "size": 0, "state": "file", "uid": 1001}

PLAY RECAP *****
localhost                  : ok=4      changed=3      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 8.3 – Running a playbook where a task file is included twice with different variable data

As can be seen here, the code to create a leading path and a file is being reused, just with different values each time, making our code really efficient to maintain.

## Passing complex data to included tasks

When wanting to pass complex data to included tasks, such as a list or hash, an alternative syntax can be used when including the file. Let's repeat the previous scenario, only this time instead of including the task file twice, we'll include it once and pass a hash of the paths and files in. First, we'll recreate the `files.yaml` file, as follows:

```
---
- name: create leading path
  ansible.builtin.file:
    path: "{{ item.value.path }}"
    state: directory
  loop: "{{ files | dict2items }}"

- name: touch the file
  ansible.builtin.file:
    path: "{{ item.value.path + '/' + item.key }}"
    state: touch
  loop: "{{ files | dict2items }}"
```

Now, we'll alter our `includer.yaml` playbook to provide the file's hash in a single `ansible.builtin.include` statement, as follows:

```
---
- name: touch files
  hosts: localhost
  gather_facts: false

  tasks:
    - ansible.builtin.include: files.yaml
      vars:
        files:
          herp:
            path: /tmp/foo
          derp:
            path: /tmp/foo
```

If we run this new playbook and task file as before, we should see a similar but slightly different output, the end result of which is the `/tmp/foo` directory already in place and the two `herp` and `derp` files being created as empty files (touched) within, as the following screenshot shows:

```
jfreeman@mastery1:~/examples/Chapter08/example04$ ansible-playbook -i mastery-hosts includer.yaml -v
No config file found; using defaults

PLAY [touch files] ****
TASK [create leading path] ****
changed: [localhost] => (item={'key': 'herp', 'value': {'path': '/tmp/foo'}}) =>
{"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "ansible_loop_var": "item", "changed": true, "deprecations": [{"msg": "Distribution Ubuntu 20.04 on host localhost should use /usr/bin/python3, but is using /usr/bin/python for backward compatibility with prior Ansible releases. A future Ansible release will default to using the discovered platform python for this host. See https://docs.ansible.com/ansible/2.11/reference_appendices/interpreter_discovery.html for more information", "version": "2.12"}], "gid": 1001, "group": "jfreeman", "item": {"key": "herp", "value": {"path": "/tmp/foo"}}, "mode": "0775", "owner": "jfreeman", "path": "/tmp/foo", "size": 4096, "state": "directory", "uid": 1001}
ok: [localhost] => (item={'key': 'derp', 'value': {'path': '/tmp/foo'}}) => {"ansible_loop_var": "item", "changed": false, "gid": 1001, "group": "jfreeman", "item": {"key": "derp", "value": {"path": "/tmp/foo"}}, "mode": "0775", "owner": "jfreeman", "path": "/tmp/foo", "size": 4096, "state": "directory", "uid": 1001}

TASK [touch the file] ****
changed: [localhost] => (item={'key': 'herp', 'value': {'path': '/tmp/foo'}}) =>
{"ansible_loop_var": "item", "changed": true, "dest": "/tmp/foo/herp", "gid": 1001, "group": "jfreeman", "item": {"key": "herp", "value": {"path": "/tmp/foo"}}, "mode": "0664", "owner": "jfreeman", "size": 0, "state": "file", "uid": 1001}
changed: [localhost] => (item={'key': 'derp', 'value': {'path': '/tmp/foo'}}) =>
{"ansible_loop_var": "item", "changed": true, "dest": "/tmp/foo/derp", "gid": 1001, "group": "jfreeman", "item": {"key": "derp", "value": {"path": "/tmp/foo"}}, "mode": "0664", "owner": "jfreeman", "size": 0, "state": "file", "uid": 1001}

PLAY RECAP ****
localhost                  : ok=2    changed=2    unreachable=0    failed=0    s
kipped=0      rescued=0   ignored=0
```

Figure 8.4 – Passing complex data to an included task file in an Ansible play

Using this manner of passing in a hash of data allows the growth of a set of things created without having to grow the number of `include` statements in the main playbook.

## Conditional task includes

Similar to passing data into included files, conditionals can also be passed into included files. This is accomplished by attaching a `when` statement to the `include` operator. This conditional does not cause Ansible to evaluate the test to determine whether the file should be included; rather, it instructs Ansible to add the conditional to each and every task within the included file (and any other files the said file may include).

**Important Note**

It is not possible to conditionally include a file. Files will always be included; however, a task condition can be applied to every task within.

Let's demonstrate this by modifying our first example that includes simple debug statements. We'll add a conditional and pass along some data for the conditional to use. First, let's modify the `includer.yaml` playbook, as follows:

```
---
- name: task inclusion
  hosts: localhost
  gather_facts: false

  tasks:
    - ansible.builtin.include: more-tasks.yaml
      when: item | bool
  vars:
    a_list:
      - true
      - false
```

Next, let's modify `more-tasks.yaml` to loop over the `a_list` variable in each task, like this:

```
---
- name: included task 1
  ansible.builtin.debug:
    msg: "I am the first included task"
  loop: "{{ a_list }}"

- name: include task 2
  ansible.builtin.debug:
    msg: "I am the second included task"
  loop: "{{ a_list }}"
```

Now, let's run the playbook with the same command as before and see our new output, which should look like this:

```
jfreeman@mastery1: ~/examples/Chapter08/example05
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example05$ ansible-playbook -i mastery-hosts includer.yaml -v
No config file found; using defaults

PLAY [task inclusion] ****
TASK [included task 1] ****
ok: [localhost] => (item=True) => {
    "msg": "I am the first included task"
}
skipping: [localhost] => (item=False) => {"ansible_loop_var": "item", "item": false}

TASK [include task 2] ****
ok: [localhost] => (item=True) => {
    "msg": "I am the second included task"
}
skipping: [localhost] => (item=False) => {"ansible_loop_var": "item", "item": false}

PLAY RECAP ****
localhost                  : ok=2      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 8.5 – Applying conditionals to all tasks in an included file

We can see a skipped iteration per task, the iteration where `item` was evaluated to a `false` Boolean. It's important to remember that all hosts will evaluate all included tasks. There is no way to influence Ansible to not include a file for a subset of hosts. At most, a conditional can be applied to every task within an `include` hierarchy so that included tasks may be skipped. One method to include tasks based on host facts is to utilize the `ansible.builtin.group_by` action plugin to create dynamic groups based on host facts. Then, you can give the groups their own plays to include specific tasks. This is an exercise left up to you.

## Tagging included tasks

When including task files, it is possible to tag all tasks within the file. The `tags` key is used to define one or more tags to apply to all tasks within the `include` hierarchy. The ability to tag at `include` time can keep the task file itself unopinionated about how the tasks should be tagged and can allow for a set of tasks to be included multiple times but with different data and tags passed along.

**Important Note**

Tags can be defined at the `include` statement or at the play itself to cover all includes (and other tasks) in a given play.

Let's create a simple demonstration to illustrate how tags can be used. We'll start by editing our `includer.yaml` file to create a playbook that includes a task file twice, each with a different tag name and different variable data. The code is illustrated in the following snippet:

```
---
```

```
- name: task inclusion
  hosts: localhost
  gather_facts: false
```

```

  tasks:
    - ansible.builtin.include: more-tasks.yaml
      vars:
        data: first
      tags: first
```

```

    - ansible.builtin.include: more-tasks.yaml
      vars:
        data: second
      tags: second
```

Now, we'll update `more-tasks.yaml` to do something with the data being provided, as follows:

```
---
```

```
- name: included task
  ansible.builtin.debug:
    msg: "My data is {{ data }}"
```

If we run this playbook without selecting tags, we'll see this task run twice, as the following screenshot shows:

```
jfreeman@mastery1: ~/examples/Chapter08/example06
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example06$ ansible-playbook -i mastery-hosts includer.yaml -v
No config file found; using defaults

PLAY [task inclusion] ****
TASK [included task] ****
ok: [localhost] => {
    "msg": "My data is first"
}

TASK [included task] ****
ok: [localhost] => {
    "msg": "My data is second"
}

PLAY RECAP ****
localhost                  : ok=2      changed=0      unreachable=0      failed=0      s
kippled=0      rescued=0      ignored=0
```

Figure 8.6 – Running a playbook with tagged include tasks, but without any tag-based filtering enabled

Now, we can select which tag to run—say, the second tag—by altering our `ansible-playbook` arguments, as follows:

```
ansible-playbook -i mastery-hosts includer.yaml -v --tags
second
```

In this instance, we should see only that occurrence of the included task being run, as the following screenshot shows:

```
jfreeman@mastery1: ~/examples/Chapter08/example06
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example06$ ansible-playbook -i mastery-hosts includer.yaml -v --tags second
No config file found; using defaults

PLAY [task inclusion] ****
TASK [included task] ****
ok: [localhost] => {
    "msg": "My data is second"
}

PLAY RECAP ****
localhost                  : ok=1      changed=0      unreachable=0      failed=0      s
kippled=0      rescued=0      ignored=0
```

Figure 8.7 – Running a playbook with tagged include tasks, only running tasks tagged "second"

Our example used the `--tags` command-line argument to indicate which tagged tasks to run. A different argument, `--skip-tags`, allows expressing the opposite—or in other words, which tagged tasks not to run.

## Task inclusions with loops

Task inclusions can be combined with loops as well. When adding a `loop` instance to a task inclusion (or a `with_loop` if using a version of Ansible earlier than 2.5), the tasks inside the file will be executed with the `item` variable, which holds the place of the current loop's value. The entire `include` file will be executed repeatedly until the loop runs out of items. Let's update our example play to demonstrate this, as follows:

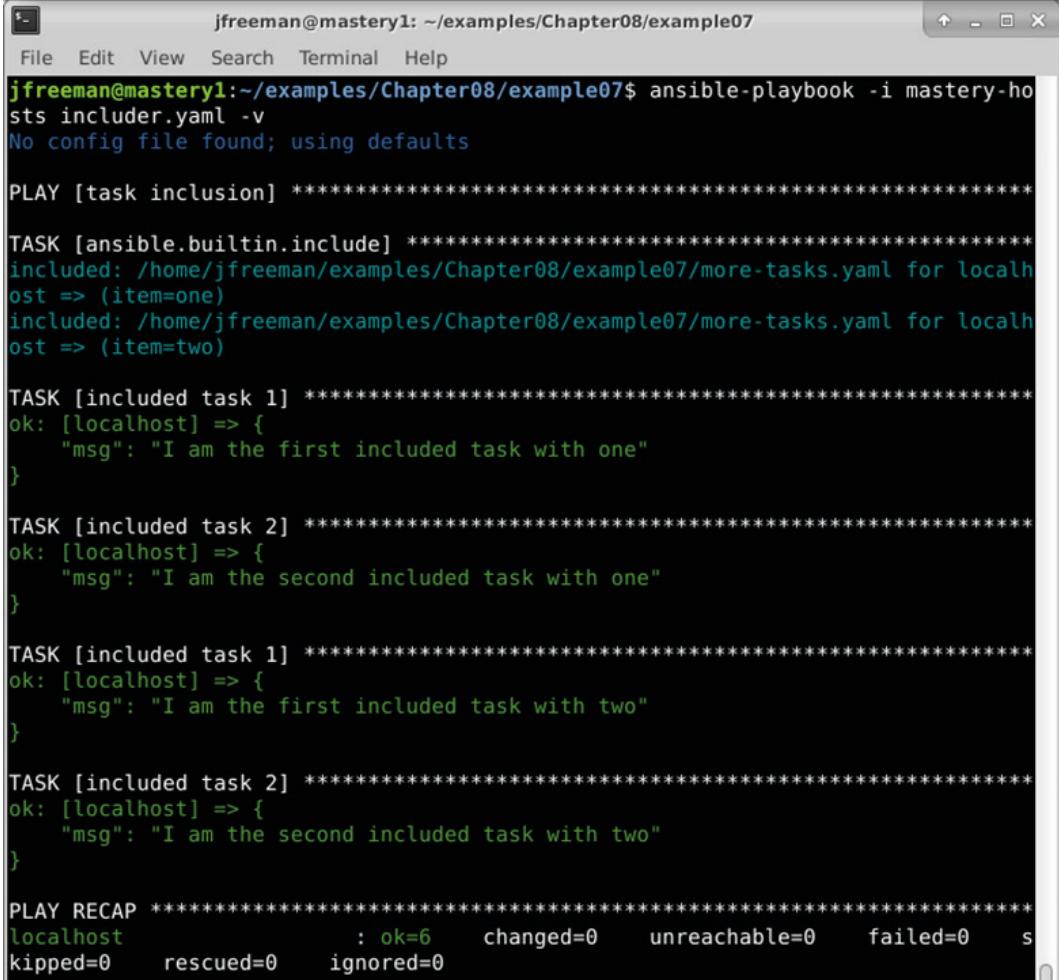
```
---
- name: task inclusion
  hosts: localhost
  gather_facts: false

  tasks:
    - ansible.builtin.include: more-tasks.yaml
      loop:
        - one
        - two
```

We also need to update our `more-tasks.yaml` file to make use of the loop `item` variable, as follows:

```
---
- name: included task 1
  ansible.builtin.debug:
    msg: "I am the first included task with {{ item }}"
- name: included task 2
  ansible.builtin.debug:
    msg: "I am the second included task with {{ item }}"
```

When executed with increased verbosity, we can tell that tasks 1 and 2 are executed a single time for each `item` variable in the loop, as the following screenshot shows:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter08/example07". The terminal is running the command "ansible-playbook -i mastery-hosts includer.yaml -v". The output shows the playbook executing tasks from the "more-tasks.yaml" file, which includes two tasks for each item in the loop. The tasks are "msg": "I am the first included task with one" and "msg": "I am the second included task with one". The final PLAY RECAP summary shows 6 tasks completed successfully (ok=6) with 0 changes, 0 unreachable hosts, 0 failed tasks, 0 skipped tasks, 0 rescued tasks, and 0 ignored tasks.

```
jfreeman@mastery1:~/examples/Chapter08/example07$ ansible-playbook -i mastery-hosts includer.yaml -v
No config file found; using defaults

PLAY [task inclusion] *****

TASK [ansible.builtin.include] *****
included: /home/jfreeman/examples/Chapter08/example07/more-tasks.yaml for localhost => (item=one)
included: /home/jfreeman/examples/Chapter08/example07/more-tasks.yaml for localhost => (item=two)

TASK [included task 1] *****
ok: [localhost] => {
    "msg": "I am the first included task with one"
}

TASK [included task 2] *****
ok: [localhost] => {
    "msg": "I am the second included task with one"
}

TASK [included task 1] *****
ok: [localhost] => {
    "msg": "I am the first included task with two"
}

TASK [included task 2] *****
ok: [localhost] => {
    "msg": "I am the second included task with two"
}

PLAY RECAP *****
localhost                  : ok=6      changed=0      unreachable=0      failed=0      skipped=0      rescued=0      ignored=0
```

Figure 8.8 – Running an included task file in a loop

Looping on **inclusion** is a powerful concept, but it does introduce one problem. What if there were tasks inside the included file that have their own loops? There will be a collision of the `item` variable, creating unexpected outcomes. For this reason, the `loop_control` feature was added to Ansible in version 2.1. Among other things, this feature provides a method to name the variable used for the loop, instead of the default of `item`. Using this, we can distinguish between the `item` instance that comes outside the inclusion from any `item` variables used inside the `include` statement. To demonstrate this, we'll add a `loop_var` loop control to our outer `include` statement, as follows:

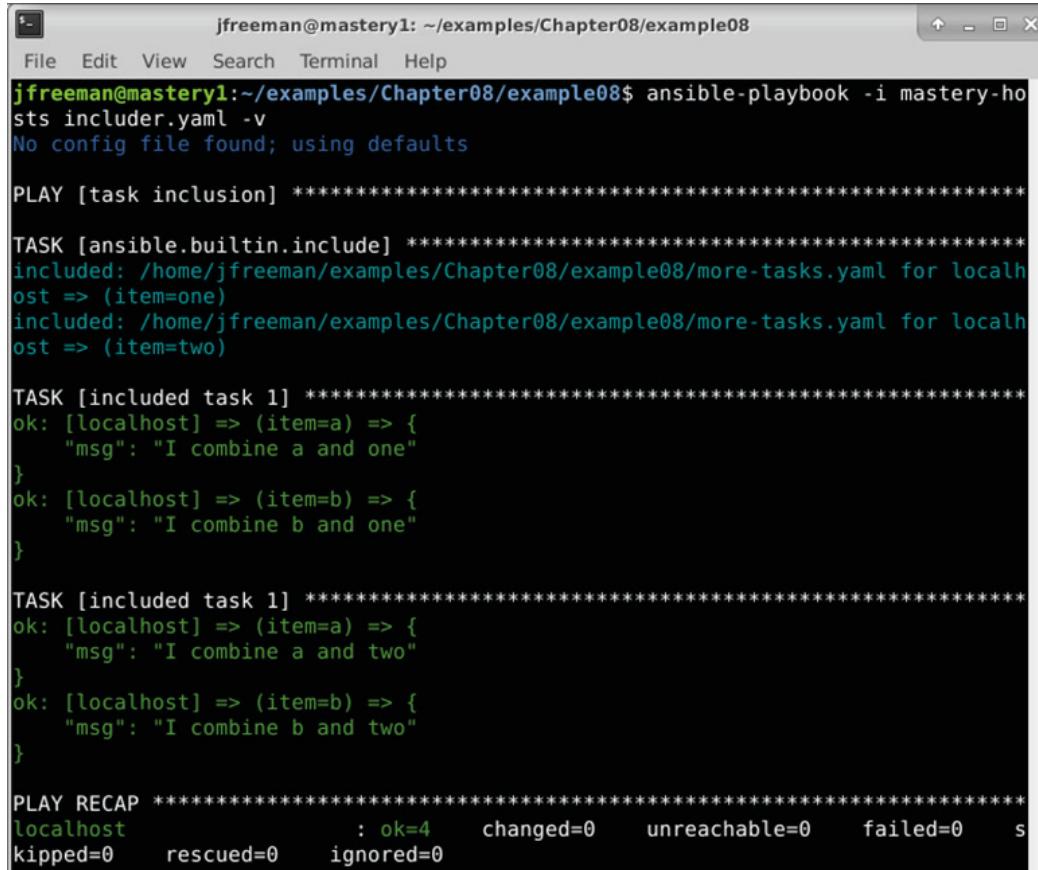
```
---
- name: task inclusion
  hosts: localhost
  gather_facts: false

  tasks:
    - ansible.builtin.include: more-tasks.yaml
      loop:
        - one
        - two
      loop_control:
        loop_var: include_item
```

Inside `more-tasks.yaml`, we'll have a task with its own loop, making use of `include_item` and the local `item` variable, as follows:

```
---
- name: included task 1
  ansible.builtin.debug:
    msg: "I combine {{ item }} and {{ include_item }}"
  loop:
    - a
    - b
```

When executed, we see that `task 1` is executed twice per inclusion loop and that the two `loop` variables are used, as the following screenshot shows:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter08/example08". The command run is "ansible-playbook -i mastery-hosts includer.yaml -v". The output shows the playbook execution:

```
jfreeman@mastery1:~/examples/Chapter08/example08$ ansible-playbook -i mastery-hosts includer.yaml -v
No config file found; using defaults

PLAY [task inclusion] ****
TASK [ansible.builtin.include] ****
included: /home/jfreeman/examples/Chapter08/example08/more-tasks.yaml for localhost => (item=one)
included: /home/jfreeman/examples/Chapter08/example08/more-tasks.yaml for localhost => (item=two)

TASK [included task 1] ****
ok: [localhost] => (item=a) => {
    "msg": "I combine a and one"
}
ok: [localhost] => (item=b) => {
    "msg": "I combine b and one"
}

TASK [included task 1] ****
ok: [localhost] => (item=a) => {
    "msg": "I combine a and two"
}
ok: [localhost] => (item=b) => {
    "msg": "I combine b and two"
}

PLAY RECAP ****
localhost                  : ok=4    changed=0    unreachable=0    failed=0    s
kipped=0      rescued=0    ignored=0
```

Figure 8.9 – Running a nested loop within an included task file, avoiding loop variable name collision

Other loop controls exist as well, such as `label`, which will define what is shown on the screen in the task output for the `item` value (useful for preventing large data structures from cluttering the screen), and `pause`, providing the ability to pause for a defined number of seconds between each loop.

## Including handlers

**Handlers** are essentially tasks. They're a set of potential tasks triggered by way of notifications from other tasks. As such, handler tasks can be included just as regular tasks can. The `include` operator is legal within the `handlers` block.

Unlike with task inclusions, variable data cannot be passed along when including `handler` tasks. However, it is possible to attach a conditional to a `handler` inclusion, to apply the conditional to every `handler` task within the file.

Let's create an example to demonstrate this. First, we'll create a playbook that has a task that will always change, and that includes a `handler` task file and attaches a conditional to that inclusion. The code is illustrated in the following snippet:

```
---  
- name: touch files  
  hosts: localhost  
  gather_facts: false  
  
  tasks:  
    - name: a task  
      ansible.builtin.debug:  
        msg: "I am a changing task"  
        changed_when: true  
      notify: a handler  
  
  handlers:  
    - ansible.builtin.include: handlers.yaml  
      when: foo | default('true') | bool
```

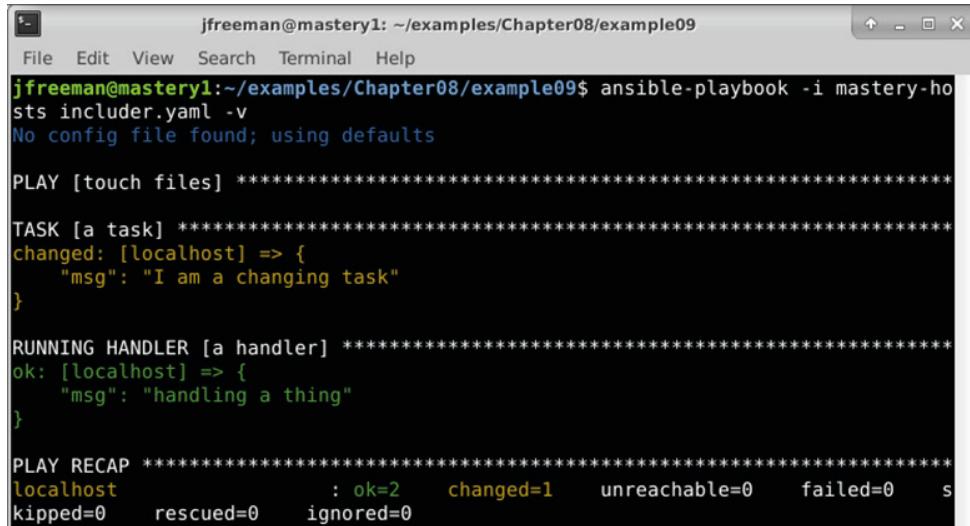
#### Important Note

When evaluating a variable that may be defined outside a playbook, it's best to use the `bool` filter to ensure that strings are properly converted to their Boolean meaning.

Next, we'll create a `handlers.yaml` file to define our `handler` task, as follows:

```
---  
- name: a handler  
  ansible.builtin.debug:  
    msg: "handling a thing"
```

If we execute this playbook without providing any further data, we should see our handler task trigger, as the following screenshot shows:



```
jfreeman@mastery1: ~/examples/Chapter08/example09
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example09$ ansible-playbook -i mastery-hosts includer.yaml -v
No config file found; using defaults

PLAY [touch files] ****
TASK [a task] ****
changed: [localhost] => {
    "msg": "I am a changing task"
}

RUNNING HANDLER [a handler] ****
ok: [localhost] => {
    "msg": "handling a thing"
}

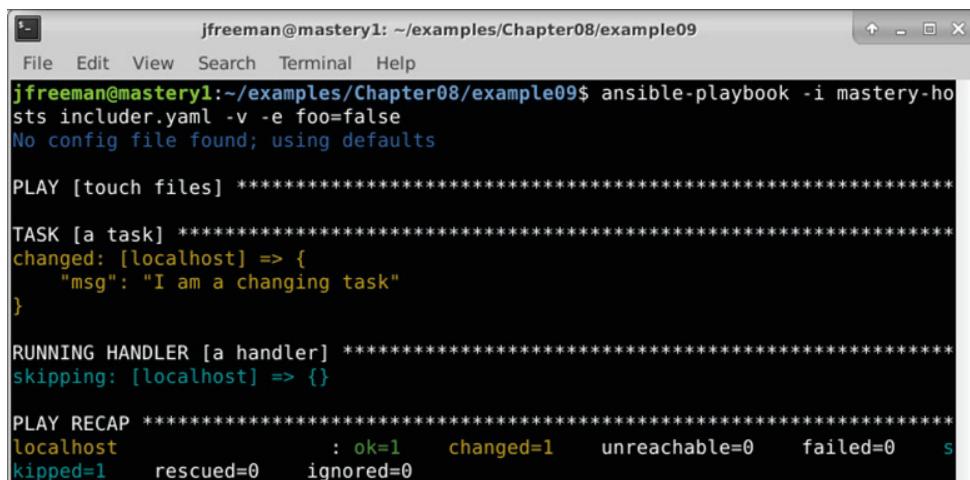
PLAY RECAP ****
localhost                  : ok=2    changed=1    unreachable=0    failed=0    s
kipped=0      rescued=0   ignored=0
```

Figure 8.10 – Using the include operator to run a handler from a task file

Now, let's run the playbook again; this time, we'll define `foo` as `extra-var` (overriding every other instance of it) and set it to `false` in our `ansible-playbook` execution arguments, as follows:

```
ansible-playbook -i mastery-hosts includer.yaml -v -e foo=false
```

This time, the output will look somewhat different, as the following screenshot shows:



```
jfreeman@mastery1: ~/examples/Chapter08/example09
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example09$ ansible-playbook -i mastery-hosts includer.yaml -v -e foo=false
No config file found; using defaults

PLAY [touch files] ****
TASK [a task] ****
changed: [localhost] => {
    "msg": "I am a changing task"
}

RUNNING HANDLER [a handler] ****
skipping: [localhost] => {}

PLAY RECAP ****
localhost                  : ok=1    changed=1    unreachable=0    failed=0    s
kipped=1      rescued=0   ignored=0
```

Figure 8.11 – Running the same play but this time forcing the `foo` conditional variable to `false`

As `foo` evaluates to `false`, our included handler gets skipped in this run of the playbook.

## Including variables

**Variable** data can also be separated into loadable files. This allows for the sharing of variables across multiple plays or playbooks or the inclusion of variable data that lives outside the project directory (such as secret data). Variable files are simple **YAML-formatted** files providing keys and values. Unlike task inclusion files, variable inclusion files cannot include more files.

Variables can be included in three different ways: via `vars_files`, via `include_vars`, or via `--extra-vars (-e)`.

### `vars_files`

The `vars_files` key is a play directive. It defines a list of files to read from to load variable data. These files are read and parsed at the time the playbook itself is parsed. Just as with including tasks and handlers, the path is relative to the file referencing the file.

Here is an example play that loads variables from a file:

```
---
- name: vars
  hosts: localhost
  gather_facts: false

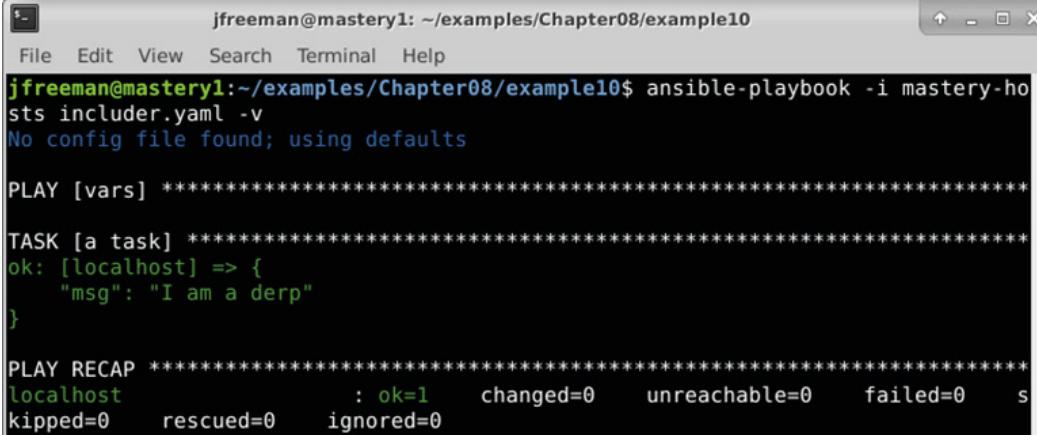
  vars_files:
    - variables.yaml

  tasks:
    - name: a task
      ansible.builtin.debug:
        msg: "I am a {{ varname }}"
```

Now, we need to create a `variables.yaml` file in the same directory as our playbook, as follows:

```
---  
varname: derp
```

Running the playbook with our usual command will show that the `varname` variable value is properly sourced from the `variables.yaml` file, as the following screenshot shows:



```
jfreeman@mastery1:~/examples/Chapter08/example10  
File Edit View Search Terminal Help  
jfreeman@mastery1:~/examples/Chapter08/example10$ ansible-playbook -i mastery-hosts includer.yaml -v  
No config file found; using defaults  
  
PLAY [vars] *****  
  
TASK [a task] *****  
ok: [localhost] => {  
    "msg": "I am a derp"  
}  
  
PLAY RECAP *****  
localhost : ok=1    changed=0    unreachable=0    failed=0    s  
kipped=0    rescued=0    ignored=0
```

Figure 8.12 – Including variables in a play using the `vars_files` directive

This is, of course, a very simple example, but it clearly demonstrates the ease of importing variables from a separate file.

## Dynamic `vars_files` inclusion

In certain scenarios, it may be desirable to parameterize the variable files to be loaded. It is possible to do this by using a variable as part of the filename; however, the variable must have a value defined at the time the playbook is parsed, just as when using variables in task names. Let's update our example play to load a variable file based on the data provided at execution time, as follows:

```
---  
- name: vars  
  hosts: localhost  
  gather_facts: false  
  
  vars_files:
```

```

- "{{ varfile }}"

tasks:
- name: a task
  ansible.builtin.debug:
    msg: "I am a {{ varname }}"

```

Now, when we execute the playbook, we'll provide the value for `varfile` with the `-e` argument, using a command like this:

```
ansible-playbook -i mastery-hosts includer.yaml -v -e
varfile=variables.yaml
```

The output should look like this:

```

jfreeman@mastery1: ~/examples/Chapter08/example11
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example11$ ansible-playbook -i mastery-hosts includer.yaml -v -e varfile=variables.yaml
No config file found; using defaults

PLAY [vars] ****
TASK [a task] ****
ok: [localhost] => {
    "msg": "I am a derp"
}

PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0      rescued=0      ignored=0

```

Figure 8.13 – Dynamically loading a `variables.yaml` file at playbook runtime

In addition to the variable value needing to be defined at execution time, the file to be loaded must also exist at execution time. This rule applies even if the file is generated by the Ansible playbook itself. Let's suppose that an Ansible playbook consists of four plays. The first play generates a YAML variable file. Then, further down, the fourth play references this file in a `vars_file` directive. Although it might initially appear as though this would work, the file does not exist at the point of execution (that is, when `ansible-playbook` is first run), and hence an error will be reported.

## include\_vars

A second method to include variable data from files is via the `include_vars` module. This module will load variables as a `task` action and will be done for each host. Unlike most modules, this module is executed locally on the Ansible host; therefore, all paths are still relative to the play file itself. Because the variable loading is done as a task, the evaluation of variables in the filename happens when the task is executed. Variable data in the filename can be host-specific and defined in a preceding task. Additionally, the file itself does not have to exist at execution time; it can be generated by a preceding task as well. This is a very powerful and flexible concept that can lead to very dynamic playbooks if used properly.

Before getting ahead of ourselves, let's demonstrate simple usage of `include_vars` by modifying our existing play to load the variable file as a task, as follows:

```
---
- name: vars
  hosts: localhost
  gather_facts: false

  tasks:
    - name: load variables
      ansible.builtin.include_vars: "{{ varfile }}"

    - name: a task
      ansible.builtin.debug:
        msg: "I am a {{ varname }}"
```

Execution of the playbook remains the same as in the previous example, where we specified the value for the `varfile` variable as an extra variable. Our output differs only slightly from previous iterations, as the following screenshot shows:

```
jfreeman@mastery1: ~/examples/Chapter08/example12
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example12$ ansible-playbook -i mastery-hosts includer.yaml -v -e varfile=variables.yaml
No config file found; using defaults

PLAY [vars] ****
TASK [load variables] ****
ok: [localhost] => {"ansible_facts": {"varname": "derp"}, "ansible_included_var_files": ["/home/jfreeman/examples/Chapter08/example12/variables.yaml"], "changed": false}

TASK [a task] ****
ok: [localhost] => {
    "msg": "I am a derp"
}

PLAY RECAP ****
localhost                  : ok=2    changed=0    unreachable=0    failed=0    s
kipped=0      rescued=0      ignored=0
```

Figure 8.14 – Running a playbook utilizing the include\_vars statement

Just as with other tasks, looping can be done to load more than one file in a single task. This is particularly effective when using the special `with_first_found` loop to iterate through a list of increasingly more generic filenames until a file is found to be loaded.

Let's demonstrate this by changing our play to use gathered host facts to try to load a variable file specific to the distribution, specific to the distribution family, or, finally, a default file, as follows:

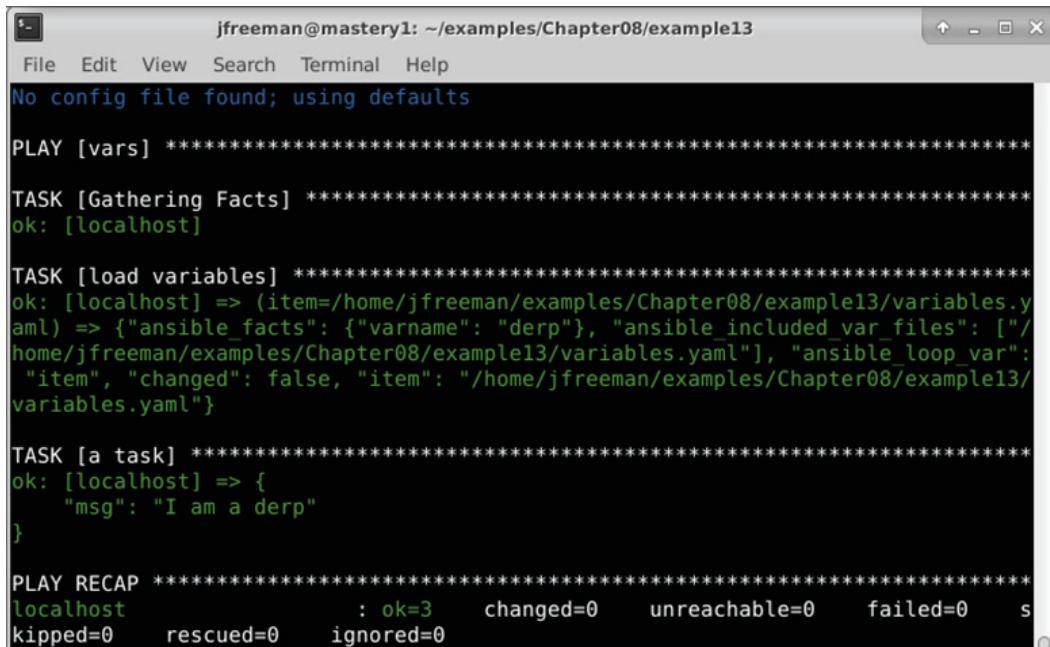
```
---
- name: vars
  hosts: localhost
  gather_facts: true

  tasks:
    - name: load variables
      ansible.builtin.include_vars: "{{ item }}"
      with_first_found:
        - "{{ ansible_distribution }}.yaml"
        - "{{ ansible_os_family }}.yaml"
        - variables.yaml

    - name: a task
```

```
ansible.builtin.debug:  
  msg: "I am a {{ varname }}"
```

The execution should look very similar to previous runs, only this time we'll see a fact-gathering task, and we will not pass along extra variable data in the execution. The output should look like this:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter08/example13". The window shows the execution of an Ansible play. The output includes:

```
jfreeman@mastery1: ~/examples/Chapter08/example13  
File Edit View Search Terminal Help  
No config file found; using defaults  
  
PLAY [vars] *****  
  
TASK [Gathering Facts] *****  
ok: [localhost]  
  
TASK [load variables] *****  
ok: [localhost] => (item=/home/jfreeman/examples/Chapter08/example13/variables.yaml) => {"ansible_facts": {"varname": "derp"}, "ansible_included_var_files": ["/home/jfreeman/examples/Chapter08/example13/variables.yaml"], "ansible_loop_var": "item", "changed": false, "item": "/home/jfreeman/examples/Chapter08/example13/variables.yaml"}  
  
TASK [a task] *****  
ok: [localhost] => {  
    "msg": "I am a derp"  
}  
  
PLAY RECAP *****  
localhost                  : ok=3      changed=0      unreachable=0      failed=0      s  
skipped=0      rescued=0      ignored=0
```

Figure 8.15 – Dynamically including the first valid variables file found in an Ansible play

We can also see from the output which file was found to load. In this case, `variables.yaml` was loaded, as the other two files did not exist. This practice is commonly used to load variables that are operating system-specific to the host in question. Variables for a variety of operating systems can be written out to appropriately named files. By utilizing the `ansible_distribution` variable, which is populated by fact-gathering, variable files that use `ansible_distribution` values as part of their name can be loaded by way of a `with_first_found` argument. A default set of variables can be provided in a file that does not use any variable data as a failsafe, as we did here in our `variables.yaml` file.

## extra-vars

The final method to load variable data from a file is to reference a file path with the `--extra-vars` (or `-e`) argument to `ansible-playbook`. Normally, this argument expects a set of `key=value` data; however, if a file path is provided and prefixed with the `@` symbol, Ansible will read the entire file to load variable data. Let's alter one of our earlier examples, where we used `-e`, and instead of defining a variable directly on the command line, we'll include the variable file we've already written out, as follows:

```
---
- name: vars
  hosts: localhost
  gather_facts: false

  tasks:
    - name: a task
      ansible.builtin.debug:
        msg: "I am a {{ varname }}"
```

When we provide a path after the `@` symbol, the path is relative to the current working directory, regardless of where the playbook itself lives. Let's execute our playbook and provide a path to `variables.yaml`, as follows:

```
ansible-playbook -i mastery-hosts includer.yaml -v -e @
variables.yaml
```

The output should look like this:

```
jfreeman@mastery1: ~/examples/Chapter08/example14
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example14$ ansible-playbook -i mastery-hosts includer.yaml -v -e @variables.yaml
No config file found; using defaults

PLAY [vars] ****
TASK [a task] ****
ok: [localhost] => {
    "msg": "I am a derp"
}

PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 8.16 – Including a `variables.yaml` file through the extra variables command-line parameter

Here, we can see that once again our `variables.yaml` file was included successfully, but, as you can see from the previous code, it is not even mentioned in the playbook itself—we were able to load it in its entirety through the `-e` flag.

#### Important Note

When including a variable file with the `--extra-vars` argument, the file must exist at ansible-playbook execution time.

Variable inclusion is incredibly powerful in Ansible—but what about playbooks themselves? Here, things are a bit different, and as the chapter progresses, we will look at how to make effective use of reusable tasks and playbook code, thus encouraging good programming practices with Ansible.

## Including playbooks

Playbook files can include other whole playbook files. This construct can be useful to tie together a few independent playbooks into a larger, more comprehensive playbook. Playbook inclusion is a bit more primitive than task inclusion. You cannot perform variable substitution when including a playbook, you cannot apply conditionals, and you cannot apply tags, either. The playbook files to be included must exist at the time of execution as well.

Prior to Ansible 2.4, playbook inclusion was achieved using the `include` keyword—however, this was removed in Ansible 2.8, and so it should not be used. Instead, you should now use `ansible.builtin.import_playbook`. This is a play-level directive—it cannot be used as a task. However, it is very easy to use. Let's define a simple example to demonstrate this. First, let's create a playbook that will be included, called `includeme.yaml`. Here's the code to do this:

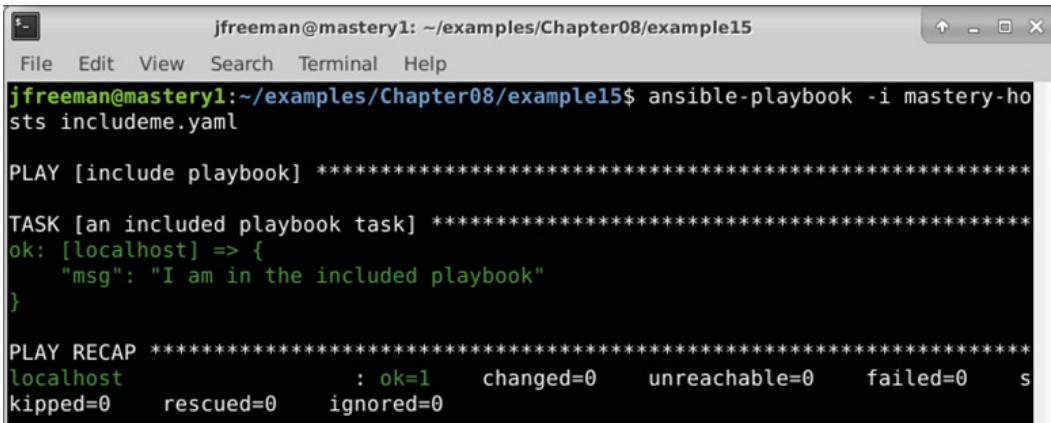
```
---
- name: include playbook
  hosts: localhost
  gather_facts: false

  tasks:
    - name: an included playbook task
      ansible.builtin.debug:
        msg: "I am in the included playbook"
```

As you will no doubt recognize by now, this is a complete standalone playbook and we could run it in isolation using the following command:

```
ansible-playbook -i mastery-hosts includeme.yaml
```

A successful run will produce output like that shown here:



```
jfreeman@mastery1: ~/examples/Chapter08/example15
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example15$ ansible-playbook -i mastery-hosts includeme.yaml

PLAY [include playbook] ****
TASK [an included playbook task] ****
ok: [localhost] => {
    "msg": "I am in the included playbook"
}

PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0      rescued=0      ignored=0
```

Figure 8.17 – Running our playbook to be included first as a standalone playbook

However, we can also import this into another playbook. Modify the original `includer.yaml` playbook so that it looks like this:

```
---
- name: include playbook
  hosts: localhost
  gather_facts: false

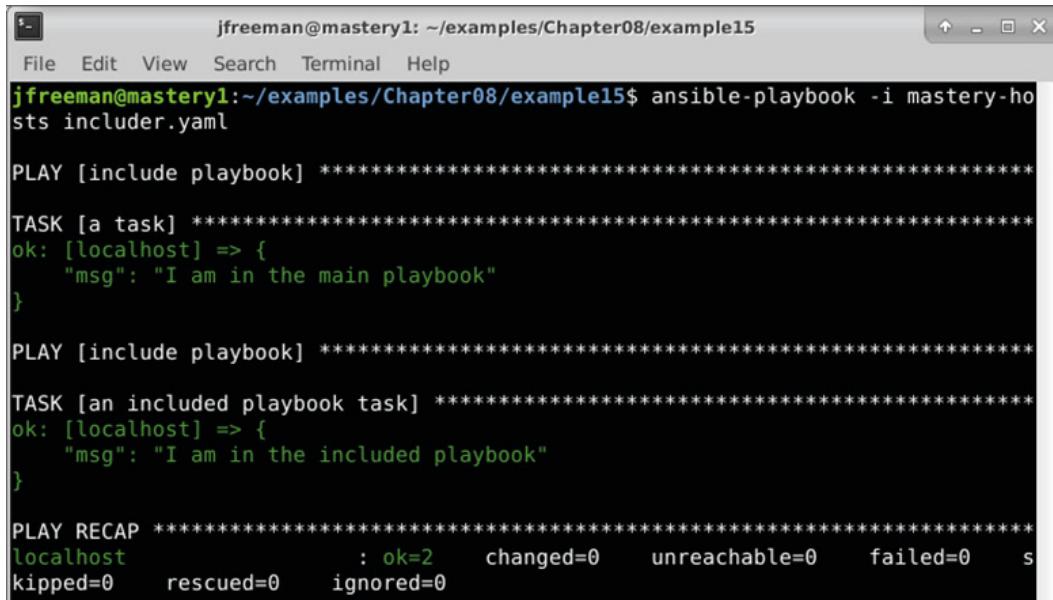
  tasks:
    - name: a task
      ansible.builtin.debug:
        msg: "I am in the main playbook"

    - name: include a playbook
      ansible.builtin.import_playbook: includeme.yaml
```

We then run it using this command:

```
ansible-playbook -i mastery-hosts includer.yaml
```

We can see that both debug messages are displayed, and the imported playbook is run after the initial task, which is the sequence we defined in the original playbook. The following screenshot shows this in action:

A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter08/example15". The window shows the output of an Ansible playbook run. It starts with a "PLAY [include playbook]" message, followed by a "TASK [a task]" message with a task definition. This is followed by another "PLAY [include playbook]" message, then a "TASK [an included playbook task]" message with its own task definition. Finally, a "PLAY RECAP" summary is shown, indicating 2 tasks were successful (ok=2), 0 changed, 0 unreachable, 0 failed, 0 skipped, 0 rescued, and 0 ignored.

```
jfreeman@mastery1:~/examples/Chapter08/example15$ ansible-playbook -i mastery-hosts includer.yaml

PLAY [include playbook] ****
TASK [a task] ****
ok: [localhost] => {
    "msg": "I am in the main playbook"
}

PLAY [include playbook] ****
TASK [an included playbook task] ****
ok: [localhost] => {
    "msg": "I am in the included playbook"
}

PLAY RECAP ****
localhost                  : ok=2      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 8.18 – Running a playbook that includes a second playbook

In this way, it is very easy to reuse whole playbooks without needing to restructure them into the format of roles, task files, or otherwise. Note, however, that this feature is subject to active development in Ansible, so it is recommended that you always refer to the documentation to ensure you can achieve the results you are looking for.

# Roles (structures, defaults, and dependencies)

With a functional understanding of the inclusion of variables, tasks, handlers, and playbooks, we can move on to the more advanced topic of **roles**. Roles bring together these different facets of Ansible code creation to provide a fully independent collection of variables, tasks, files, templates, and modules that can be reused over again in different playbooks. Although not limited as such by design, it is normal practice for each role to be typically limited to a particular purpose or desired end result, with all the necessary steps to reach that result either within the role itself or through dependencies (in other words, further roles that themselves are specified as dependencies of a role). It is important to note that roles are not playbooks, and there is no way to directly execute a role. Roles have no settings for which host(s) the role will apply to. Top-level playbooks are the glue that binds the hosts from your inventory to roles that should be applied to those hosts. As we saw in *Chapter 2, Migrating from Earlier Ansible Versions*, roles can also be part of Ansible collections. As we have already looked at the structure of collections in this earlier chapter, in this section we will focus in greater depth on how to construct the roles themselves.

## Role structure

**Roles** have a structured layout on the **filesystem**. This structure exists to provide automation around including tasks, handlers, variables, modules, and role dependencies. The structure also allows for the easy reference of files and templates from anywhere within the role.

In *Chapter 2, Migrating from Earlier Ansible Versions*, we look at how to reference roles from collections. They do not have to be used as part of a collection, however, and assuming you are working with roles outside of this context, they all live in a subdirectory of a playbook directory structure below the `roles/` directory. This is, of course, configurable by way of the `roles_path` general configuration key, but let's stick to the defaults. Each role is itself a directory tree. The role name is the directory name within `roles/`. Each role can have a number of subdirectories with special meanings that are processed when a role is applied to a set of hosts.

A role may contain all these elements or as few as just one of them. Missing elements are simply ignored. Some roles exist just to provide common handlers across a project. Other roles exist as a single dependency point that in turn just depends on numerous other roles.

## Tasks

The task file is the core part of a role, and if `roles/<role_name>/tasks/main.yaml` exists, then all the tasks within this file (and indeed any other files it includes) will be loaded in the play and executed.

## Handlers

Similar to tasks, handlers are automatically loaded from `roles/<role_name>/handlers/main.yaml`, if the file exists. These handlers can be referenced by any task within the role, or by any tasks within any other role that lists this role as a dependency.

## Variables

There are two types of variables that can be defined in a role. There are role variables, loaded from `roles/<role_name>/vars/main.yaml`, and there are role defaults, loaded from `roles/<role_name>/defaults/main.yaml`. The difference between `vars` and `defaults` has to do with precedence order. Refer to *Chapter 1, The System Architecture and Design of Ansible*, for a detailed description of the order. **Role defaults** are the lowest-order variables. Literally any other definition of a variable will take precedence over a role default. Role defaults can be thought of as placeholders for actual data, a reference of which variables a developer may be interested in defining with site-specific values. **Role variables**, on the other hand, have a higher order of precedence. Role variables can be overridden, but they are generally used when the same dataset is referenced more than once within a role. If the dataset is to be redefined with site-local values, then the variable should be listed in the role defaults rather than the role variables.

## Modules and plugins

A role can include custom modules as well as plugins. While we are in the transitional phase to Ansible 4.0 and beyond, this is still supported, but you will no doubt have noticed that collections can also include custom **modules** and **plugins**. At the current time, where you place your modules and plugins will depend upon the target version of Ansible you are writing your role for. If you wish to maintain backward compatibility with the 2.x releases, then you should place modules and plugins into your role directory structure, as described here. If you only want compatibility with Ansible releases 3.0 and later, you could consider placing them in a collection instead. Note, however, that with the move to collections, your plugins and modules are less likely to be accepted into the `ansible-core` package, unless they provide what is considered core functionality.

Modules (if present in a role) are loaded from `roles/<role_name>/library/` and can be utilized by any task in the role, or indeed by any later role in the play. It is important to note that modules provided in this path will override any other copies of the same module name, and once again it is important to reference modules using FQCNs wherever possible to avoid any unexpected results.

Plugins will automatically be loaded if found inside of a role, in one of the following subdirectories:

- action\_plugins
- lookup\_plugins
- callback\_plugins
- connection\_plugins
- filter\_plugins
- strategy\_plugins
- cache\_plugins
- test\_plugins
- shell\_plugins

## Dependencies

Roles can express a **dependency** upon another role. It is a common practice for sets of roles to all depend on a common role for tasks, handlers, modules, and so on. Those roles may depend upon only having to be defined once. When Ansible processes a role for a set of hosts, it first looks for dependencies listed in `roles/<role_name>/meta/main.yaml`. Should any be defined, then those roles will be processed immediately and the tasks contained in those roles will be executed (after checking for any dependencies also listed within them). This process carries on until all dependencies have been established and loaded (and tasks executed where present) before Ansible results to starting on the initial role tasks. Remember—dependencies are always executed before the role itself. We will describe role dependencies in more depth later in this chapter.

## Files and templates

Task and handler modules can reference files using only relative paths within `roles/<role_name>/files/`. The filename can be provided without any prefix (although this is allowed if you wish) and will be sourced from `roles/<role_name>/files/<relative_directory>/<file_name>`. Modules such as `ansible.builtin.template`, `ansible.builtin.copy`, and `ansible.builtin.script` are three typical ones that you will see many examples of, taking advantage of this useful feature.

Similarly, templates used by the `ansible.builtin.template` module can be referenced relatively within `roles/<role_name>/templates/`. The following code sample uses a relative path to load the `derp.j2` template from the full `roles/<role_name>/templates/herp/derp.j2` path:

```
- name: configure herp
  ansible.builtin.template:
    src: herp/derp.j2
    dest: /etc/herp/derp.j2
```

In this way, it is easy to organize files within the standard role directory structure and still access them easily from within the role without having to type in long and complex paths. Later in this chapter, we'll introduce you to the `ansible-galaxy role init` command, which will help you build skeleton directory structures for new roles with even greater ease—see the *Role sharing* section for more details.

## Putting it all together

To illustrate what a full role structure might look like, here is an example role by the name of `demo`:

```
roles/demo
  ├── defaults
  |   |--- main.yaml
  ├── files
  |   |--- foo
  ├── handlers
  |   |--- main.yaml
  ├── library
  |   |--- samplemod.py
  ├── meta
  |   |--- main.yaml
  ├── tasks
  |   |--- main.yaml
  ├── templates
  |   |--- bar.j2
  └── vars
      |--- main.yaml
```

When creating a role, not every directory or file is required. Only files that exist will be processed. Thus, our example of a role does not require or use handlers; the entire `handlers` part of the tree could simply be left out.

## Role dependencies

As stated before, roles can depend on other roles. These relationships are called dependencies and they are described in a role's `meta/main.yaml` file. This file expects a top-level data hash with a key of `dependencies`; the data within is a list of roles. You can see an illustration of this in the following code snippet:

```
---  
dependencies:  
  - role: common  
  - role: apache
```

In this example, Ansible will fully process the `common` role first (and any dependencies it may express) before continuing with the `apache` role and then finally starting on the role's tasks.

Dependencies can be referenced by name without any prefix if they exist within the same directory structure or live within the configured `roles_path` configuration key. Otherwise, full paths can be used to locate roles, as illustrated here:

```
role: /opt/ansible/site-roles/apache
```

When expressing a dependency, it is possible to pass along data to the dependency. The data can be variables, tags, or even conditionals.

## Role dependency variables

Variables that are passed along when listing a dependency will override values for matching variables defined in `defaults/main.yaml` or `vars/main.yaml`. This can be useful for using a common role, such as an `apache` role, as a dependency while providing site-specific data, such as which ports to open in the firewall or which `apache` modules to enable. Variables are expressed as additional keys to the role listing. Thus, continuing our hypothetical example, consider that we need to pass some variables to both the `common` and `apache` role dependencies we discussed, as follows:

```
---  
dependencies:  
  - role: common
```

```
simple_var_a: True
simple_var_b: False
- role: apache
complex_var:
    key1: value1
    key2: value2
short_list:
    - 8080
    - 8081
```

When providing dependency variable data, two names are reserved and should not be used as role variables: `tags` and `when`. The former is used to pass tag data into a role, and the latter is used to pass a conditional into a role.

## Tags

Tags can be applied to all the tasks found within a dependency role. This functions much in the same way as tags being applied to included task files, as described earlier in this chapter. The syntax is simple: the `tags` key can be a single item or a list. To demonstrate, let's further expand our theoretical example by adding some tags, as follows:

```
---
dependencies:
  - role: common
    simple_var_a: True
    simple_var_b: False
    tags: common_demo
  - role: apache
    complex_var:
        key1: value1
        key2: value2
    short_list:
        - 8080
        - 8081
    tags:
        - apache_demo
        - 8080
        - 8181
```

As with adding tags to the included task files, all the tasks found within a dependency (and any dependency within that hierarchy) will gain the provided tags.

## Role dependency conditionals

While it is not possible to prevent the processing of a dependency role with a conditional, it is possible to skip all the tasks within a dependency role hierarchy by applying a conditional to a dependency. This mirrors the functionality of task inclusion with conditionals as well. The `when` key is used to express the conditional. Once again, we'll grow our example by adding a dependency to demonstrate the syntax, as follows:

```
---  
dependencies:  
  - role: common  
    simple_var_a: True  
    simple_var_b: False  
    tags: common_demo  
  - role: apache  
    complex_var:  
      key1: value1  
      key2: value2  
    short_list:  
      - 8080  
      - 8081  
    tags:  
      - apache_demo  
      - 8080  
      - 8181  
    when: backend_server == 'apache'
```

In this example, the `apache` role will always be processed, but tasks within the role will only run when the `backend_server` variable contains the `apache` string.

## Role application

Roles are not plays. They do not possess any opinions about which hosts the role tasks should run on, which connection methods to use, whether to operate serially, or any other play behaviors described in *Chapter 1, The System Architecture and Design of Ansible*. Roles must be applied inside a play within a playbook, where all these opinions can be expressed.

To apply a role within a play, the `roles` operator is used. This operator expects a list of roles to apply to the hosts in the play. Much like describing role dependencies, when describing roles to apply, data can be passed along, such as variables, tags, and conditionals. The syntax is exactly the same.

To demonstrate applying roles within a play, let's create a simple role and apply it to a simple playbook. First, let's build a role named `simple`, which will have a single `debug` task in `roles/simple/tasks/main.yaml` that prints the value of a role default variable defined in `roles/simple/defaults/main.yaml`. First, let's create a task file (in the `tasks/` subdirectory), as follows:

```
---
```

```
- name: print a variable
  ansible.builtin.debug:
    var: derp
```

Next, we'll write our default file with a single variable, `derp`, like this:

```
---
```

```
derp: derp
```

To execute this role, we'll write a playbook with a single play to apply the role. We'll call our playbook `roleplay.yaml`, and it'll live at the same directory level as the `roles/` directory. The code is illustrated in the following snippet:

```
---
```

```
- hosts: localhost
  gather_facts: false
```

```
  roles:
    - role: simple
```

**Important Note**

If no data is provided with the role, an alternative syntax that just lists the roles to apply can be used, instead of the hash. However, for consistency, I feel it's best to always use the same syntax within a project.

We'll reuse our `mastery-hosts` inventory from earlier chapters and execute the playbook in the normal manner (we don't need any added verbosity here), by running the following command:

```
ansible-playbook -i mastery-hosts roleplay.yaml
```

The output should look something like this:



```
jfreeman@mastery1: ~/examples/Chapter08/example16
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example16$ ansible-playbook -i mastery-hosts roleplay.yaml

PLAY [localhost] ****
TASK [simple : print a variable] ****
ok: [localhost] => {
    "derp": "herp"
}

PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0      rescued=0    ignored=0
```

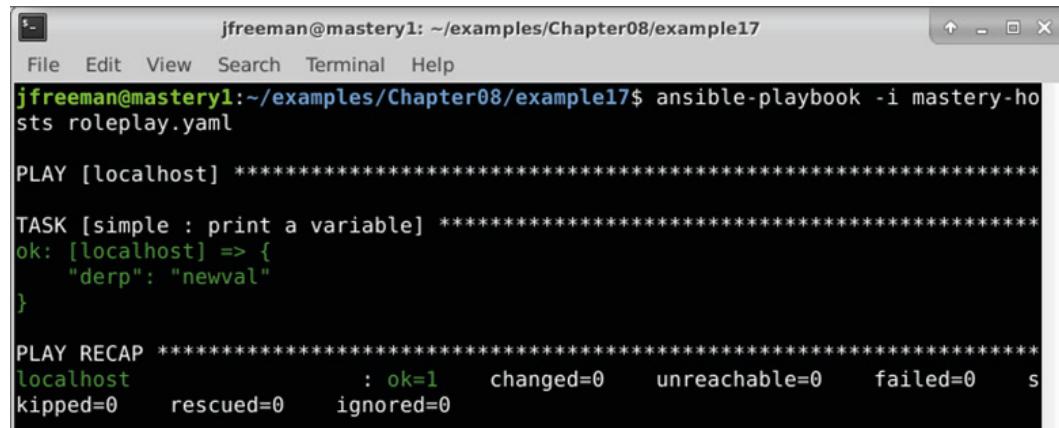
Figure 8.19 – Running our simple role from a playbook using the default role variable data

Thanks to the magic of roles, the `derp` variable value was automatically loaded from the role defaults. Of course, we can override the default value when applying the role. Let's modify our playbook and supply a new value for `derp`, as follows:

```
---
- hosts: localhost
  gather_facts: false

  roles:
    - role: simple
      derp: newval
```

This time, when we execute (using the same command as used previously), we'll see `newval` as the value for `derp`, as shown in the following screenshot:



```
jfreeman@mastery1:~/examples/Chapter08/example17$ ansible-playbook -i mastery-hosts roleplay.yaml

PLAY [localhost] ****
TASK [simple : print a variable] ****
ok: [localhost] => {
    "derp": "newval"
}

PLAY RECAP ****
localhost                  : ok=1      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 8.20 – Running the same role but this time overriding the default variable data at the play level

Multiple roles can be applied within a single play. The `roles :` key expects a list value. Just add more roles to apply more roles, as shown here (this next example is theoretical and is left as an exercise for you):

```
---
- hosts: localhost
  gather_facts: false

  roles:
    - role: simple
      derp: newval
    - role: second_role
      othervar: value
    - role: third_role
    - role: another_role
```

This playbook will load a total of four roles—`simple`, `second_role`, `third_role`, and `another_role`—and each will be executed in the sequence in which they are listed.

## Mixing roles and tasks

Plays that use roles are not limited to just roles. These plays can have tasks of their own, as well as two other blocks of tasks: `pre_tasks` and `post_tasks` blocks. In a break to the task execution order we have looked at throughout this book, the order in which these tasks are executed is not dependent upon which order these sections are listed in the play itself; instead, there is a strict order to block execution within a play. See *Chapter 1, The System Architecture and Design of Ansible*, for details on the playbook order of operations.

**Handlers** for a play are flushed at multiple points. If there is a `pre_tasks` block, handlers are flushed after all `pre_tasks` blocks are executed. Then, the roles and tasks blocks are executed (roles first, then tasks, regardless of the order they are written in the playbook), after which the handlers will be flushed again. Finally, if a `post_tasks` block exists, the handlers will be flushed once again after all `post_tasks` blocks have executed. Of course, handlers can be flushed at any time with the `meta: flush_handlers` call. Let's expand on our `roleplay.yaml` file to demonstrate all the different times at which handlers can be triggered, as follows:

```
---
- hosts: localhost
  gather_facts: false

  pre_tasks:
    - name: pretask
      ansible.builtin.debug:
        msg: "a pre task"
      changed_when: true
      notify: say hi

  roles:
    - role: simple
      derp: newval

  tasks:
    - name: task
      ansible.builtin.debug:
        msg: "a task"
      changed_when: true
```

```
    notify: say hi

post_tasks:
  - name: posttask
    ansible.builtin.debug:
      msg: "a post task"
      changed_when: true
    notify: say hi

handlers:
  - name: say hi
    ansible.builtin.debug:
      msg: "hi"
```

We'll also modify our simple role's tasks to notify the `say_hi` handler as well, as follows:

```
---  
  - name: print a variable  
    ansible.builtin.debug:  
      var: derp  
    changed_when: true  
    notify: say hi
```

#### Important Note

This only works because the `say_hi` handler has been defined in the play that is calling the `simple` role. If the handler is not defined, an error will occur. It's best practice to only notify handlers that exist within the same role or any role marked as a dependency.

Running our playbook again using the same command as in the previous examples should result in the `say_hi` handler being called a total of three times: once for `pre_tasks` blocks, once for roles and tasks, and once for `post_tasks` blocks, as the following screenshot demonstrates:

```
jfreeman@mastery1: ~/examples/Chapter08/example18
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example18$ ansible-playbook -i mastery-ho
sts roleplay.yaml

PLAY [localhost] ****
TASK [pretask] ****
changed: [localhost] => {
    "msg": "a pre task"
}

RUNNING HANDLER [say hi] ****
ok: [localhost] => {
    "msg": "hi"
}

TASK [simple : print a variable] ****
ok: [localhost] => {
    "derp": "newval"
}

TASK [task] ****
changed: [localhost] => {
    "msg": "a task"
}

RUNNING HANDLER [say hi] ****
ok: [localhost] => {
    "msg": "hi"
}

TASK [posttask] ****
changed: [localhost] => {
    "msg": "a post task"
}

RUNNING HANDLER [say hi] ****
ok: [localhost] => {
    "msg": "hi"
}

PLAY RECAP ****
localhost                  : ok=7      changed=3      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 8.21 – Running a playbook to demonstrate mixing roles and tasks, and handler execution

While the order in which `pre_tasks`, `roles`, `tasks`, and `post_tasks` blocks are written into a play does not impact the order in which those sections are executed, it's best practice to write them in the order in which they will be executed. This is a visual cue to help remember the order and to avoid confusion when reading the playbook later.

## Role includes and imports

With Ansible version 2.2, a new `ansible.builtin.include_role` action plugin was made available as a technical preview. Then, in **Ansible version 2.4**, this concept was further developed by the addition of the `ansible.builtin.import_role` plugin. We will refer to these plugins without their FQCNs for conciseness.

These plugins are used in a task to include and execute an entire role directly from a task. The difference between the two is subtle but important—the `include_role` plugin is considered dynamic, meaning the code is processed during runtime when the task referencing it is encountered.

The `import_role` plugin, on the other hand, is considered static, meaning all imports are preprocessed at the time the playbook is initially parsed. This has various impacts on their use in playbooks—for example, `import_role` cannot be used in loops, while `include_role` can.

### Important Note

Full details of the trade-offs between importing and including can be found in the official Ansible documentation here: [https://docs.ansible.com/ansible/latest/user\\_guide/playbooks\\_reuse.html](https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse.html).

In the previous edition of this book, these plugins were considered a technical preview—however, they are now part of the `ansible.builtin` collection and so can now be considered stable and used for your code as you see fit.

## Role sharing

One of the advantages of using roles is the ability to share a role across plays, playbooks, entire project spaces, and even across organizations. Roles are designed to be self-contained (or to clearly reference dependent roles) so that they can exist outside of a project space where the playbook that applies the role lives. Roles can be installed in shared paths on an Ansible host, or they can be distributed via source control.

## Ansible Galaxy

**Ansible Galaxy** (<https://galaxy.ansible.com/>), as we discussed in *Chapter 2, Migrating from Earlier Ansible Versions*, is a community hub for finding and sharing Ansible roles and collections. Anybody can visit the website to browse these and reviews; plus, users who create a login can provide reviews of the roles they've tested. Roles from Galaxy can be downloaded using the `ansible-galaxy` utility provided with Ansible.

The `ansible-galaxy` utility can connect to and install roles from the Ansible Galaxy website. This utility will default to installing roles into `/etc/ansible/roles`. If `roles_path` is configured or if a runtime path is provided with the `--roles-path` (or `-p`) option, the roles will be installed there instead. If any roles have been installed to the `roles_path` option or the provided path, `ansible-galaxy` can list those and show information about those as well. To demonstrate the usage of `ansible-galaxy`, let's use it to install a role for installing and managing Docker on Ubuntu from Ansible Galaxy into the `roles` directory we've been working with. Installing roles from Ansible Galaxy requires `username . rolename`, as multiple users may have uploaded roles with the same name. To work through an example, we will use the `docker_ubuntu` role from the `angstwad` user, as shown in the following screenshot:

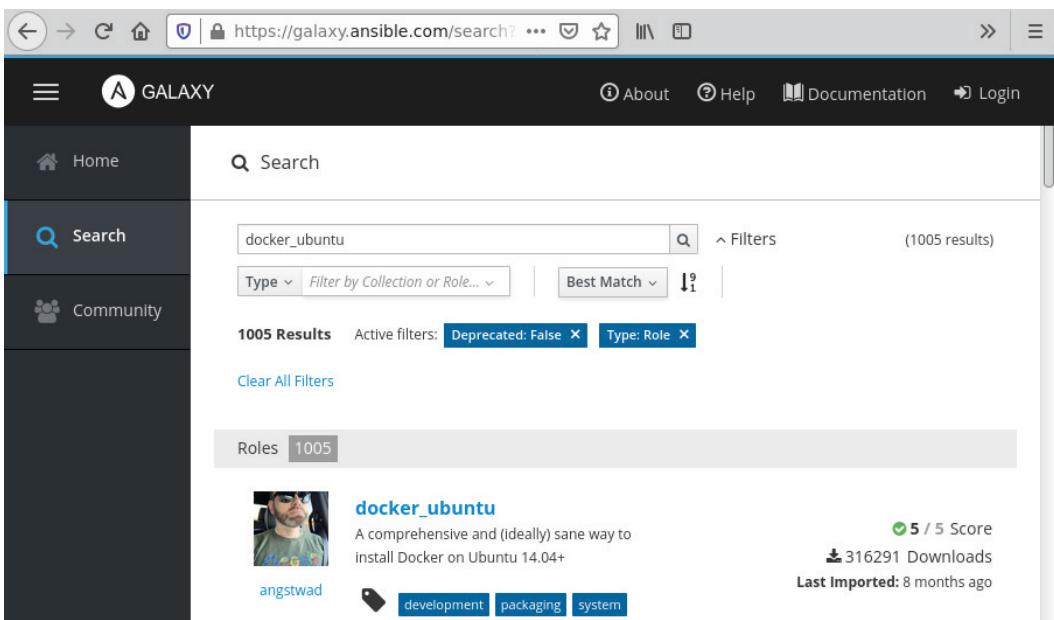


Figure 8.22 – Locating an example community-contributed role on Ansible Galaxy

We can now make use of this role by referencing `angstwad.docker_ubuntu` in a play or another role's dependencies block. However, let's get started by demonstrating how we can install this in our current working directory. We'll first off create a `roles/` directory, and then install the aforementioned role into this directory with the following commands:

```
mkdir roles/
ansible-galaxy role install -p roles/ angstwad.docker_ubuntu
```

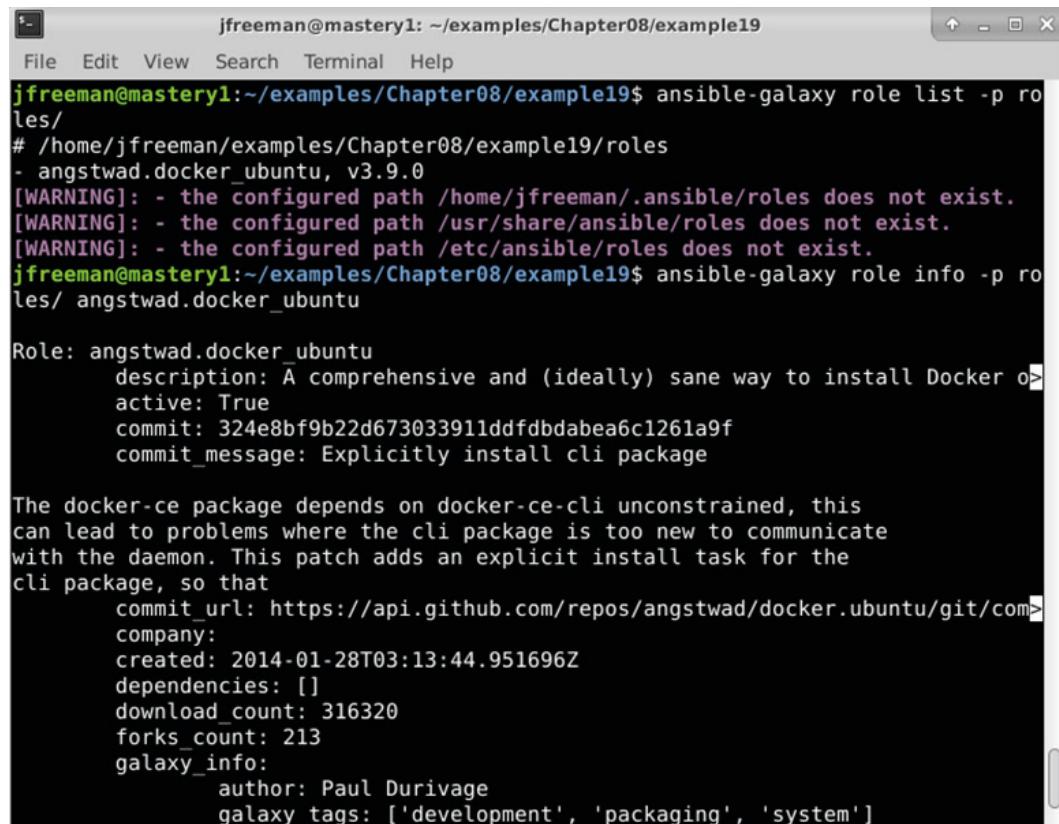
Once we've installed our example role, we can query it (and any other roles that might exist in the `roles/` directory) using the following command:

```
ansible-galaxy role list -p roles/
```

You can also query information about the role such as the description, creator, version, and so on locally using the following command:

```
ansible-galaxy role info -p roles/ angstwad.docker_ubuntu
```

The following screenshot gives an idea of the kind of output you can expect from the two preceding commands:



A terminal window titled "jfreeman@mastery1: ~/examples/Chapter08/example19". The window shows the following command-line session:

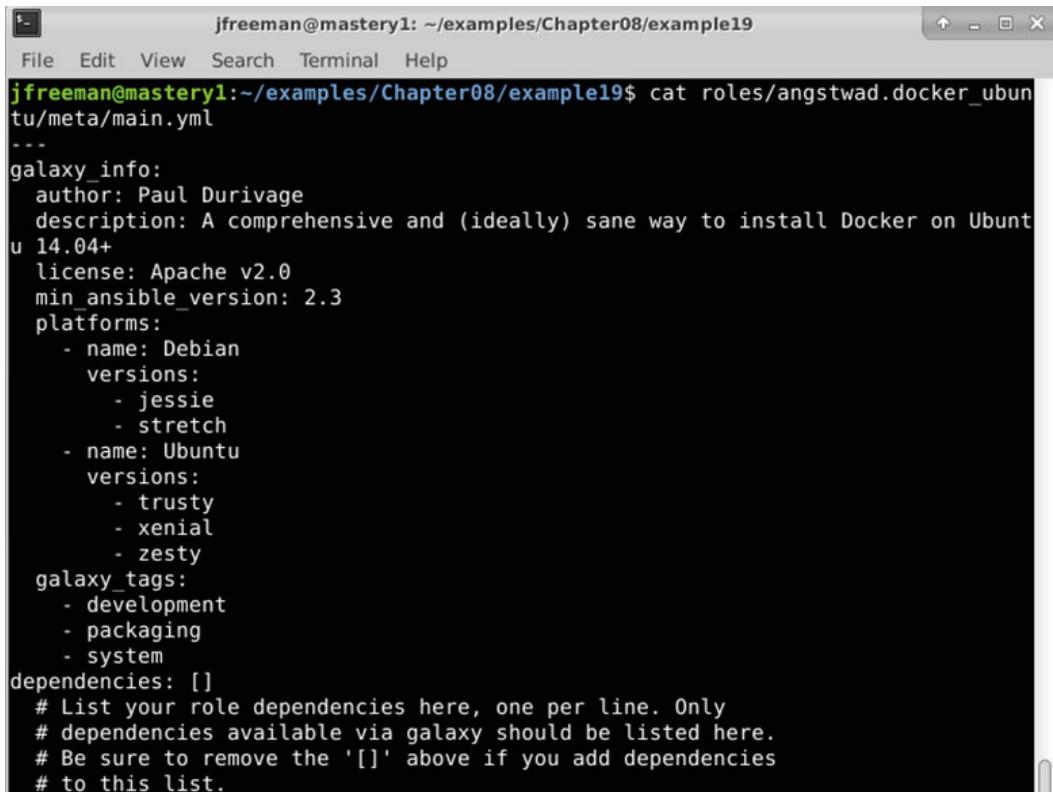
```
jfreeman@mastery1:~/examples/Chapter08/example19$ ansible-galaxy role list -p roles/
# /home/jfreeman/examples/Chapter08/example19/roles
- angstwad.docker_ubuntu, v3.9.0
[WARNING]: - the configured path /home/jfreeman/.ansible/roles does not exist.
[WARNING]: - the configured path /usr/share/ansible/roles does not exist.
[WARNING]: - the configured path /etc/ansible/roles does not exist.
jfreeman@mastery1:~/examples/Chapter08/example19$ ansible-galaxy role info -p roles/ angstwad.docker_ubuntu

Role: angstwad.docker_ubuntu
      description: A comprehensive and (ideally) sane way to install Docker on Ubuntu
      active: True
      commit: 324e8bf9b22d673033911ddfdbdabea6c1261a9f
      commit_message: Explicitly install cli package

The docker-ce package depends on docker-ce-cli unconstrained, this
can lead to problems where the cli package is too new to communicate
with the daemon. This patch adds an explicit install task for the
cli package, so that
      commit_url: https://api.github.com/repos/angstwad/docker.ubuntu/git/commit/324e8bf9b22d673033911ddfdbdabea6c1261a9f
      company:
      created: 2014-01-28T03:13:44.951696Z
      dependencies: []
      download_count: 316320
      forks_count: 213
      galaxy_info:
          author: Paul Durivage
          galaxy_tags: ['development', 'packaging', 'system']
```

Figure 8.23 – Querying installed roles with the `ansible-galaxy` command

The output has been truncated to save space in the book, and there is much more useful information if you scroll through the output. Some of the data being displayed by the `info` command lives within the role itself, in the `meta/main.yml` file. Previously, we've only seen dependency information in this file, and it may not have made much sense to name the directory `meta`, but now we see that other metadata lives in this file as well, as the following screenshot shows:



```
jfreeman@mastery1: ~/examples/Chapter08/example19
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter08/example19$ cat roles/angstwad.docker_ubuntu/meta/main.yml
---
galaxy_info:
  author: Paul Durivage
  description: A comprehensive and (ideally) sane way to install Docker on Ubuntu 14.04+
  license: Apache v2.0
  min_ansible_version: 2.3
  platforms:
    - name: Debian
      versions:
        - jessie
        - stretch
    - name: Ubuntu
      versions:
        - trusty
        - xenial
        - zesty
  galaxy_tags:
    - development
    - packaging
    - system
dependencies: []
# List your role dependencies here, one per line. Only
# dependencies available via galaxy should be listed here.
# Be sure to remove the '[]' above if you add dependencies
# to this list.
```

Figure 8.24 – An example of the metadata that can be placed in the `meta/main.yml` file of a role

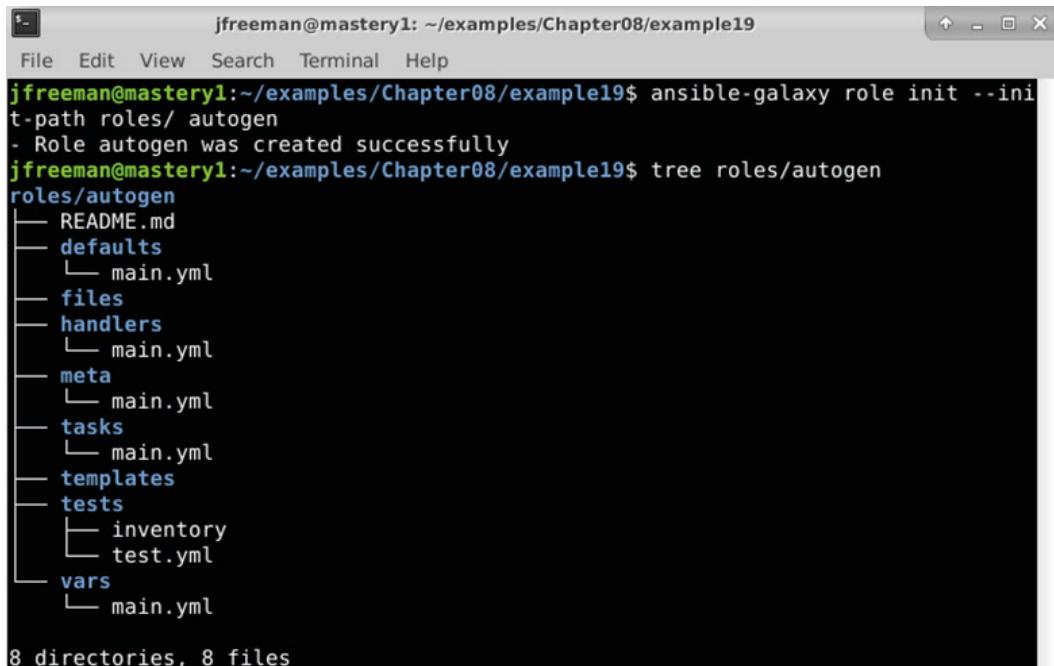
The `ansible-galaxy` tool can also help with the creation of new roles.

The `role init` method will create a skeleton directory tree for the role, as well as populating the `meta/main.yml` file with placeholders for Galaxy-related data.

Let's demonstrate this capability by creating a new role in our working directory named `autogen`, using this command:

```
ansible-galaxy role init --init-path roles/ autogen
```

If you examine the directory structure this command creates, you will see all the directories and placeholder files needed to create a brand-new role, as the following screenshot illustrates:



The terminal window shows the following session:

```
jfreeman@mastery1:~/examples/Chapter08/example19$ ansible-galaxy role init --init-path roles/ autogen
- Role autogen was created successfully
jfreeman@mastery1:~/examples/Chapter08/example19$ tree roles/autogen
roles/autogen
├── 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
```

Figure 8.25 – Creating a skeletal empty role using the ansible-galaxy tool

Note that where we have used the `-p` switch in the past for specifying the local `roles/` directory, we have to use the `--init-path` switch instead with the `init` command. For roles that are not suitable for Ansible Galaxy, such as roles dealing with in-house systems, `ansible-galaxy` can install directly from a Git **Uniform Resource Locator (URL)**. Instead of just providing a role name to the `install` method, a full Git URL with an optional version can be provided. For example, if we wanted to install the `foowhiz` role from our internal Git server, we could simply run the following command:

```
ansible-galaxy role install -p /opt/ansible/roles git+git@git.
internal.site:ansible-roles/foowhiz
```

Without version information, the master branch will be used. Without name data, the name will be determined from the URL itself. To provide a version, append a comma and the version string that Git can understand, such as a tag or branch name—for example, v1, as illustrated here:

```
ansible-galaxy role install -p /opt/ansible/roles git+git@git.  
internal.site:ansible-roles/foowhiz,v1
```

A name for the role can be added with another comma followed by the name string, as illustrated in the following code snippet. If you need to supply a name but do not wish to supply a version, an empty slot is still required for the version:

```
ansible-galaxy role install -p /opt/ansible/roles git+git@git.  
internal.site:ansible-roles/foowhiz,,foo-whiz-common
```

Roles can also be installed directly from tarballs as well, by providing a URL to the tarball in lieu of a full Git URL or a role name to fetch from Ansible Galaxy.

When you need to install many roles for a project, it's possible to define multiple roles to download and install in a YAML-formatted file that ends with .yaml (or .yml). The format of this file allows you to specify multiple roles from multiple sources and retain the ability to specify versions and role names. In addition, the source control method can be listed (currently, only git and hg are supported). You can see an example of this in the following code snippet:

```
---  
- src: <name or url>  
  version: <optional version>  
  name: <optional name override>  
  scm: <optional defined source control mechanism, defaults to  
    git>
```

To install all the roles within a file, use the --roles-file (-r) option with the role install method, as follows:

```
ansible-galaxy role install -r foowhiz-reqs.yaml
```

In this manner, it is very easy to gather all your role dependencies prior to running your playbooks, and whether the roles you need are publicly available on Ansible Galaxy or held in your own internal source control management system, this simple step can greatly speed along playbook deployment while supporting code reuse.

## Summary

Ansible provides the capability to divide content logically into separate files. This capability helps project developers not repeat the same code over and over again. Roles within Ansible take this capability a step further and wrap some magic around the paths to the content. Roles are tunable, reusable, portable, and shareable blocks of functionality. Ansible Galaxy exists as a community hub for developers to find, rate, and share roles as well as collections. The `ansible-galaxy` command-line tool provides a method to interact with the Ansible Galaxy site or other role-sharing mechanisms. These capabilities and tools help with the organization and utilization of common code.

In this chapter, you learned all about inclusion concepts relating to tasks, handlers, variables, and even entire playbooks. Then, you expanded on this knowledge by learning about roles—their structure, setting default variable values, and handling role dependencies. You then proceeded to learn about designing playbooks to utilize roles effectively and applying options such as tags that roles otherwise lack. Finally, you learned about sharing roles across projects using repositories such as Git and Ansible Galaxy.

In the next chapter, we'll cover useful and effective troubleshooting techniques to help you when your Ansible deployments run into trouble.

## Questions

1. Which Ansible module can be used to run tasks from a separate external task file when a playbook is run?
  - a) `ansible.builtin.import`
  - b) `ansible.builtin.include`
  - c) `ansible.builtin.tasks_file`
  - d) `ansible.builtin.with_tasks`
2. Variable data can be passed to an external task file when it is called:
  - a) True
  - b) False
3. The default name of the variable containing the current loop value is:
  - a) `i`
  - b) `loop_var`
  - c) `loop_value`
  - d) `item`

4. When looping over external task files, it is important to consider setting which special variable to prevent loop variable name collisions?
  - a) `loop_name`
  - b) `loop_item`
  - c) `loop_var`
  - d) `item`
5. Handlers are generally run:
  - a) Once, at the end of the play
  - b) Once each, at the end of the `pre_tasks`, `roles/tasks`, and `post_tasks` sections of the play
  - c) Once each, at the end of the `pre_tasks`, `roles/tasks`, and `post_tasks` sections of the play and only when notified
  - d) Once each, at the end of the `pre_tasks`, `roles/tasks`, and `post_tasks` sections of the play and only when imported
6. Ansible can load variables from the following external sources:
  - a) Static `vars_files` inclusion
  - b) Dynamic `vars_files` inclusion
  - c) Through the `include_vars` statement
  - d) Through the `extra-vars` command-line parameter
  - e) All of the above
7. Roles obtain their name from the role directory name (for example, `roles/testrole1` has the name `testrole1`):
  - a) True
  - b) False
8. If a role is missing the `tasks/main.yml` file, Ansible will:
  - a) Abort the play with an error
  - b) Skip the role entirely
  - c) Still reference any other valid parts of the role, including metadata, default variables, and handlers
  - d) Display a warning

9. Roles can have dependencies on other roles:
  - a) True
  - b) False
10. When you specify a tag for a role, Ansible's behavior is to:
  - a) Apply the tag to the entire role
  - b) Apply the tag to each task within the role
  - c) Skip the role entirely
  - d) Only execute tasks from a role with the same tag

# 9

# Troubleshooting Ansible

Ansible is beautifully simple, yet incredibly powerful. The simplicity of Ansible means that its operation is easy to understand and follow. However, even with the simplest and most user-friendly of systems, things do go wrong from time to time—perhaps as we are learning to write our own code (playbooks, roles, modules, or otherwise) and need to debug it, or, more rarely, when we might have found a bug in a released version of a collection or `ansible-core`.

Being able to understand and follow the operation of Ansible is critically important when debugging unexpected behavior, wherever it may arise. Ansible provides a number of options and tools to help you troubleshoot the operation of its core components, as well as your own playbook code. We will explore these in detail in this chapter, with the goal of empowering you to troubleshoot your own Ansible work with confidence.

Specifically, in this chapter, we will look at the following topics:

- Playbook logging and verbosity
- Variable introspection
- Debugging code execution

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running **Ansible 4.3** or a newer version. Almost any flavor of Linux should do—for those interested in specifics, all the code presented in this chapter was tested on Ubuntu Server 20.04 Long-Term Support (LTS) unless stated otherwise, and on Ansible 4.3. The example code that accompanies this chapter can be downloaded from GitHub at this link: <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter09>.

Check out the following video to see the Code in Action: <https://bit.ly/2Xx46Ym>

## Playbook logging and verbosity

Increasing the verbosity of Ansible output can solve many problems. From invalid module arguments to incorrect connection commands, increased verbosity can be critical in pinpointing the source of an error. Playbook logging and verbosity were briefly discussed in *Chapter 3, Protecting Your Secrets with Ansible*, with regard to protecting secret values while executing playbooks. This section will cover verbosity and logging in further detail.

### Verbosity

When executing playbooks with `ansible-playbook`, the output is displayed on **standard output (stdout)**. With the default level of verbosity, very little information is displayed. As a play is executed, `ansible-playbook` will print a **play** header with the name of the play. Then, for each task, a **task** header is printed with the name of the task. As each host executes the task, the name of the host is displayed along with the task state, which can be `ok`, `fatal`, or `changed`. No further information about the task is displayed—such as the module being executed, the arguments provided to the module, or the return data from the execution. While this is fine for well-established playbooks, I tend to want a little more information about my plays. In a few of the earlier examples in this book, we used higher levels of verbosity, up to a level of two (`-vv`), so that we could see the location of the task and return data. There are five total levels of verbosity, as outlined here:

- **None:** The default level
- **One (-v):** Where the return data and conditional information is displayed
- **Two (-vv):** For task location and handler notification information

- **Three** (-vvv): Provides details of the connection attempts and task invocation information
- **Four** (-vvvv): Passes along extra verbosity options to the connection plugins (such as passing -vvv to the ssh commands)

Increasing the verbosity can help pinpoint where errors might be occurring, as well as providing extra insight into how Ansible is performing its operations.

As we mentioned in *Chapter 3, Protecting Your Secrets with Ansible*, verbosity beyond level one can leak sensitive data to standard out and log files, so care should be taken when using increased verbosity in a potentially shared environment.

## Logging

While the default is for `ansible-playbook` to log to `stdout`, the amount of output may be greater than the buffer of the terminal emulator being used; therefore, it may be necessary to save all the output to a file. While various shells provide some mechanism to redirect output, a more elegant solution is to direct `ansible-playbook` to log to a file. This is accomplished by way of either a `log_path` definition in the `ansible.cfg` file or by setting `ANSIBLE_LOG_PATH` as an environment variable. The value of either should be the path to a file. If the path does not exist, Ansible will attempt to create a file. If the file does exist, Ansible will append to the file, allowing consolidation of multiple `ansible-playbook` execution logs.

The use of a log file is not mutually exclusive with logging to `stdout`. Both can happen at the same time, and the verbosity level that's provided has an effect on both, simultaneously. Logging is of course helpful, but it doesn't necessarily tell us what's going on in our code, and what our variables might contain. We'll look at how to perform variable introspection in the next section to help you with this very task.

## Variable introspection

A common set of problems that are encountered when developing Ansible playbooks is the improper use, or invalid assumption, of the value of variables. This is particularly common when registering the results of one task in a variable, and later using that variable in a task or template. If the desired element of the result is not accessed properly, the end result will be unexpected, or perhaps even harmful.

To troubleshoot improper variable usage, an inspection of the variable value is the key. The easiest way to inspect a variable's value is with the `ansible.builtin.debug` module. The `ansible.builtin.debug` module allows the display of freeform text on screen, and as with other tasks, the arguments to the module can take advantage of the Jinja2 template syntax as well. Let's demonstrate this usage by creating a sample play that executes a task, registers the result, and then shows the result in an `ansible.builtin.debug` statement using Jinja2 syntax to render the variable, as follows:

```
---
- name: variable introspection demo
  hosts: localhost
  gather_facts: false

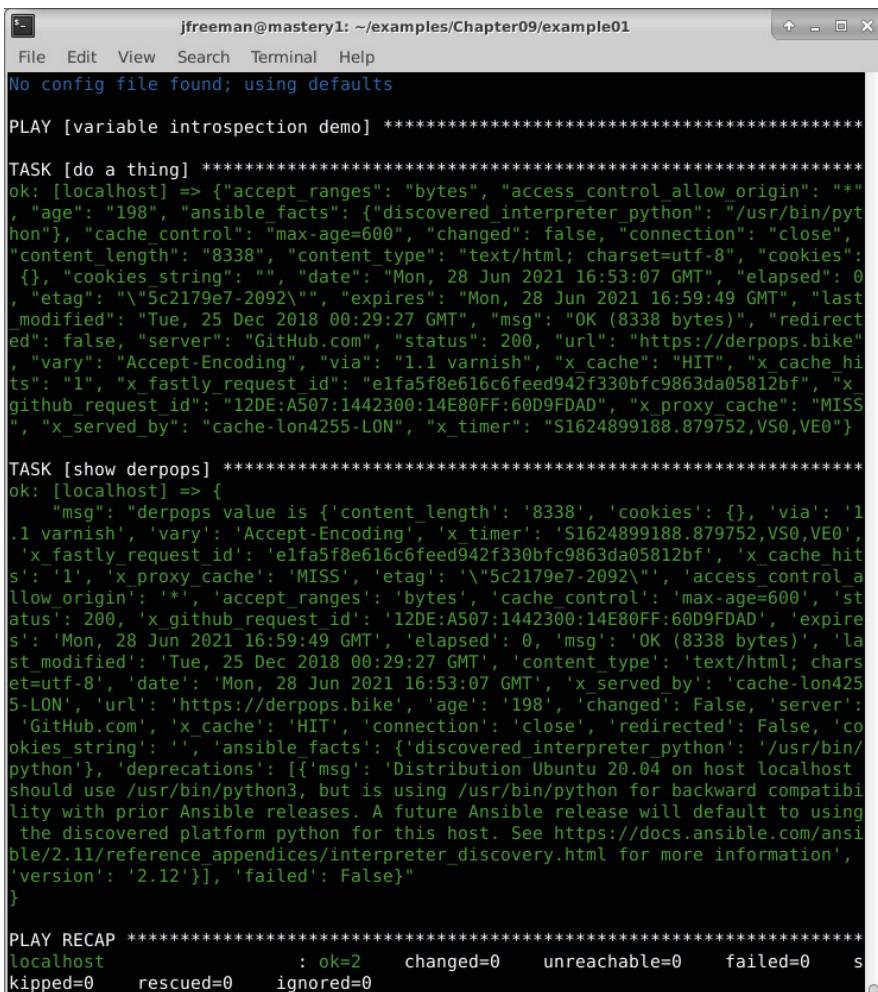
  tasks:
    - name: do a thing
      ansible.builtin.uri:
        url: https://derpops.bike
      register: derpops

    - name: show derpops
      ansible.builtin.debug:
        msg: "derpops value is {{ derpops }}"
```

We will run this play with level one verbosity using the following command:

```
ansible-playbook -i mastery-hosts vintro.yaml -v
```

Assuming the website we're testing against is accessible, we'll see a displayed value for derpops, as shown in the following screenshot:



```
jfreeman@mastery1: ~/examples/Chapter09/example01
File Edit View Search Terminal Help
No config file found; using defaults

PLAY [variable introspection demo] *****

TASK [do a thing] *****
ok: [localhost] => {"accept_ranges": "bytes", "access_control_allow_origin": "*",
"age": "198", "ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "cache_control": "max-age=600", "changed": false, "connection": "close", "content_length": "8338", "content_type": "text/html; charset=utf-8", "cookies": {}, "cookies_string": "", "date": "Mon, 28 Jun 2021 16:53:07 GMT", "elapsed": 0, "etag": "\\"5c2179e7-2092\\\"", "expires": "Mon, 28 Jun 2021 16:59:49 GMT", "last_modified": "Tue, 25 Dec 2018 00:29:27 GMT", "msg": "OK (8338 bytes)", "redirected": false, "server": "GitHub.com", "status": 200, "url": "https://derpops.bike", "vary": "Accept-Encoding", "via": "1.1 varnish", "x_cache": "HIT", "x_cache_hits": "1", "x_fastly_request_id": "e1fa5f8e616c6feed942f330bfc9863da05812bf", "x.github_request_id": "12DE:A507:1442300:14E80FF:60D9FDAD", "x_proxy_cache": "MISS", "x_served_by": "cache-lon4255-LON", "x_timer": "S1624899188.879752,VS0,VE0"}

TASK [show derpops] *****
ok: [localhost] => {
    "msg": "derpops value is {'content_length': '8338', 'cookies': {}, 'via': '1.1 varnish', 'vary': 'Accept-Encoding', 'x_timer': '51624899188.879752,VS0,VE0', 'x_fastly_request_id': 'e1fa5f8e616c6feed942f330bfc9863da05812bf', 'x_cache_hits': '1', 'x_proxy_cache': 'MISS', 'etag': '\\"5c2179e7-2092\\\"", 'access_control_allow_origin': '*', 'accept_ranges': 'bytes', 'cache_control': 'max-age=600', 'status': 200, 'x.github_request_id': '12DE:A507:1442300:14E80FF:60D9FDAD', 'expires': 'Mon, 28 Jun 2021 16:59:49 GMT', 'elapsed': 0, 'msg': 'OK (8338 bytes)', 'last_modified': 'Tue, 25 Dec 2018 00:29:27 GMT', 'content_type': 'text/html; charset=utf-8', 'date': 'Mon, 28 Jun 2021 16:53:07 GMT', 'x_served_by': 'cache-lon4255-LON', 'url': 'https://derpops.bike', 'age': '198', 'changed': False, 'server': 'GitHub.com', 'x_cache': 'HIT', 'connection': 'close', 'redirected': False, 'cookies_string': "", 'ansible_facts': {'discovered_interpreter_python': '/usr/bin/python'}, 'deprecations': [{"msg": "Distribution Ubuntu 20.04 on host localhost should use /usr/bin/python3, but is using /usr/bin/python for backward compatibility with prior Ansible releases. A future Ansible release will default to using the discovered platform python for this host. See https://docs.ansible.com/ansible/2.11/reference_appendices/interpreter_discovery.html for more information", "version": "2.12"}], 'failed': False}
}

PLAY RECAP *****
localhost                  : ok=2      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 9.1 – Using level one verbosity to examine registered variable values

The `ansible.builtin.debug` module has a different option that may be useful as well. Instead of printing a freeform string to debug template usage, the module can simply print the value of any variable. This is done using the `var` argument instead of the `msg` argument. Let's repeat our example, but this time we'll use the `var` argument, and we'll access just the `server` subelement of the `derpops` variable, as follows:

```
---  
- name: variable introspection demo
```

```

hosts: localhost
gather_facts: false

tasks:
  - name: do a thing
    ansible.builtin.uri:
      url: https://derpops.bike
    register: derpops

  - name: show derpops
    ansible.builtin.debug:
      var: derpops.server

```

Running this modified play with the same level of verbosity as before will show just the `server` portion of the `derpops` variable, as demonstrated in the following screenshot:

```

jfreeman@mastery1:~/examples/Chapter09/example02$ ansible-playbook -i mastery-hosts vintro.yaml -v
No config file found; using defaults

PLAY [variable introspection demo] ****
TASK [do a thing] ****
ok: [localhost] => {"accept_ranges": "bytes", "access_control_allow_origin": "", "age": "575", "ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "cache_control": "max-age=600", "changed": false, "connection": "close", "content_length": "8338", "content_type": "text/html; charset=utf-8", "cookies": {}, "cookies_string": "", "date": "Mon, 28 Jun 2021 16:59:07 GMT", "elapsed": 0, "etag": "\\"5c2179e7-2092\\\"", "expires": "Mon, 28 Jun 2021 16:59:32 GMT", "last_modified": "Tue, 25 Dec 2018 00:29:27 GMT", "msg": "OK (8338 bytes)", "redirected": false, "server": "GitHub.com", "status": 200, "url": "https://derpops.bike", "vary": "Accept-Encoding", "via": "1.1 varnish", "x_cache": "HIT", "x_cache_hits": "1", "x_fastly_request_id": "a9aa64a1629112faf8042bcf13a88c0b5ab41452", "x_github_request_id": "E064:68DA:1454D3E:14FE84C:60D9FD9C", "x_proxy_cache": "MISS", "x_served_by": "cache-lcy19272-LCY", "x_timer": "S1624899548.858726,VS0,VE1"}
```

```

TASK [show derpops] ****
ok: [localhost] => {
    "derpops.server": "GitHub.com"
}

PLAY RECAP ****
localhost                  : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

```

Figure 9.2 – Using the `var` parameter of the `debug` module to inspect variable subelements

In our example that used the `msg` argument to `ansible.builtin.debug`, the variable needed to be expressed inside curly brackets, but when using `var`, it did not. This is because `msg` expects a string, and so Ansible needs to render the variable as a string via the template engine. However, `var` expects a single unrendered variable.

## Variable subelements

Another frequent mistake in playbooks is to improperly reference a subelement of a complex variable. A complex variable is more than simply a string—it is either a list or a hash. Often, the wrong subelement will be referenced, or the element will be improperly referenced, expecting a different type.

While lists are fairly easy to work with, hashes present some unique challenges. A hash is an unordered key-value set of potentially mixed types, which could also be nested. A hash can have one element that is a single string, while another element can be a list of strings, and a third element can be another hash with further elements inside it. Knowing how to properly access the right subelement is critical to success.

For example, let's modify our previous play a bit more. This time, we'll allow Ansible to gather facts, and then we'll show the value of `ansible_python`. Here's the code we'll need:

```
---
- name: variable introspection demo
  hosts: localhost

  tasks:
    - name: show a complex hash
      ansible.builtin.debug:
        var: ansible_python
```

Run this code with level one verbosity, and you should see the following output:

```
jfreeman@mastery1: ~/examples/Chapter09/example03$ ansible-playbook -i mastery-hosts vintro.yaml -v
No config file found; using defaults

PLAY [variable introspection demo] ****
TASK [Gathering Facts] ****
ok: [localhost]

TASK [show a complex hash] ****
ok: [localhost] => {
  "ansible_python": {
    "executable": "/usr/bin/python",
    "has_sslcontext": true,
    "type": "CPython",
    "version": {
      "major": 2,
      "micro": 18,
      "minor": 7,
      "releaselevel": "final",
      "serial": 0
    },
    "version_info": [
      2,
      7,
      18,
      "final",
      0
    ]
  }
}

PLAY RECAP ****
localhost                  : ok=2      changed=0      unreachable=0      failed=0      s
kiped=0      rescued=0      ignored=0
```

Figure 9.3 – Inspecting the `ansible_python` fact subelement using `ansible.builtin.debug`

Using `ansible.builtin.debug` to display an entire complex variable is a great way to learn all the names of the subelements.

This variable has elements that are strings, along with elements that are lists of strings. Let's access the last item in the list of flags, as follows:

```
---
- name: variable introspection demo
  hosts: localhost

  tasks:
```

```
- name: show a complex hash
  ansible.builtin.debug:
    var: ansible_python.version_info[-1]
```

The output is shown here:

```
jfreeman@mastery1:~/examples/Chapter09/example04
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter09/example04$ ansible-playbook -i mastery-hosts vintro.yaml -v
No config file found; using defaults

PLAY [variable introspection demo] ****
TASK [Gathering Facts] ****
ok: [localhost]

TASK [show a complex hash] ****
ok: [localhost] => {
  "ansible_python.version_info[-1]": "0"
}

PLAY RECAP ****
localhost                  : ok=2    changed=0    unreachable=0    failed=0    s
kipped=0      rescued=0    ignored=0
```

Figure 9.4 – Inspecting the `ansible_python` fact subelement further

Because `ansible_python.version_info` is a list, we can use the **list index method** to select a specific item from the list. In this case, `-1` will give us the very last item in the list.

## Subelements versus the Python object method

A less common but confusing gotcha comes from a quirk of the Jinja2 syntax. Complex variables within Ansible playbooks and templates can be referenced in two ways. The first style is to reference the base element by the name, followed by a bracket, and the subelement within quotes inside the brackets. This is the **standard subscript syntax**. For example, to access the `herp` subelement of the `derp` variable, we will use the following code:

```
{{ derp['herp'] }}
```

The second style is a convenience method that Jinja2 provides, which is to use a period to separate the elements. This is called **dot notation**, and it looks like this:

```
{{ derp.herp }}
```

There is a subtle difference in how these styles work, and it has to do with Python objects and object methods. As Jinja2 is, at its heart, a Python utility, variables in Jinja2 have access to their native Python methods. A string variable has access to Python string methods, a list has access to list methods, and a dictionary has access to dictionary methods. When using the first style, Jinja2 will first search the element for a subelement of the provided name. If no subelements are found, Jinja2 will then attempt to access a Python method of the provided name. However, the order is reversed when using the second style; first, a Python object method is searched for, and if not found, then a subelement is searched for. This difference matters when there is a name collision between a subelement and a method. Imagine a variable named `derp`, which is a complex variable. This variable has a subelement named `keys`. Using each style to access the `keys` element will result in different values. Let's build a playbook to demonstrate this, as follows:

```
---
```

```
- name: sub-element access styles
```

```
hosts: localhost
```

```
gather_facts: false
```

```
vars:
```

```
  - derp:
```

```
    keys:
```

```
      - c
```

```
      - d
```

```
tasks:
```

```
  - name: subscript style
```

```
    ansible.builtin.debug:
```

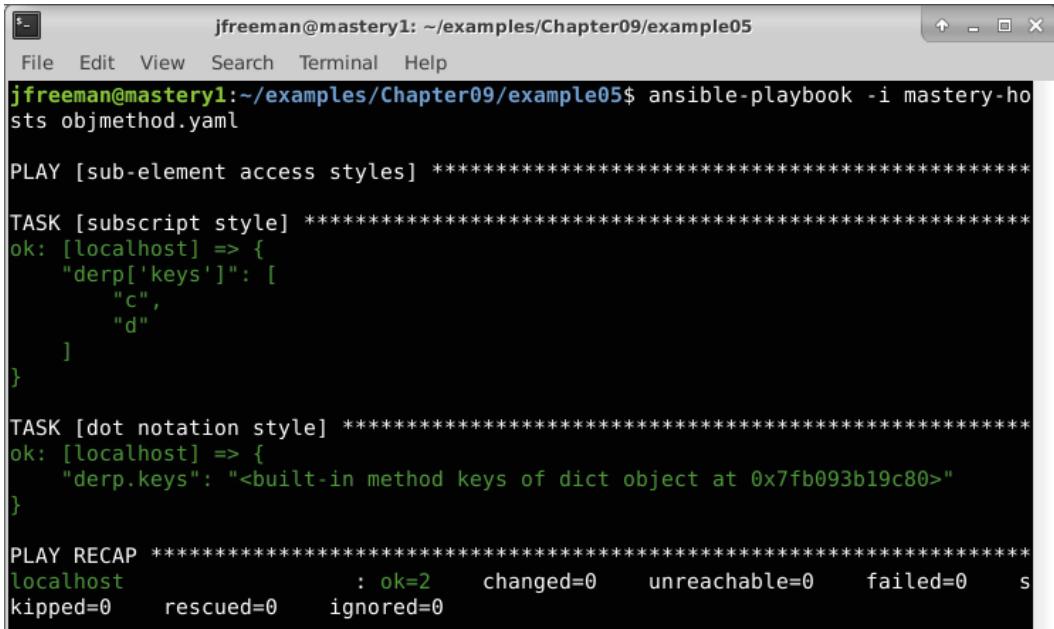
```
      var: derp['keys']
```

```
  - name: dot notation style
```

```
    ansible.builtin.debug:
```

```
      var: derp.keys
```

When running this play, we can clearly see the difference between the two styles. The first style successfully references the `keys` subelement, while the second style references the `keys` method of Python dictionaries, as the following screenshot illustrates:



```
jfreeman@mastery1: ~/examples/Chapter09/example05
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter09/example05$ ansible-playbook -i mastery-hosts objmethod.yaml

PLAY [sub-element access styles] ****
TASK [subscript style] ****
ok: [localhost] => {
    "derp['keys']": [
        "c",
        "d"
    ]
}

TASK [dot notation style] ****
ok: [localhost] => {
    "derp.keys": "<built-in method keys of dict object at 0x7fb093b19c80>"
}

PLAY RECAP ****
localhost                  : ok=2      changed=0      unreachable=0      failed=0      s
kiped=0      rescued=0      ignored=0
```

Figure 9.5 – Demonstrating the difference between standard subscript syntax and dot notation when name collision occurs

Generally, it's best to avoid using subelement names that conflict with Python object methods. However, if that's not possible, the next best thing to do is to be aware of the difference in subelement reference styles and choose the appropriate one.

Of course, variables are only one aspect of playbook behavior—sometimes, we need to actually get into debugging the code itself, and we'll look at just that in the next section.

## Debugging code execution

Sometimes, the logging and inspection of variable data are not enough to troubleshoot a problem. When this happens, it can be necessary to interactively debug the playbook, or to dig deeper into the internals of Ansible code. There are two main sets of Ansible code: code that runs locally on the Ansible host, and module code that runs remotely on the target host.

## Playbook debugging

Playbooks can be interactively debugged by using an execution strategy that was introduced in Ansible 2.1, **the debug strategy**. If a play uses this strategy when an error state is encountered, an interactive debugging session starts. This interactive session can be used to display variable data, display task arguments, update task arguments, update variables, redo task execution, continue execution, or exit the debugger.

Let's demonstrate this with a play that has a successful task, followed by a task with an error, followed by a final successful task. We'll reuse the playbook we've been using, but update it a bit, as shown in the following code:

```
--  
- name: sub-element access styles  
  hosts: localhost  
  gather_facts: false  
  strategy: debug  
  
vars:  
  - derp:  
    keys:  
      - c  
      - d  
  
tasks:  
  - name: subscript style  
    ansible.builtin.debug:  
      var: derp['keys']  
  
  - name: failing task  
    ansible.builtin.debug:  
      msg: "this is {{ derp['missing'] }}"  
  
  - name: final task  
    ansible.builtin.debug:  
      msg: "my only friend the end"
```

Upon execution, Ansible will encounter an error in our failing task and present the (debug) prompt, as shown in the following screenshot:

```
jfreeman@mastery1: ~/examples/Chapter09/example06
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter09/example06$ ansible-playbook -i mastery-ho
sts objmethod.yaml

PLAY [sub-element access styles] ****
TASK [subscript style] ****
ok: [localhost] => {
    "derp['keys']": [
        "c",
        "d"
    ]
}

TASK [failing task] ****
fatal: [localhost]: FAILED! => {"msg": "The task includes an option with an unde
fined variable. The error was: 'dict object' has no attribute 'missing'\n\nThe e
rror appears to be in '/home/jfreeman/examples/Chapter09/example06/objmethod.yam
l': line 18, column 7, but may\\nbe elsewhere in the file depending on the exact
syntax problem.\n\nThe offending line appears to be:\n      - name: failing ta
sk\n        ^ here\n"}
[localhost] TASK: failing task (debug)>
```

Figure 9.6 – The Ansible debugger starting during a failed task execution (when execution strategy is debug)

From this prompt, we can display the task and the arguments to the task by using the p command, as the following screenshot shows:

```
[localhost] TASK: failing task (debug)> p task
TASK: failing task
[localhost] TASK: failing task (debug)> p task.args
{'msg': "this is {{ derp['missing'] }}"}
[localhost] TASK: failing task (debug)>
```

Figure 9.7 – Using the p command to inspect details of the failed play task

We can also change the playbook on the fly to try different arguments or variable values. Let's define the missing key of the `derp` variable, and then retry the execution. All of the variables are within the top-level `vars` dictionary. We can directly set the variable data using Python syntax and the `task_vars` command, and then retry with the `r` command, as illustrated in the following screenshot:

```
[localhost] TASK: failing task (debug)> task_vars['derp']['missing'] = "the end"
[localhost] TASK: failing task (debug)> r
ok: [localhost] => {
    "msg": "this is the end"
}

TASK [final task] *****
ok: [localhost] => {
    "msg": "my only friend the end"
}

PLAY RECAP *****
localhost                  : ok=2      changed=0      unreachable=0      failed=0      s
kiped=0      rescued=0      ignored=0
```

Figure 9.8 – Adding previously undefined variable values and retrying the play from the debugger

The debug execution strategy is a handy tool for quickly iterating through different task arguments and variable combinations to figure out the correct path forward. However, because errors result in interactive consoles, the debug strategy is inappropriate for automated executions of playbooks, as there is no human on the console to manipulate the debugger.

#### Important Point

Changing data within the debugger will not save the changes to backing files. Always remember to update playbook files to reflect discoveries that are made during debugging.

## Debugging local code

The local Ansible code is the lion's share of the code that comes with Ansible. All the playbook, play, role, and task parsing code lives locally. All of the task result processing code and transport code lives locally. All of the code, except for the assembled module code that is transported to the remote host, lives locally.

Local Ansible code can still be broken down into three major sections: **inventory**, **playbook**, and **executor**. Inventory code deals with parsing inventory data from host files, dynamic inventory scripts, or combinations of the two, in directory trees. Playbook code is used to parse the playbook **YAML Ain't Markup Language** (YAML) code into Python objects within Ansible. Executor code is the core **application programming interface** (API) and deals with forking processes, connecting to hosts, executing modules, handling results, and most other things. Learning the general area to start debugging comes with practice, but the general areas that are described here are a starting point.

As Ansible is written in Python, the tool for debugging local code execution is the `pdb` Python debugger. This tool allows us to insert breakpoints inside the Ansible code and interactively walk through the execution of the code, line by line. This is very useful for examining the internal state of Ansible as the local code executes. Many books and websites cover the usage of `pdb`, and these can be found with a simple web search for an introduction to Python `pdb`, so we will not repeat them here. If you are looking for a hands-on introduction to using `pdb`, there are lots of great examples in the book *Django 1.1 Testing and Debugging*, Karen M. Tracey, Packt Publishing, which will enable you to practice real-world debugging techniques with `pdb` in Django (which is written in Python). The official Python documentation also offers much in the way of information on using the debugger. You can view this here: <https://docs.python.org/3/library/pdb.html>. The basics are to edit the source file to be debugged, insert a new line of code to create a breakpoint, and then execute the code. Code execution will stop where the breakpoint was created, and a prompt will be provided to explore the code state.

Of course, Ansible has lots of different components that come together to build up its functionality, from inventory handling code to the actual playbook execution engine itself. It is possible to add breakpoints and debugging to all of these places to help resolve issues you might be facing, though the files you need to edit are slightly different in each case. We'll look at the details of the most common aspects of the Ansible code you might need to debug in the following subsections of this chapter.

## Debugging inventory code

Inventory code deals with finding inventory sources, reading or executing the discovered files, parsing the inventory data into inventory objects, and loading variable data for the inventory. To debug how Ansible will deal with an inventory, a breakpoint must be added inside `inventory/__init__.py` or one of the other files within the `inventory/` subdirectory. This directory will be located on the local filesystem wherever Ansible has been installed. As most installations of Ansible 4.0 will have been performed via pip at this time, the exact path of your installation will vary greatly, depending on factors such as whether you used a virtual environment, whether you installed Ansible in your user directory, or whether you used sudo to install Ansible system-wide. As an example, on my Ubuntu 20.04 test system, this file may be found in the `/usr/local/lib/python3.8/dist-packages/ansible/inventory` path. To help you discover where Ansible is installed, simply type `which ansible` from the command line. This command will show you where the Ansible executable is installed and may indicate a Python virtual environment. For this book, Ansible has been installed as root using the operating system Python distribution, with the Ansible binaries located in `/usr/local/bin/`.

To discover the path to the Ansible Python code, simply type `python3 -c "import ansible; print(ansible)"`. Note that, like me, you might have both Python 2 and Python 3 installed—if you are unsure of which version of Python Ansible is running under, you will need to execute both the version 2 and 3 binaries in order to discover your module locations.

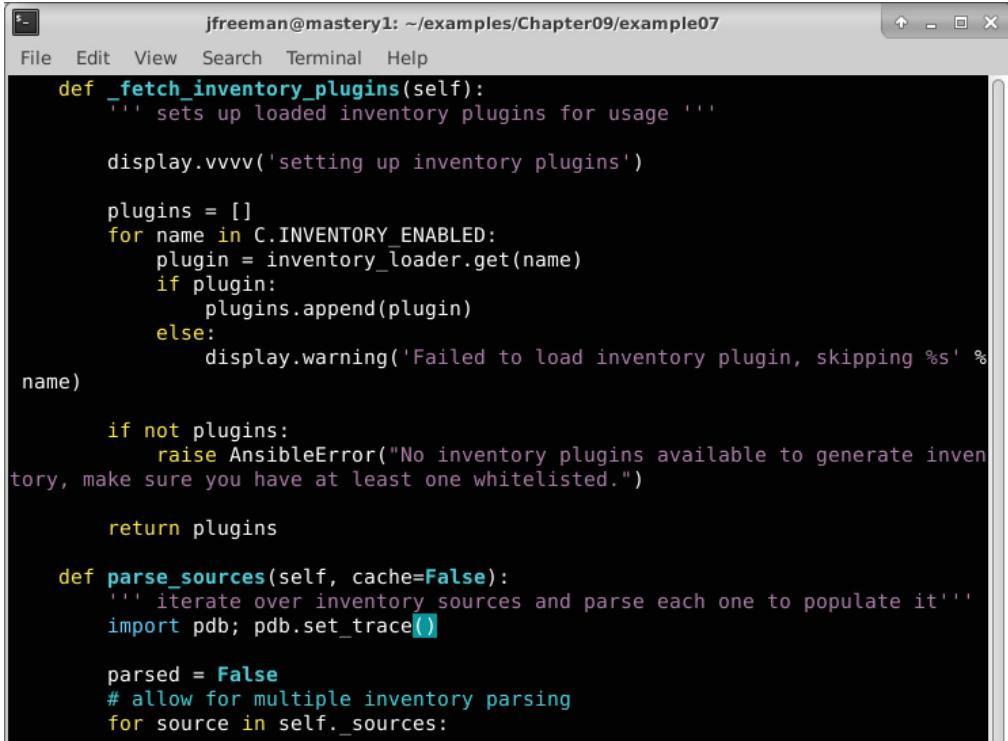
On my system, this shows `<module 'ansible' from '/usr/local/lib/python3.8/dist-packages/ansible/__init__.py'>`, from which we can deduce that the inventory subdirectory is located at `/usr/local/lib/python3.8/dist-packages/ansible/inventory/`.

The inventory directory was restructured in later releases of Ansible, and in version 4.0, we need to look in `inventory/manager.py`. Note that this file comes from the `ansible-core` package and not the `ansible` package that depends upon it.

Within this file, there is a class definition for the `Inventory` class. This is the inventory object that will be used throughout a playbook run, and it is created when `ansible-playbook` parses the options provided to it for an inventory source. The `__init__` method of the `Inventory` class does all the inventory discovery, parsing, and variable loading. To troubleshoot an issue in those three areas, a breakpoint should be added within the `__init__()` method. A good place to start would be after all of the class variables are given an initial value, and just before any data is processed.

In version 2.11.1 of ansible-core, this would be line 167 of `inventory/manager.py`, where the `parse_sources` function is called.

We can skip down to the `parse_sources` function definition on line 215 to insert our breakpoint. To insert a breakpoint, we must first import the `pdb` module and then call the `set_trace()` function, as shown in the following screenshot:



```
jfreeman@mastery1: ~/examples/Chapter09/example07
File Edit View Search Terminal Help
def _fetch_inventory_plugins(self):
    """ sets up loaded inventory plugins for usage """

    display.vvvv('setting up inventory plugins')

    plugins = []
    for name in C.INVENTORY_ENABLED:
        plugin = inventory_loader.get(name)
        if plugin:
            plugins.append(plugin)
        else:
            display.warning('Failed to load inventory plugin, skipping %s' %
name)

    if not plugins:
        raise AnsibleError("No inventory plugins available to generate inventory, make sure you have at least one whitelisted.")

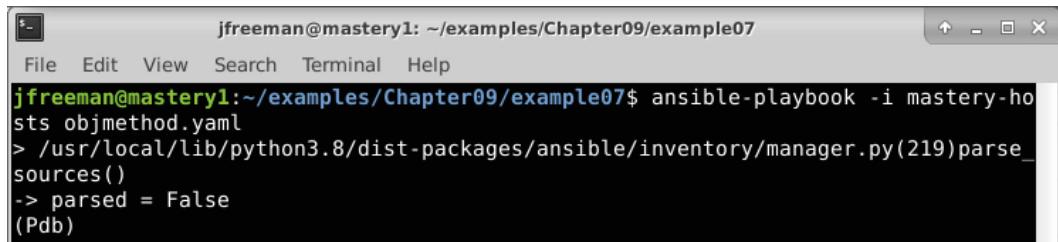
    return plugins

def parse_sources(self, cache=False):
    """ iterate over inventory sources and parse each one to populate it """
    import pdb; pdb.set_trace()

    parsed = False
    # allow for multiple inventory parsing
    for source in self._sources:
```

Figure 9.9 – Adding a pdb breakpoint into the ansible-core inventory manager code

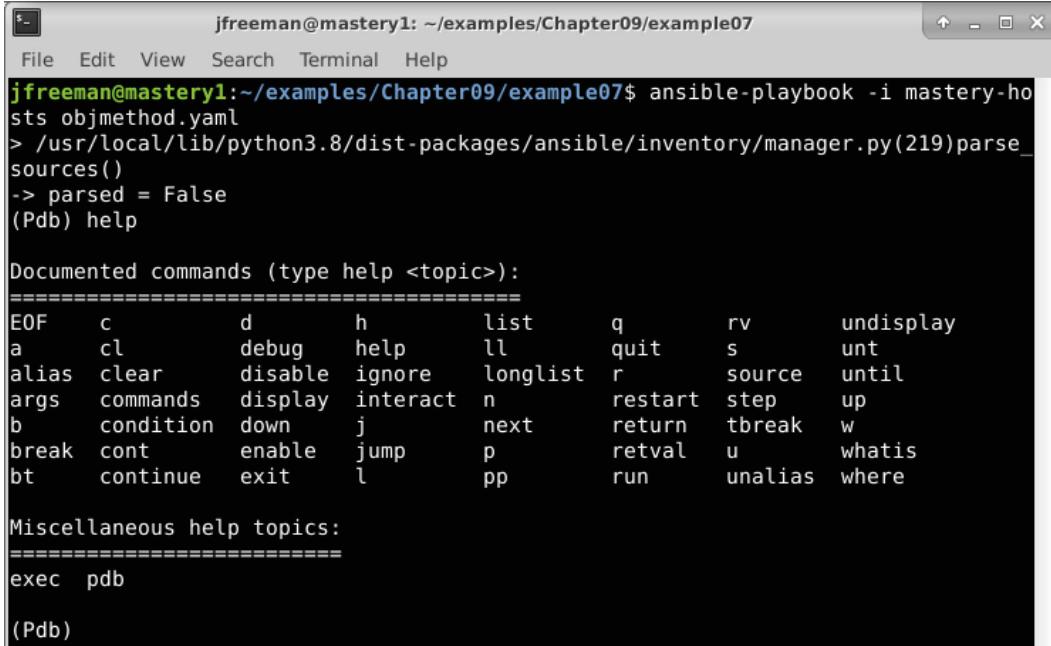
To start debugging, save the source file and then execute `ansible-playbook` as normal. When the breakpoint is reached, the execution will stop and a `pdb` prompt will be displayed, as shown in the following screenshot:



```
jfreeman@mastery1:~/examples/Chapter09/example07$ ansible-playbook -i mastery-hosts objmethod.yaml
> /usr/local/lib/python3.8/dist-packages/ansible/inventory/manager.py(219)parse_
sources()
-> parsed = False
(Pdb)
```

Figure 9.10 – Ansible reaching a pdb breakpoint as it starts to set up the inventory for our play

From here, we can issue any number of debugger commands, such as the `help` command, as the following screenshot shows:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter09/example07". The user has run the command "ansible-playbook -i mastery-hosts objmethod.yaml" which triggered a pdb session. The user then typed "(Pdb) help" to see the available commands. The output shows documented commands, miscellaneous help topics, and the current pdb command.

```
jfreeman@mastery1:~/examples/Chapter09/example07$ ansible-playbook -i mastery-hosts objmethod.yaml
> /usr/local/lib/python3.8/dist-packages/ansible/inventory/manager.py(219)parse_sources()
-> parsed = False
(Pdb) help

Documented commands (type help <topic>):
=====
EOF      c          d          h          list      q          rv         undisplay
a          cl         debug     help      ll        quit      s          unt
alias    clear     disable   ignore   longlist r        source   until
args     commands  display  interact n        restart  step     up
b        condition down     j        next     return   tbreak   w
break   cont      enable   jump     p        retval   u        whatis
bt      continue exit     l        pp       run     unalias where
bt

Miscellaneous help topics:
=====
exec    pdb

(Pdb)
```

Figure 9.11 – Demonstrating the help command of the pdb debugger

The `where` and `list` commands can help us determine where we are in the stack and where we are in the code, as illustrated in the following screenshot:

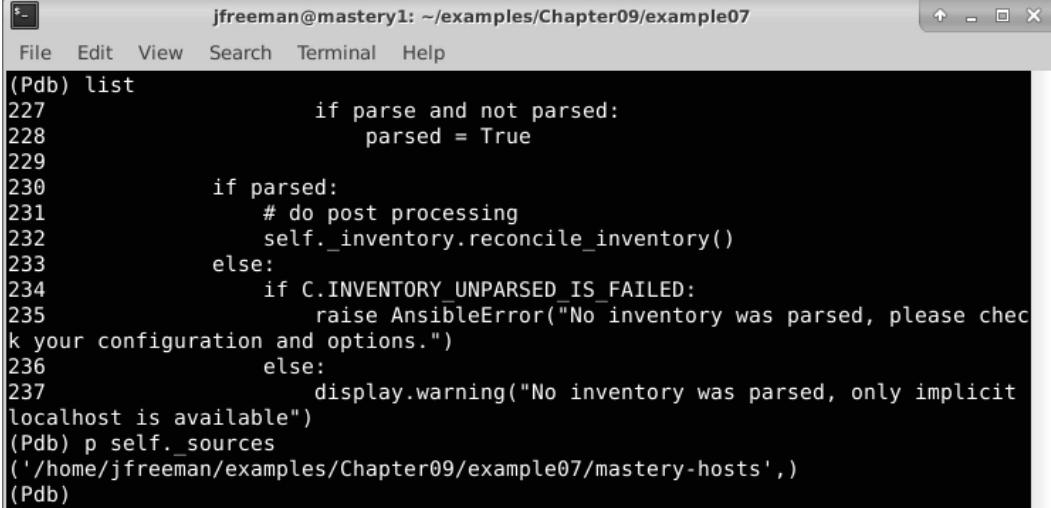
```
jfreeman@mastery1: ~/examples/Chapter09/example07
File Edit View Search Terminal Help
(Pdb) where
  /usr/local/bin/ansible-playbook(135)<module>()
-> exit_code = cli.run()
  /usr/local/lib/python3.8/dist-packages/ansible/cli/playbook.py(118)run()
-> loader, inventory, variable_manager = self._play_prereqs()
  /usr/local/lib/python3.8/dist-packages/ansible/cli/_init_.py(470)_play_prereqs()
-> inventory = InventoryManager(loader=loader, sources=options['inventory'])
  /usr/local/lib/python3.8/dist-packages/ansible/inventory/manager.py(167)__init__()
-> self.parse_sources(cache=True)
> /usr/local/lib/python3.8/dist-packages/ansible/inventory/manager.py(219)parse_sources()
-> parsed = False
(Pdb) list
214
215     def parse_sources(self, cache=False):
216         ''' iterate over inventory sources and parse each one to populate it'''
217         import pdb; pdb.set_trace()
218
219 ->     parsed = False
220     # allow for multiple inventory parsing
221     for source in self._sources:
222
223         if source:
224             if ',' not in source:
(Pdb)
```

Figure 9.12 – Demonstrating the where and list pdb commands

The `where` command shows us that we're in `inventory/manager.py` in the `parse_sources()` method. The next frame up is the same file—the `__init__()` function. Before that is a different file, the `playbook.py` file, and the function in that file is `run()`. This line calls `ansible.inventory.InventoryManager` to create an `inventory` object. Before that is the original file, `ansible-playbook`, calling `cli.run()`.

The `list` command shows the source code around our current point of execution, five lines before and five lines after.

From here, we can guide pdb through the function line by line with the `next` command, and if we choose to, we can trace into other function calls with the `step` command. We can also print variable data to inspect values, as shown in the following screenshot:



```
jfreeman@mastery1: ~/examples/Chapter09/example07
File Edit View Search Terminal Help
(Pdb) list
227             if parse and not parsed:
228                 parsed = True
229
230             if parsed:
231                 # do post processing
232                 self._inventory.reconcile_inventory()
233             else:
234                 if C.INVENTORY_UNPARSED_IS_FAILED:
235                     raise AnsibleError("No inventory was parsed, please check your configuration and options.")
236                 else:
237                     display.warning("No inventory was parsed, only implicit localhost is available")
(Pdb) p self._sources
('/home/jfreeman/examples/Chapter09/example07/mastery-hosts',)
(Pdb)
```

Figure 9.13 – Demonstrating the `print` command to analyze variable values during execution

We can see that the `self._sources` variable has a full path of our `mastery-hosts` inventory file, which is the string we gave `ansible-playbook` for our inventory data. We can continue to walk through or jump around, or just use the `continue` command to run until the next breakpoint or the completion of the code.

## Debugging playbook code

Playbook code is responsible for loading, parsing, and executing playbooks. The main entry point for playbook handling is found by locating the Ansible path, just as we did in the *Debugging inventory code* section, and then locating the `playbook/_init_.py` file. Inside this file lives the `PlayBook` class. A good starting point for debugging playbook handling is around line 68 (for `ansible-core 2.11.1`), though this will vary depending upon the version you have installed. The following screenshot shows the adjacent code to help you locate the correct lines for your version:

```
jfreeman@mastery1: ~/examples/Chapter09/example08
File Edit View Search Terminal Help
else:
    self._basedir = os.path.normpath(os.path.join(self._basedir, os.path
.dirname(file_name)))

    # set the loaders basedir
    cur_basedir = self._loader.get_basedir()
    self._loader.set_basedir(self._basedir)

    add_all_plugin_dirs(self._basedir)

    self._file_name = file_name

    import pdb; pdb.set_trace()
try:
    ds = self._loader.load_from_file(os.path.basename(file_name))
except UnicodeDecodeError as e:
    raise AnsibleParserError("Could not read playbook (%s) due to encoding issues: %s" % (file_name, to_native(e)))

    # check for errors and restore the basedir in case this error is caught
and handled
    if ds is None:
        self._loader.set_basedir(cur_basedir)
:
```

Figure 9.14 – Adding the pdb debugger for debugging playbook loading and execution

Putting a breakpoint here will allow us to trace through finding the playbook file and parsing it. Specifically, by stepping into the `self._loader.load_from_file()` function call, we will be able to follow the parsing in action.

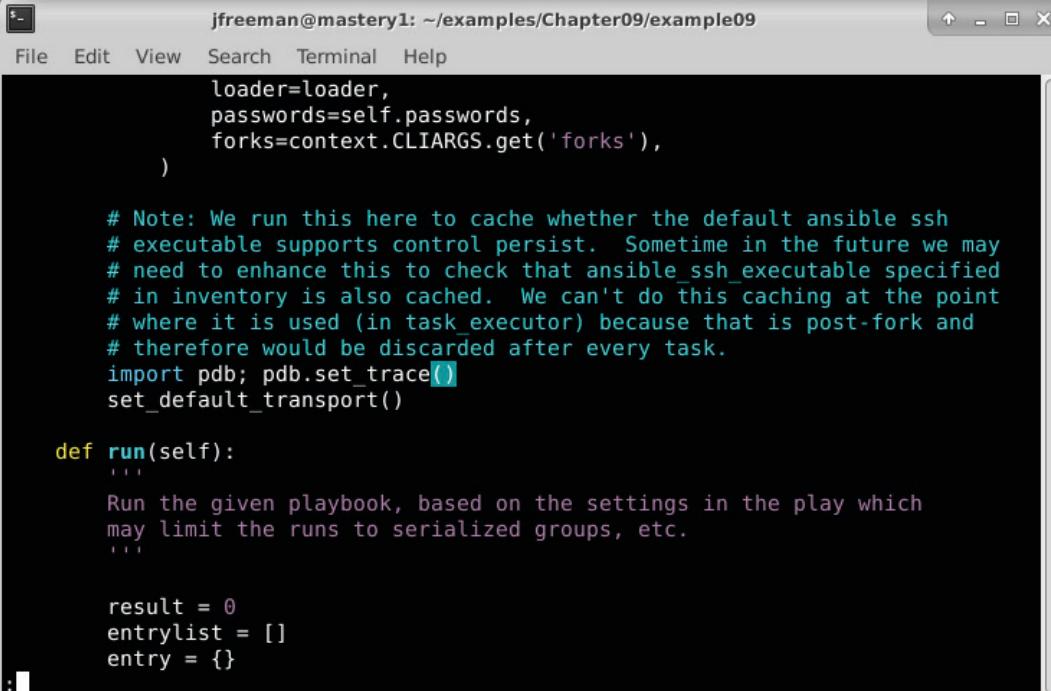
The `PlayBook` class `_load_playbook_data()` function just does the initial parsing. Other classes within other directories are used for the execution of plays and tasks. A particularly interesting directory is the `executor/` directory, which holds files with classes to execute playbooks, plays, and tasks. The `run()` function within the `PlaybookExecutor` class that's defined in the `executor/playbook_executor.py` file will loop through all of the plays in the playbook and execute the plays, which will, in turn, execute the individual tasks. This is the function to walk through if facing an issue related to play parsing, play or task callbacks, tags, play host selection, serial operation, handler running, or anything in between.

## Debugging executor code

Executor code in Ansible is the connector code that binds together inventory data, playbooks, plays, tasks, and connection methods. While each of those other code bits can be individually debugged, how they interact can be examined within executor code.

The executor classes are defined in various files within `executor/` and the `PlaybookExecutor` class. This class handles the execution of all of the plays and tasks within a given playbook. The `__init__()` class creation function creates a series of placeholder attributes as well as setting some default values, while the `run()` function is where most of the fun happens.

Debugging can often take you from one file to another, jumping around the code base. For example, in the `__init__()` function of the `PlaybookExecutor` class, there is code to cache whether or not the default **Secure Shell (SSH)** executable supports `ControlPersist`. You can find that by locating the `executor/playbook_executor.py` file within your Ansible installation path (just as we have done in the preceding sections) and looking for the line that states `set_default_transport()`. This is on line 76 in `ansible-core 2.11.1`, to give you an idea of where to look. Once you locate the appropriate place in the code, put a breakpoint here to allow you to follow the code, as shown in the following screenshot:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter09/example09". The window shows Python code for the `PlaybookExecutor` class. The code includes a `__init__` method with parameters `loader=loader`, `passwords=self.passwords`, and `forks=context.CLIARGS.get('forks')`. It also contains a note about caching the default ansible ssh executable supports control persist, and imports `pdb` to set a trace. The `run` method is defined with a docstring explaining it runs the given playbook based on play settings, which may limit runs to serialized groups. Finally, `result = 0`, `entrylist = []`, and `entry = {}` are initialized.

```
jfreeman@mastery1: ~/examples/Chapter09/example09
File Edit View Search Terminal Help
    loader=loader,
    passwords=self.passwords,
    forks=context.CLIARGS.get('forks'),
)

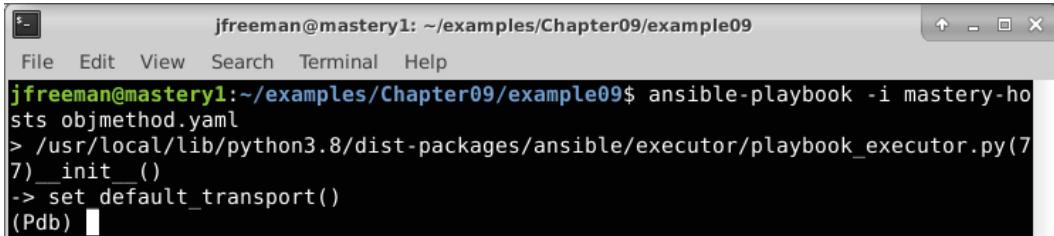
# Note: We run this here to cache whether the default ansible ssh
# executable supports control persist. Sometime in the future we may
# need to enhance this to check that ansible_ssh_executable specified
# in inventory is also cached. We can't do this caching at the point
# where it is used (in task_executor) because that is post-fork and
# therefore would be discarded after every task.
import pdb; pdb.set_trace()
set_default_transport()

def run(self):
    ...
    Run the given playbook, based on the settings in the play which
    may limit the runs to serialized groups, etc.
    ...

    result = 0
    entrylist = []
    entry = {}
```

Figure 9.15 – Inserting the Python debugger into the Ansible playbook executor code

We can now run our `objmethod.yml` playbook again to get into a debugging state, as illustrated in the following screenshot:



```
jfreeman@mastery1: ~/examples/Chapter09/example09
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter09/example09$ ansible-playbook -i mastery-hosts objmethod.yaml
> /usr/local/lib/python3.8/dist-packages/ansible/executor/playbook_executor.py(7)
7) __init__()
-> set_default_transport()
(Pdb) █
```

Figure 9.16 – Executing an example playbook to trigger the debugger

We'll need to step into the function to follow the execution. Stepping into the function will take us to a different file, as shown here:

```
(Pdb) step
--Call--
> /usr/local/lib/python3.8/dist-packages/ansible/utils/ssh_functions.py(55)set_d
efault_transport()
-> def set_default_transport():
(Pdb) █
```

Figure 9.17 – Stepping into the code to follow the execution

From here, we can use `list` to see the code in our new file, as illustrated in the following screenshot:

```
(Pdb) step
--Call--
> /usr/local/lib/python3.8/dist-packages/ansible/utils/ssh_functions.py(55)set_d
efault_transport()
-> def set_default_transport():
(Pdb) list
 50         _HAS_CONTROLPERSIST[ssh_executable] = has_cp
 51         return has_cp
 52
 53
 54     # TODO: move to 'smart' connection plugin that subclasses to ssh/paramiko
o as needed.
 55     -> def set_default_transport():
 56
 57         # deal with 'smart' connection .. one time ..
 58         if C.DEFAULT_TRANSPORT == 'smart':
 59             # TODO: check if we can deprecate this as ssh w/o control persis
t should
 60             # not be as common anymore.
```

Figure 9.18 – Listing the adjacent code to our current position in the debugger

Walking a few more lines down, we come to a block of code that will execute an `ssh` command and check the output to determine whether `ControlPersist` is supported, as illustrated in the following screenshot:

```
(Pdb) n
> /usr/local/lib/python3.8/dist-packages/ansible/utils/ssh_functions.py(41)check
_for_controlopersist()
-> has_cp = True
(Pdb) l
 36         return _HAS_CONTROLPERSIST[ssh_executable]
 37     except KeyError:
 38         pass
 39
 40     b_ssh_exec = to_bytes(ssh_executable, errors='surrogate_or_strict')
 41 ->     has_cp = True
 42     try:
 43         cmd = subprocess.Popen([b_ssh_exec, '-o', 'ControlPersist'], std
out=subprocess.PIPE, stderr=subprocess.PIPE)
 44         (out, err) = cmd.communicate()
 45         if b"Bad configuration option" in err or b"Usage:" in err:
 46             has_cp = False
```

Figure 9.19 – Locating the code to establish whether `ControlPersist` is supported

Let's walk through the next couple of lines and then print out what the value of `err` is. This will show us the result of the `ssh` execution and the whole string that Ansible will be searching within, as illustrated in the following screenshot:

```
jfreeman@mastery1: ~/examples/Chapter09/example09
File Edit View Search Terminal Help
out=subprocess.PIPE, stderr=subprocess.PIPE)
44         (out, err) = cmd.communicate()
45         if b"Bad configuration option" in err or b"Usage:" in err:
46             has_cp = False
(Pdb) n
> /usr/local/lib/python3.8/dist-packages/ansible/utils/ssh_functions.py(42)check
_for_controlopersist()
-> try:
(Pdb) n
> /usr/local/lib/python3.8/dist-packages/ansible/utils/ssh_functions.py(43)check
_for_controlopersist()
-> cmd = subprocess.Popen([b_ssh_exec, '-o', 'ControlPersist'], stdout=subproces
s.PIPE, stderr=subprocess.PIPE)
(Pdb) n
> /usr/local/lib/python3.8/dist-packages/ansible/utils/ssh_functions.py(44)check
_for_controlopersist()
-> (out, err) = cmd.communicate()
(Pdb) n
> /usr/local/lib/python3.8/dist-packages/ansible/utils/ssh_functions.py(45)check
_for_controlopersist()
-> if b"Bad configuration option" in err or b"Usage:" in err:
(Pdb) p err
b'command-line line 0: Missing ControlPersist argument.\r\n'
```

Figure 9.20 – Analyzing the SSH connection results using the pdb debugger

As we can see, the search string is not within the `err` variable, so the value of `has_cp` remains as the default of `True`.

#### A Quick Note on Forks and Debugging

When Ansible uses multiprocessing for multiple forks, debugging becomes difficult. A debugger may be attached to one fork and not another, which will make it very difficult to debug the code. Unless specifically debugging the multiprocessing code, it's a best practice to stick to a single fork.

## Debugging remote code

Remote code is code that Ansible transports to a remote host to execute it. This is typically module code, or in the case of action plugins, other snippets of code. Using the debugging method we discussed in the previous section to debug module execution will not work, as Ansible simply copies the code over and then executes it. There is no terminal attached to remote code execution, and thus there is no way to attach it to a debugging prompt—that is, without editing the module code.

To debug module code, we need to edit the module code itself to insert a debugger breakpoint. Instead of directly editing the installed module file, create a copy of the file in a `library/` directory relative to the playbooks. This copy of the module code will be used instead of the installed file, which makes it easy to temporarily edit a module without disrupting other users of modules on the system.

Unlike other Ansible code, module code cannot be directly debugged with `pdb` because the module code is assembled and then transported to a remote host. Thankfully, there is a solution in the form of a slightly different debugger named `rpdb`—the remote Python debugger. This debugger has the ability to start a listening service on a provided port to allow remote connections into the Python process. Connecting to the process remotely will allow the code to be debugged line by line, just as we did with other Ansible code.

To demonstrate how this debugger works, we're first going to need a remote host. For this example, we're using a remote host by the name of `debug.example.com` (though feel free to use your own example with the appropriate adjustments to hostnames). Next, we need a playbook to execute a module that we'd like to debug. The code is illustrated in the following snippet:

```
---  
- name: remote code debug  
  hosts: debug.example.com  
  gather_facts: false
```

```
become: true

tasks:
  - name: a remote module execution
    systemd:
      name: nginx
      state: stopped
      enabled: no
```

#### Important Note

The eagle-eyed among you will have noticed that, for the first time in this book, we have not used the **fully qualified class name (FQCN)** for the module reference. This is because the FQCN tells Ansible to use its own built-in module from the location it expects, where we actually want to load the local copy we will place in our local `library/` directory. As a result, we must use just the short-form name of the module in this one instance.

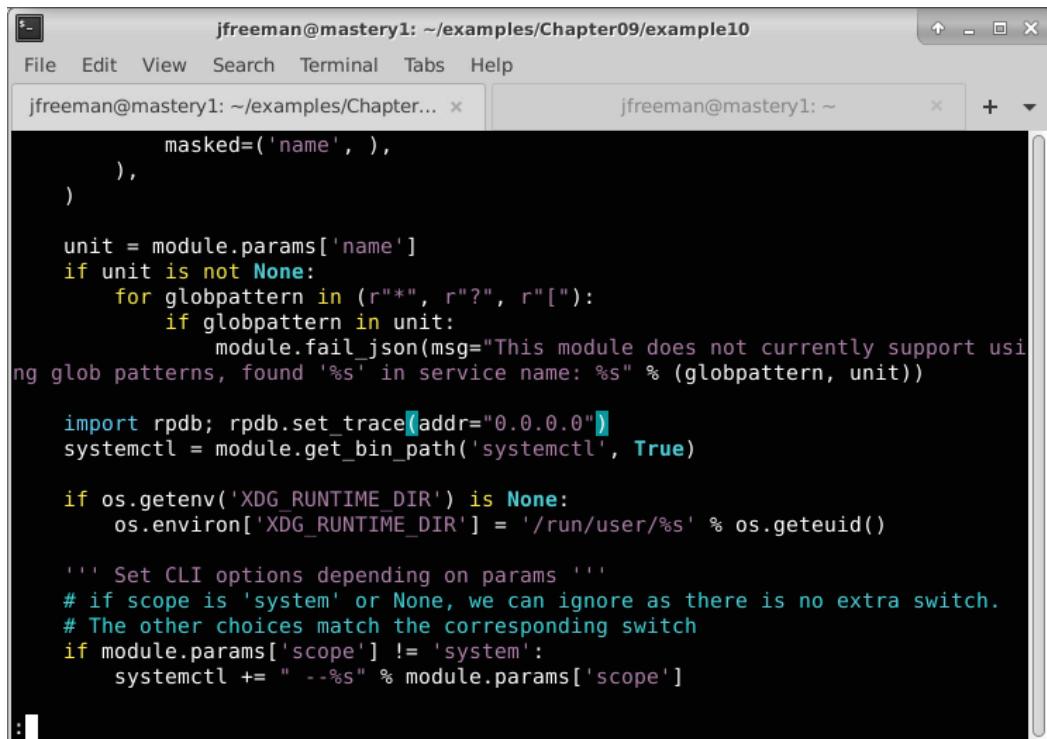
We will also need a new inventory file to reference our new test host. As I don't have the **Domain Name System (DNS)** entry set up for this host, I'm using the special `ansible_host` variable in the inventory to tell Ansible which **Internet Protocol (IP)** address to connect to `debug.example.com` on, as illustrated in the following code snippet:

```
debug.example.com ansible_host=192.168.81.154
```

#### Important Note

Don't forget to set up SSH authentication between your two hosts—I'm using an SSH key so that I don't need to type in a password every time I run `ansible-playbook`.

This play simply calls the `ansible.builtin.systemd` module to ensure that the `nginx` service is stopped and will not start up on boot. As we stated previously, we need to make a copy of the service module and place it in `library/`. The location of the service module to copy from will vary based on the way Ansible is installed. On my demo system for this book, it is located in `/usr/local/lib/python3.8/dist-packages/ansible/modules/systemd.py`. Then, we can edit it to put in our breakpoint. I am inserting this at line 358 on my system—this is correct for `ansible-core 2.11.1` but may change as newer versions are released. However, the following screenshot should give you an idea of where to insert the code:



```
jfreeman@mastery1: ~/examples/Chapter09/example10
File Edit View Search Terminal Tabs Help
jfreeman@mastery1: ~/examples/Chapter... x jfreeman@mastery1: ~ + -
    masked=('name', ),
)
)

unit = module.params['name']
if unit is not None:
    for globpattern in (r"*, r"?", r"["):
        if globpattern in unit:
            module.fail_json(msg="This module does not currently support using glob patterns, found '%s' in service name: %s" % (globpattern, unit))

import rpdb; rpdb.set_trace(addr="0.0.0.0")
systemctl = module.get_bin_path('systemctl', True)

if os.getenv('XDG_RUNTIME_DIR') is None:
    os.environ['XDG_RUNTIME_DIR'] = '/run/user/%s' % os.geteuid()

''' Set CLI options depending on params '''
# if scope is 'system' or None, we can ignore as there is no extra switch.
# The other choices match the corresponding switch
if module.params['scope'] != 'system':
    systemctl += " --%s" % module.params['scope']

:
```

Figure 9.21 – Inserting the remote Python debugger into Ansible module code

We'll put the breakpoint just before the `systemctl` variable value gets created. First, the `rpdb` module must be imported (meaning that the `rpdb` Python library needs to exist on the remote host), and then the breakpoint needs to be created with `set_trace()`.

#### Important Note

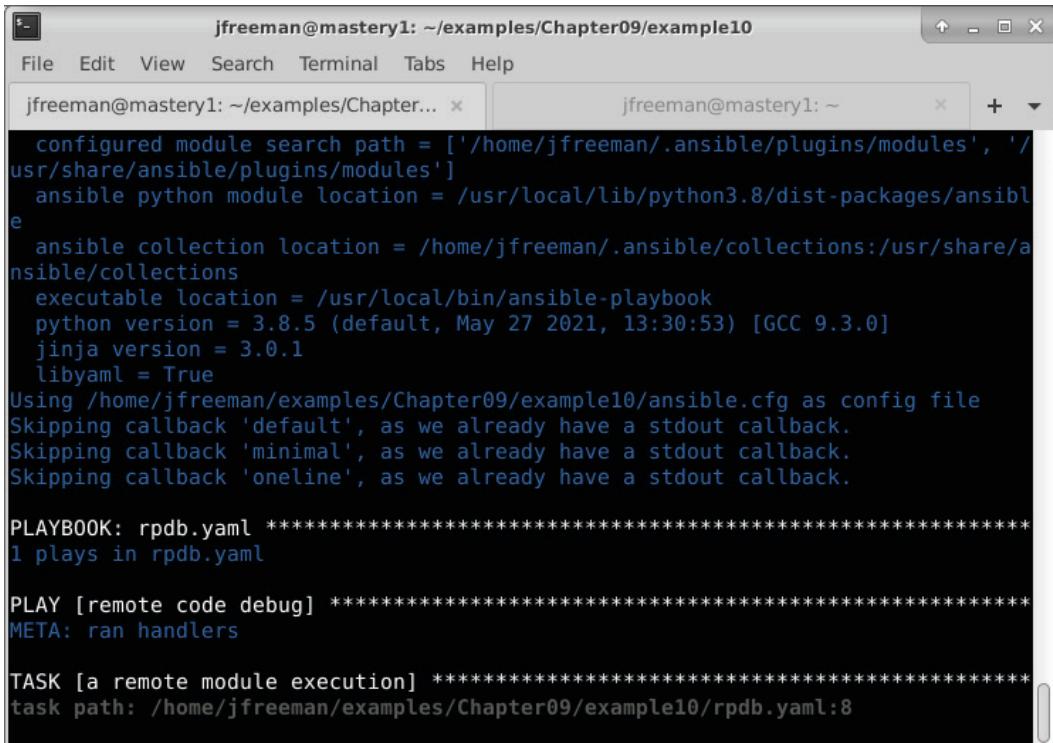
On Ubuntu Server 20.04 (as with the host that was used in the demo), `rpdb` can be installed with `pip` using the following command: `sudo pip3 install rpdb`.

Unlike the regular debugger, this function will open a port and listen for external connections. By default, the function will listen for connections to port 4444 on the address `127.0.0.1`. However, that address is not exposed over the network, so in my example, I've instructed `rpdb` to listen on address `0.0.0.0`, which is effectively every address on the host (though as I'm sure you'll understand, this has security implications you need to be careful of!).

**Important Note**

If the host on which you are running `rpdb` has a firewall (for example, `firewalld` or `ufw`), you will need to open port 4444 for the example given here to work.

We can now run this playbook to set up the server that will wait for a client connection, as follows:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter09/example10". The terminal displays the execution of an Ansible playbook named "rpdb.yaml". The output includes configuration details like module search paths, Python version (3.8.5), and Jinja version (3.0.1). It also shows the configuration file being used and callbacks being skipped. The final part of the output shows the task definition for remote module execution.

```
jfreeman@mastery1: ~/examples/Chapter09/example10
File Edit View Search Terminal Tabs Help
jfreeman@mastery1: ~/examples/Chapter... x jfreeman@mastery1: ~ x + -
configured module search path = ['/home/jfreeman/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
ansible python module location = /usr/local/lib/python3.8/dist-packages/ansible
ansible collection location = /home/jfreeman/.ansible/collections:/usr/share/ansible/collections
executable location = /usr/local/bin/ansible-playbook
python version = 3.8.5 (default, May 27 2021, 13:30:53) [GCC 9.3.0]
jinja version = 3.0.1
libyaml = True
Using /home/jfreeman/examples/Chapter09/example10/ansible.cfg as config file
Skipping callback 'default', as we already have a stdout callback.
Skipping callback 'minimal', as we already have a stdout callback.
Skipping callback 'oneline', as we already have a stdout callback.

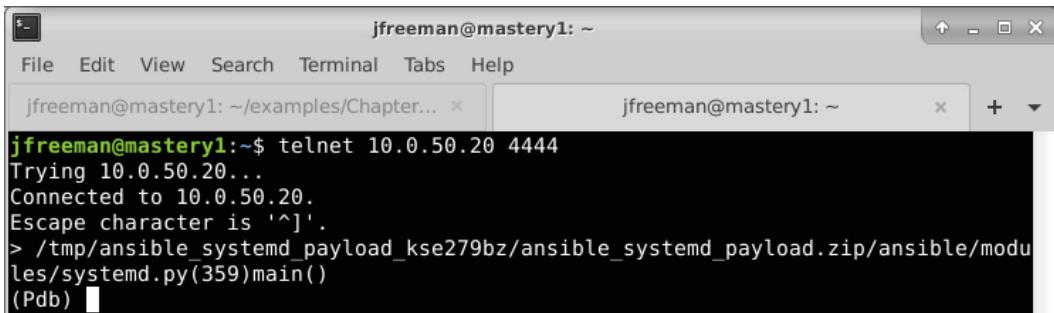
PLAYBOOK: rpdb.yaml ****
1 plays in rpdb.yaml

PLAY [remote code debug] ****
META: ran handlers

TASK [a remote module execution] ****
task path: /home/jfreeman/examples/Chapter09/example10/rpdb.yaml:8
```

Figure 9.22 – Running a test playbook for remote module debugging

Now that the server is running, we can connect to it from another terminal. Connecting to the running process can be accomplished with the `telnet` program, as illustrated in the following screenshot:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~". It has two tabs open. The left tab shows the command `jfreeman@mastery1:~/examples/Chapter... $ telnet 10.0.50.20 4444`. The right tab shows the output of the telnet session, which includes "Trying 10.0.50.20...", "Connected to 10.0.50.20.", "Escape character is '^]'.", and a stack trace starting with "`> /tmp/ansible_systemd_payload_kse279bz/ansible_systemd_payload.zip/ansible/modules/systemd.py(359)main()`". Below the stack trace, "(Pdb)" is followed by a blank line.

Figure 9.23 – Using telnet to connect to a remote Python debugger session for module debugging

From this point on, we can debug as normal. The commands we used before still exist, such as `list` to show where in the code the current frame is, as illustrated in the following screenshot:

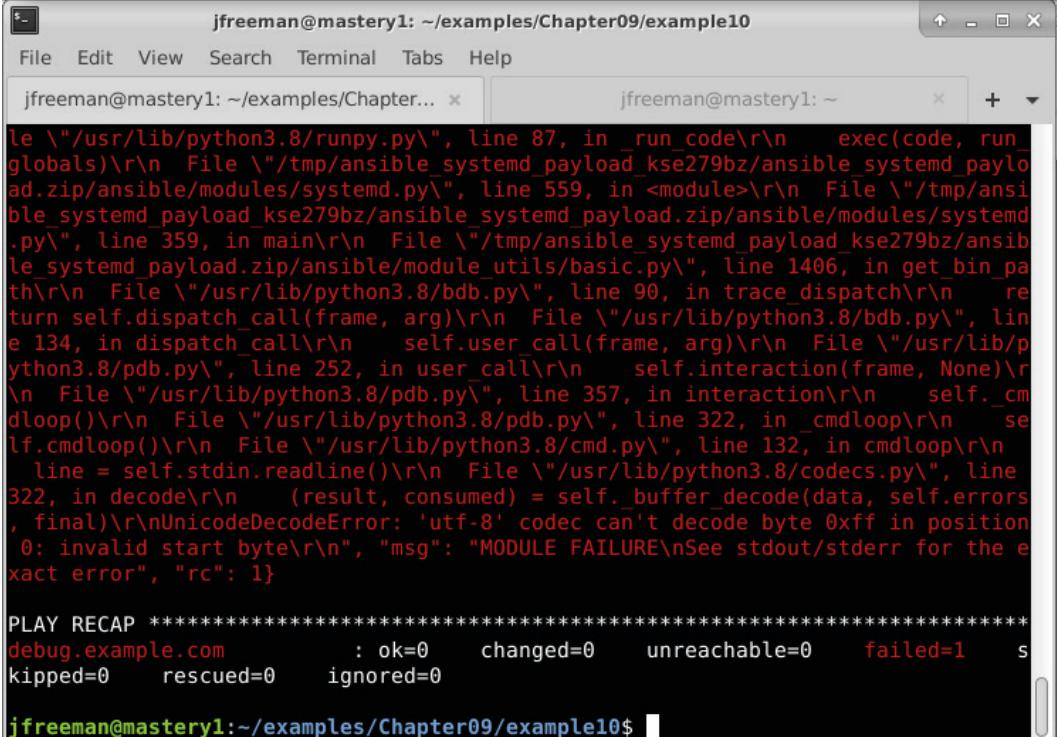


The screenshot shows a terminal window with a Python debugger session. The command `(Pdb)` is followed by several lines of code from the `get_bin_path` function in `basic.py`. The code includes annotations like "pass", "# we won't error here, as it may \*not\* be a problem, # and we don't want to break modules unnecessarily", and "return None". The line numbers range from 1401 to 1411. The command `(Pdb) list` is shown, along with the start of the `def get_bin_path` definition and its docstrings. The command `(Pdb)` is followed by a blank line.

Figure 9.24 – Using the now-familiar Python debugger commands in a remote debugging session

Using the debugger, we can walk through the `systemd` module to track how it determines the path to the underlying tool, trace which commands are executed on the host, determine how a change is computed, and so on. The entire file can be stepped through, including any other external libraries the module may make use of, allowing the debugging of other non-module code on the remote host as well.

If the debugging session allows the module to exit cleanly, the playbook's execution will return to normal. However, if the debugging session is disconnected before the module completes, the playbook will produce an error, as shown in the following screenshot:



```
jfreeman@mastery1: ~/examples/Chapter09/example10
File Edit View Search Terminal Tabs Help
jfreeman@mastery1: ~/examples/Chapter... × jfreeman@mastery1: ~ × + ▾
le "/usr/lib/python3.8/runpy.py", line 87, in _run_code\r\n    exec(code, run_globals)\r\n  File "/tmp/ansible_systemd_payload_kse279bz/ansible_systemd_payload.zip/ansible/modules/systemd.py", line 559, in <module>\r\n  File "/tmp/ansible_systemd_payload_kse279bz/ansible_systemd_payload.zip/ansible/modules/systemd.py", line 359, in main\r\n  File "/tmp/ansible_systemd_payload_kse279bz/ansible_systemd_payload.zip/ansible/module_utils/basic.py", line 1406, in get_bin_path\r\n  File "/usr/lib/python3.8/bdb.py", line 90, in trace_dispatch\r\n      return self.dispatch_call(frame, arg)\r\n  File "/usr/lib/python3.8/bdb.py", line 134, in dispatch_call\r\n      self.user_call(frame, arg)\r\n  File "/usr/lib/python3.8/pdb.py", line 252, in user_call\r\n      self.interaction(frame, None)\r\n  File "/usr/lib/python3.8/pdb.py", line 357, in interaction\r\n      self._cmdloop()\r\n  File "/usr/lib/python3.8/pdb.py", line 322, in _cmdloop\r\n      self.cmdloop()\r\n  File "/usr/lib/python3.8/cmd.py", line 132, in cmdloop\r\n      line = self.stdin.readline()\r\n  File "/usr/lib/python3.8/codecs.py", line 322, in decode\r\n      (result, consumed) = self._buffer_decode(data, self.errors, final)\r\nUnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 0: invalid start byte\r\n", "msg": "MODULE FAILURE\nSee stdout/stderr for the exact error", "rc": 1}

PLAY RECAP ****
debug.example.com : ok=0    changed=0    unreachable=0    failed=1    s
kipped=0    rescued=0    ignored=0

jfreeman@mastery1:~/examples/Chapter09/example10$
```

Figure 9.25 – An example of an error produced when terminating the remote debugging session early  
Because of this side effect, it is best to not exit the debugger early, and instead issue a continue command when your debugging is finished.

## Debugging action plugins

Some modules are actually action plugins. These are tasks that will execute some code locally before transporting code to the remote host. Some example action plugins include `copy`, `fetch`, `script`, and `template`. The source to these plugins can be found in `plugins/action/`. Each plugin will have its own file in this directory that can be edited to have breakpoints inserted in order to debug the code that's executed, prior to (or in lieu of) sending code to the remote host. Debugging these is typically done with `pdb` since most of the code is executed locally.

## Summary

Ansible is a piece of software, and software breaks; it's not a matter of if, but when. Invalid input, improper assumptions, and unexpected environments are all things that can lead to a frustrating situation when tasks and plays are not performing as expected. Introspection and debugging are troubleshooting techniques that can quickly turn frustration into elation when a root cause is discovered.

In this chapter, we learned about how to get Ansible to log its actions to a file, and how to change the verbosity level of Ansible's output. We then learned how to inspect variables to ensure their values are in line with your expectations before we moved on to debugging Ansible code in detail. Furthermore, we walked through the process of inserting breakpoints into core Ansible code and executed both local and remote Python debugging sessions using standard Python tools.

In the next chapter, we will learn how to extend the functionality of Ansible by writing our own modules, plugins, and inventory sources.

## Questions

1. What level of verbosity would you need to launch Ansible with to see details such as connection attempts?
  - a) Level 3 or above
  - b) Level 2 or above
  - c) Level 1 or above
  - d) Level 4
2. Why should you be careful with verbosity levels above level one if you are using sensitive data in your playbook?
  - a) Higher verbosity levels don't support the use of vaults.
  - b) Higher verbosity levels may log sensitive data to the console and/or log file.
  - c) Higher verbosity levels will print SSH passwords.
3. Ansible can be centrally configured to log its output to a file by:
  - a) Using the `ANSIBLE_LOG_PATH` environment variable
  - b) Using the `log_path` directive in `ansible.cfg`
  - c) Redirecting the output of each playbook run to a file
  - d) All of these

4. The name of the module used for variable introspection is:
  - a) `ansible.builtin.analyze`
  - b) `ansible.builtin.introspect`
  - c) `ansible.builtin.debug`
  - d) `ansible.builtin.print`
5. When referencing subelements in Ansible variables, which syntax is the safest to prevent clashes with reserved Python names?
  - a) Dot notation
  - b) Standard subscript syntax
  - c) Ansible subelement notation
  - d) Standard dot notation
6. Unless you need to perform low-level code debugging, you can debug the flow of a playbook using:
  - a) The debug strategy
  - b) Debug execution
  - c) Debug task planner
  - d) None of these
7. The name of the Python local debugger as demonstrated in this book is:
  - a) PyDebug
  - b) `python-debug`
  - c) `pdb`
  - d) `pdebug`
8. You can also debug the execution of modules on remote hosts:
  - a) Using the Python `rpdb` module.
  - b) By copying the playbook to the host and using `pdb`.
  - c) Via a packet tracer such as `tcpdump`.
  - d) This is not possible.

9. Unless configured otherwise, the remote Python debugger listens for connections on:
  - a) 127.0.0.1:4433
  - b) 0.0.0.0:4444
  - c) 127.0.0.1:4444
  - d) 0.0.0.0:4433
10. Why should you not end your remote Python debugging session without letting the code run to completion?
  - a) It results in an error in your playbook run.
  - b) It will result in loss of files.
  - c) It might corrupt your Ansible installation.
  - d) It will result in a hung debug session.



# 10

# Extending Ansible

It must be said that **Ansible** takes the *kitchen sink* approach to functionality and tries to provide, out of the box, every piece of functionality you might ever need. With the `ansible-core` package and its associated collections, there are almost 6,000 modules available for use within Ansible at the time of writing – compare that to the (roughly) 800 that were included when the second edition of this book was published! In addition to these, there is a rich plugin and filter architecture with numerous callback plugins, lookup plugins, filter plugins, and dynamic inventory plugins included. Now, collections provide a whole new vector through which new functionality can be provided.

Despite this, there will always be cases where Ansible doesn't quite perform the tasks required, especially in large and complex environments, or ones where bespoke in-house systems have been developed. Luckily, the design of Ansible, coupled with its open source nature, makes it easy for anyone to extend it by developing features. The advent of collections with Ansible 3.0 has meant it is easier than ever to extend functionality. However, in this chapter, we will focus on the specifics of contributing to the `ansible-core` package. If you wish to contribute by creating a collection, you can easily follow the steps provided in this chapter to develop the code you require (for example, creating a new module) and then package it as a collection, as we described in *Chapter 2, Migrating from Earlier Ansible Versions*. How you contribute is up to you and your target audience – if you feel your code will help everyone who uses Ansible, then you may wish to submit it to `ansible-core`; otherwise, it is probably best built into a collection.

This chapter will explore the following ways in which new capabilities can be added to Ansible:

- Developing modules
- Developing plugins
- Developing dynamic inventory plugins
- Contributing code to the Ansible project

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running **Ansible 4.3** or newer. Almost any flavor of Linux should do – for those interested in the specifics, all the code presented in this chapter was tested on **Ubuntu Server 20.04 LTS** unless stated otherwise and on Ansible 4.3. The example code that accompanies this chapter can be downloaded from GitHub at <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter10>.

Check out the following video to see the code in action: <https://bit.ly/3DTKL35>.

## Developing modules

Modules are the workhorse of Ansible. They provide just enough abstraction so that playbooks can be stated simply and clearly. There are over 100 modules and plugins maintained by the core Ansible development team and they are distributed as part of the `ansible-core` package, covering commands, files, package management, source control, system, utilities, and so on. In addition, there are nearly 6,000 other modules maintained by community contributors that expand functionality in many of these categories and many others, such as public cloud providers, databases, networking, and so on, through collections. The real magic happens inside the module's code, which takes in the arguments that are passed to it and works to establish the desired outcome.

Modules in Ansible are the pieces of code that get transported to the remote host to be executed. They can be written in any language that the remote host can execute; however, Ansible provides some very useful shortcuts for writing modules in Python, and you will find that most are indeed written in Python.

## The basic module construct

A module exists to satisfy a need – the need to do a piece of work on a host. Modules usually, but not always, expect input, and will return some sort of output. Modules also strive to be idempotent, allowing the module to be run over and over again without it having a negative impact. In Ansible, the input is in the form of command-line arguments to the module, and the output is delivered as JSON to STDOUT.

Input is generally provided in the space-separated key=value syntax, and it's up to the module to deconstruct these into usable data. If you're using Python, there are convenience functions to manage this, and if you're using a different language, then it is up to your module code to fully process the input.

The output is JSON formatted. Convention dictates that in a successful scenario, the JSON output should have at least one key, changed, which is a Boolean, to indicate whether the module execution resulted in a change. Additional data can be returned as well, which may be useful for defining what changed or to provide important information back to the playbook for later use. Additionally, host facts can be returned in the JSON data to automatically create host variables based on the module execution results. We will look at this in more detail later, in the *Providing fact data* section.

## Custom modules

Ansible provides an easy mechanism to utilize custom modules other than those that come with Ansible. As we learned in *Chapter 1, The System Architecture and Design of Ansible*, Ansible will search many locations to find a requested module. One such location, and indeed the first location, is the `library/` subdirectory of the path where the top-level playbook resides. This is where we will place our custom module so that we can use it in our example playbook, as our focus is on developing for the `ansible-core` package. However, as we have already stated, you can also distribute modules via collections, and *Chapter 2, Migrating from Earlier Ansible Versions*, described (with a practical example taken from this chapter) how to package up modules for distribution via a collection.

In addition to this, modules can also be embedded within roles to deliver the added functionality that a role may depend upon. These modules are only available to the role that contains the module, or any other roles or tasks that are executed after the role containing the module. To deliver a module with a role, place the module in the `library/` subdirectory of the role's root. While this is still a viable route, it is expected that as Ansible releases of 3.0 and later become commonplace, you will distribute your modules via collections. A period of overlap is being provided to support the many Ansible 2.9 and earlier distributions that exist.

## Example – simple module

To demonstrate the ease of writing Python-based modules, let's create a simple module. The purpose of this module will be to remotely copy a source file to a destination file, a simple task that we can build up from. To start our module, we need to create the module file. For easy access to our new module, we'll create the file in the `library`/ subdirectory of the working directory we've already been using. We'll call this module `remote_copy.py`, and to start it off, we'll need to put in a shebang line to indicate that this module is to be executed with Python:

```
#!/usr/bin/python  
#
```

For Python-based modules, the convention is to use `/usr/bin/python` as the listed executable. When executed on a remote system, the configured Python interpreter for the remote host is used to execute the module, so fret not if your Python code doesn't exist in this path. Next, we'll import a Python library we'll use later in the module, called `shutil`:

```
import shutil
```

Now, we're ready to create our `main` function. The `main` function is essentially the entry point to the module, where the arguments to the module will be defined and where the execution will start. When creating modules in Python, we can take some shortcuts in this `main` function to bypass a lot of boilerplate code and get straight to the argument definitions.

We can do this by creating an `AnsibleModule` object and giving it an `argument_spec` dictionary for the arguments:

```
def main():  
    module = AnsibleModule(  
        argument_spec = dict(  
            source=dict(required=True, type='str'),  
            dest=dict(required=True, type='str')  
        )  
    )
```

In our module, we're providing two arguments. The first argument is `source`, which we'll use to define the source file for the copy. The second argument is `dest`, which is the destination for the copy. Both of these arguments are marked as required, which will raise an error when executed if one of the two is not provided. Both arguments are of the `string` type. The location of the `AnsibleModule` class has not been defined yet as that happens later in the file.

With a module object at our disposal, we can now create the code that will do the actual work on the remote host. We'll make use of `shutil.copy` and our provided arguments to accomplish this:

```
shutil.copy(module.params['source'],
            module.params['dest'])
```

The `shutil.copy` function expects a source and a destination, which we've provided by accessing `module.params`. The `module.params` dictionary holds all of the parameters for the module. Having completed the copy, we are now ready to return the results to Ansible. This is done via another `AnsibleModule` method, `exit_json`. This method expects a set of `key=value` arguments and will format it appropriately for a JSON return. Since we're always performing a copy, we will always return a change for simplicity's sake:

```
module.exit_json(changed=True)
```

This line will exit the function, and thus the module. This function assumes a successful action and will exit the module with the appropriate return code for success: 0. We're not done with our module's code, though; we still have to account for the `AnsibleModule` location. This is where a bit of magic happens, where we tell Ansible what other code to combine with our module to create a complete work that can be transported:

```
from ansible.module_utils.basic import *
```

That's all it takes! That one line gets us access to all of the basic `module_utils`, a decent set of helper functions and classes. There is one last thing we should put into our module: a couple of lines of code telling the interpreter to execute the `main()` function when the module file is executed:

```
if __name__ == '__main__':
    main()
```

Now, our module file is complete, which means we can test it with a playbook. We'll call our playbook `simple_module.yaml` and store it in the same directory as the `library/` directory, where we've just written our module file. We'll run the play on `localhost` for simplicity's sake and use a couple of filenames in `/tmp` for the source and destination. We'll also use a task to ensure that we have a source file to begin with:

```
---
- name: test remote_copy module
  hosts: localhost
  gather_facts: false

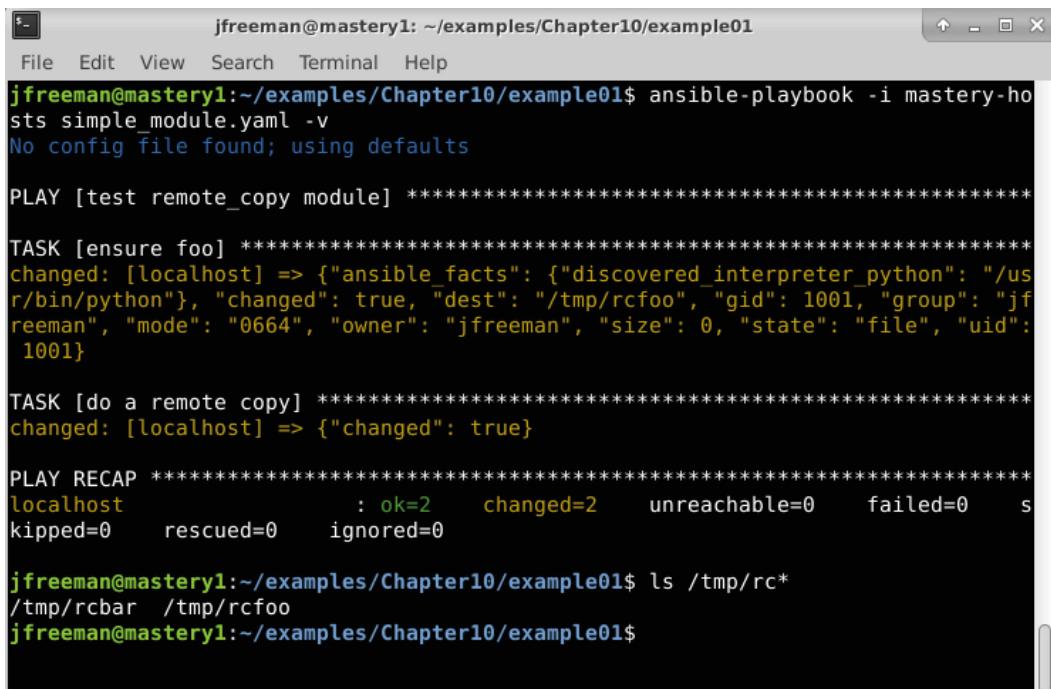
  tasks:
    - name: ensure foo
      ansible.builtin.file:
        path: /tmp/rcfoo
        state: touch

    - name: do a remote copy
      remote_copy:
        source: /tmp/rcfoo
        dest: /tmp/rcbar
```

As our new module is being run from a `library/` directory local to the playbook, it does not have a **fully qualified collection name (FQCN)**, so we will reference it by its short name only in the playbook. To run this playbook, we'll run the following command:

```
ansible-playbook -i mastery-hosts simple_module.yaml -v
```

If the `remote_copy` module file is written to the correct location, everything will work just fine, and the screen output will look as follows:



```
jfreeman@mastery1: ~/examples/Chapter10/example01
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter10/example01$ ansible-playbook -i mastery-hosts simple_module.yaml -v
No config file found; using defaults

PLAY [test remote_copy module] ****
TASK [ensure foo] ****
changed: [localhost] => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "dest": "/tmp/rcfoo", "gid": 1001, "group": "jfreeman", "mode": "0664", "owner": "jfreeman", "size": 0, "state": "file", "uid": 1001}

TASK [do a remote copy] ****
changed: [localhost] => {"changed": true}

PLAY RECAP ****
localhost : ok=2    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

jfreeman@mastery1:~/examples/Chapter10/example01$ ls /tmp/rc*
/tmp/rcbar /tmp/rcfoo
jfreeman@mastery1:~/examples/Chapter10/example01$
```

Figure 10.1 – Running a simple playbook to test our first custom Ansible module

Our first task touches on the `/tmp/rcfoo` path to ensure that it exists, and then our second task makes use of `remote_copy` to copy `/tmp/rcfoo` to `/tmp/rcbar`. Both tasks are successful, resulting in a `changed` status each time.

## Documenting a module

No module should be considered complete unless it contains documentation regarding how to operate it. Documentation for a module exists within the module itself, in special variables called `DOCUMENTATION`, `EXAMPLES`, and `RETURN`.

The `DOCUMENTATION` variable contains a specially formatted string describing the module's name, the version of either `ansible-core` or the parent collection that it was added to, a short description of the module, a longer description, a description of the module arguments, the author and license information, additional requirements, and any extra notes that are useful to users of the module. Let's add a `DOCUMENTATION` string to our module under the existing `import shutil` statement:

```
import shutil

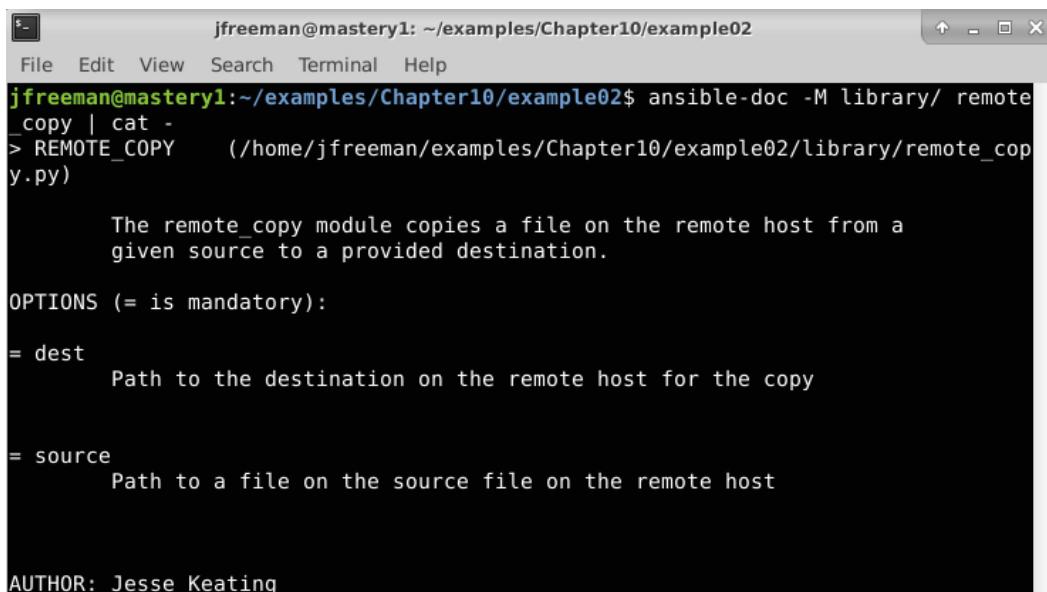
DOCUMENTATION = '''
```

```
---
module: remote_copy
version_added: future
short_description: Copy a file on the remote host
description:
  - The remote_copy module copies a file on the remote host
    from a given source to a provided destination.
options:
  source:
    description:
      - Path to a file on the source file on the remote host
    required: True
  dest:
    description:
      - Path to the destination on the remote host for the copy
    required: True
  author:
    - Jesse Keating
'''
```

The format of the string is essentially YAML, with some top-level keys containing hash structures within it (the same as the `options` key). Each option has sub-elements to describe the option, indicate whether the option is required, list any aliases for the option, list static choices for the option, or indicate a default value for the option. With this string saved to the module, we can test our formatting to ensure that the documentation will render correctly. This is done via the `ansible-doc` tool, with an argument to indicate where to search for the modules. If we run it from the same place as our playbook, the command will be as follows:

```
ansible-doc -M library/ remote_copy
```

The output should look as follows:



```
jfreeman@mastery1: ~/examples/Chapter10/example02
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter10/example02$ ansible-doc -M library/ remote_copy | cat -
> REMOTE_COPY      (/home/jfreeman/examples/Chapter10/example02/library/remote_copy.py)
The remote_copy module copies a file on the remote host from a given source to a provided destination.

OPTIONS (= is mandatory):

= dest
    Path to the destination on the remote host for the copy

= source
    Path to a file on the source file on the remote host

AUTHOR: Jesse Keating
```

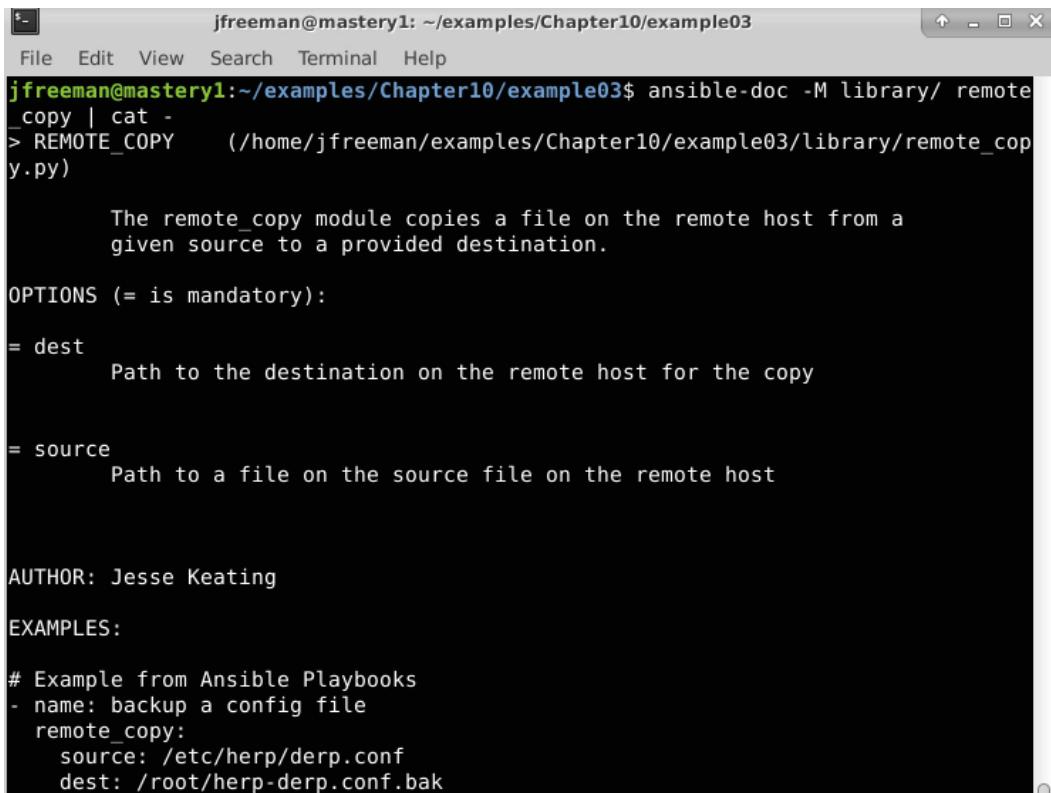
Figure 10.2 – Using the ansible-doc tool to view our new module's documentation

In this example, I've piped the output into `cat` to prevent the pager from hiding the execution line. Our documentation string appears to be formatted correctly and provides the user with important information regarding the usage of the module.

The `EXAMPLES` string is used to provide one or more example uses of the module, snippets of the task code that you would use in a playbook. Let's add an example task to demonstrate its usage. This variable definition traditionally goes after the `DOCUMENTATION` definition:

```
EXAMPLES = '''
# Example from Ansible Playbooks
- name: backup a config file
  remote_copy:
    source: /etc/herp/derp.conf
    dest: /root/herp-derp.conf.bak
'''
```

With this variable defined, our `ansible-doc` output will now include the example, as shown here:



```
jfreeman@mastery1: ~/examples/Chapter10/example03
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter10/example03$ ansible-doc -M library/ remote_copy | cat -
> REMOTE_COPY      (/home/jfreeman/examples/Chapter10/example03/library/remote_copy.py)

    The remote_copy module copies a file on the remote host from a
    given source to a provided destination.

OPTIONS (= is mandatory):

= dest
    Path to the destination on the remote host for the copy

= source
    Path to a file on the source file on the remote host

AUTHOR: Jesse Keating

EXAMPLES:

# Example from Ansible Playbooks
- name: backup a config file
  remote_copy:
    source: /etc/herp/derp.conf
    dest: /root/herp-derp.conf.bak
```

Figure 10.3 – Expanding our module documentation with an EXAMPLES section

The last documentation variable, `RETURN`, is used to describe the return data from a module's execution. Return data is often useful as a registered variable for later usage and having documentation of what return data to expect can aid playbook development. Our module doesn't have any return data yet; so, before we can document any, we have to add return data. This can be done by modifying the `module.exit_json` line to add more information. Let's add the `source` and `dest` data to the return output:

```
    module.exit_json(changed=True, source=module.
params['source'],
                  dest=module.params['dest'])
```

Rerunning the playbook will show extra data being returned, as shown in the following screenshot:

```
jfreeman@mastery1: ~/examples/Chapter10/example04
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter10/example04$ ansible-playbook -i mastery-hosts simple_module.yaml -v
No config file found; using defaults

PLAY [test remote_copy module] *****

TASK [ensure foo] *****
changed: [localhost] => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "dest": "/tmp/rcfoo", "gid": 1001, "group": "jfreeman", "mode": "0664", "owner": "jfreeman", "size": 0, "state": "file", "uid": 1001}

TASK [do a remote copy] *****
changed: [localhost] => {"changed": true, "dest": "/tmp/rcbar", "gid": 1001, "group": "jfreeman", "mode": "0664", "owner": "jfreeman", "size": 0, "source": "/tmp/rcfoo", "state": "file", "uid": 1001}

PLAY RECAP *****
localhost : ok=2    changed=2    unreachable=0    failed=0    skipped=0   rescued=0   ignored=0
```

Figure 10.4 – Running our expanded module with return data added

Looking closely at the return data, we can see more data than we put in our module. This is a bit of helper functionality within Ansible; when a return dataset includes a `dest` variable, Ansible will gather more information about the destination file. The extra data that's gathered is `gid` (group ID), `group` (group name), `mode` (permissions), `uid` (owner ID), `owner` (owner name), `size`, and `state` (file, link, or directory). We can document all of these return items in our `RETURN` variable, which is added after the `EXAMPLES` variable. Everything between the two sets of three single quotes (' '' ') is returned – thus, this first part returns the file paths and ownership:

```
RETURN = '''
source:
  description: source file used for the copy
  returned: success
  type: string
  sample: "/path/to/file.name"

dest:
  description: destination of the copy
  returned: success
```

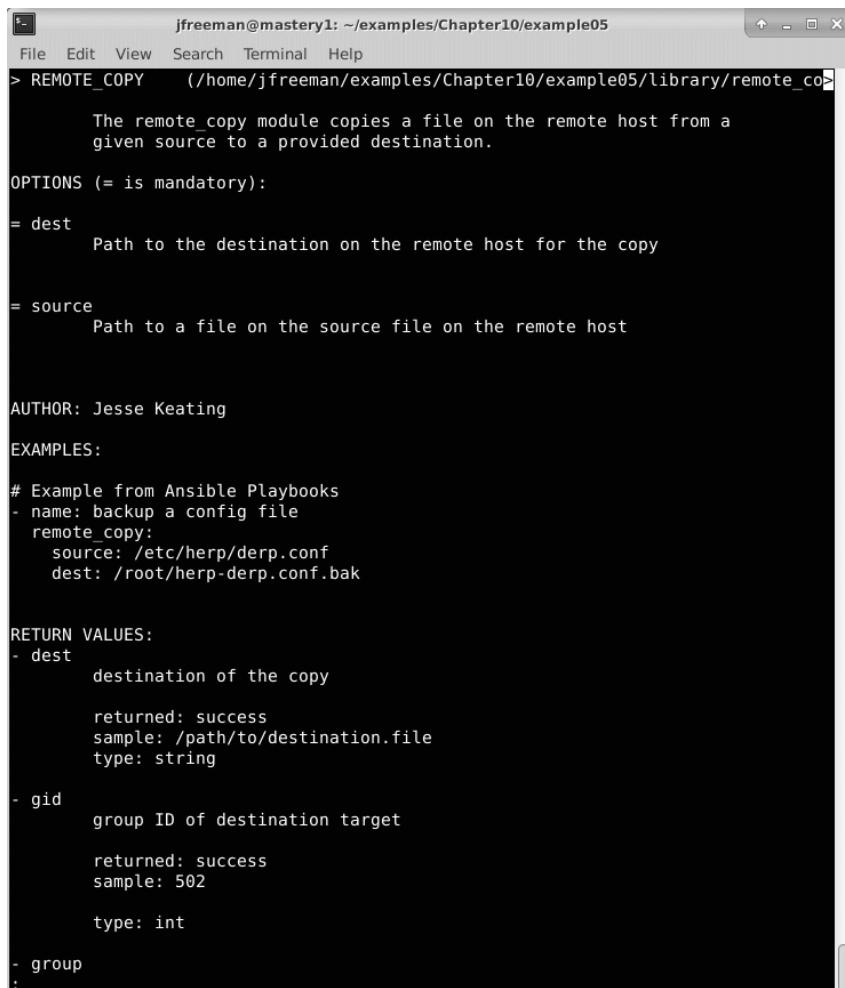
```
type: string
sample: "/path/to/destination.file"
gid:
  description: group ID of destination target
  returned: success
  type: int
  sample: 502
group:
  description: group name of destination target
  returned: success
  type: string
  sample: "users"
uid:
  description: owner ID of destination target
  returned: success
  type: int
  sample: 502
owner:
  description: owner name of destination target
  returned: success
  type: string
  sample: "fred"
```

Continuing with this part of the module definition file, this section returns the details about the file's size, state, and permissions:

```
mode:
  description: permissions of the destination target
  returned: success
  type: int
  sample: 0644
size:
  description: size of destination target
  returned: success
  type: int
  sample: 20
state:
```

```
description: state of destination target
returned: success
type: string
sample: "file"
'''
```

Each returned item is listed with a description, the cases when the item would be in the return data, the type of item it is, and a sample of the value. The RETURN string is parsed by ansible-doc but the return values are sorted into alphabetical order, wherein the previous version of this book, we saw that the values were printed in the order in which they are listed in the module itself. The following screenshot shows this:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter10/example05". The window displays the documentation for the "remote\_copy" module. It includes sections for "OPTIONS", "dest", "source", "AUTHOR", "EXAMPLES", and "RETURN VALUES". The "RETURN VALUES" section lists "dest", "gid", and "group" with their respective descriptions, returned values, samples, types, and descriptions from the provided code block.

```
jfreeman@mastery1: ~/examples/Chapter10/example05
File Edit View Search Terminal Help
> REMOTE_COPY      (/home/jfreeman/examples/Chapter10/example05/library/remote_co>
    The remote_copy module copies a file on the remote host from a
    given source to a provided destination.

OPTIONS (= is mandatory):

= dest
    Path to the destination on the remote host for the copy

= source
    Path to a file on the source file on the remote host

AUTHOR: Jesse Keating

EXAMPLES:

# Example from Ansible Playbooks
- name: backup a config file
  remote_copy:
    source: /etc/herp/derp.conf
    dest: /root/herp-derp.conf.bak

RETURN VALUES:
- dest
    destination of the copy

    returned: success
    sample: /path/to/destination.file
    type: string

- gid
    group ID of destination target

    returned: success
    sample: 502
    type: int

- group
:
```

Figure 10.5 – Adding return data documentation to our module

In this way, we have built up a module that contains documentation that's incredibly useful for others if we are contributing it to the community, or even for ourselves when we come back to it after a while.

## Providing fact data

Similar to data returned as part of a module, such as `exit`, a module can directly create facts for a host by returning data in a key named `ansible_facts`. Providing facts directly from a module eliminates the need to register the return of a task with a subsequent `set_fact` task. To demonstrate this usage, let's modify our module to return the `source` and `dest` data as facts. Because these facts will become top-level host variables, we'll want to use more descriptive fact names than `source` and `dest`. Replace the current `module.exit_json` line in our module with the code listed here:

```
facts = {'rc_source': module.params['source'],
         'rc_dest': module.params['dest']}  
  
module.exit_json(changed=True, ansible_facts=facts)
```

We'll also add a task to our playbook to use one of the facts in a `debug` statement:

```
- name: show a fact
  ansible.builtin.debug:
    var: rc_dest
```

Now, running the playbook will show the new return data, plus the use of the variable, as shown in the following screenshot:

```
jfreeman@mastery1: ~/examples/Chapter10/example06
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter10/example06$ ansible-playbook -i mastery-hosts simple_module.yaml -v
No config file found; using defaults

PLAY [test remote_copy module] ****
TASK [ensure foo] ****
changed: [localhost] => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": true, "dest": "/tmp/rcfoo", "gid": 1001, "group": "jfreeman", "mode": "0664", "owner": "jfreeman", "size": 0, "state": "file", "uid": 1001}

TASK [do a remote copy] ****
changed: [localhost] => {"ansible_facts": {"rc_dest": "/tmp/rcbar", "rc_source": "/tmp/rcfoo"}, "changed": true}

TASK [show a fact] ****
ok: [localhost] => {
    "rc_dest": "/tmp/rcbar"
}

PLAY RECAP ****
localhost                  : ok=3    changed=2    unreachable=0    failed=0    s
kipped=0      rescued=0      ignored=0
```

Figure 10.6 – Adding facts to our custom module and viewing their values during playbook execution

If our module does not return facts (and our previous version of `remote_copy.py` didn't), we will have to register the output and use `set_fact` to create the fact for us, as shown in the following code:

```
- name: do a remote copy
  remote_copy:
    source: /tmp/rcfoo
    dest: /tmp/rcbar
    register: mycopy

- name: set facts from mycopy
  ansible.builtin.set_fact:
    rc_dest: "{{ mycopy.dest }}"
```

Although it is useful to be able to do this, when designing our modules, it is better to have the module define the facts required. If this is not done, then the previous register and the `set_fact` code would need to be repeated for every use of our module in a playbook!

## Check mode

Since the early days of its existence, Ansible has supported **check mode**, a mode of operation that will pretend to make changes to a system without actually changing the system. Check mode is useful for testing whether a change will happen, or whether a system state has drifted since the last Ansible run. Check mode depends on modules to support it and return data, as if it had completed the change. Supporting check mode in our module requires two changes; the first is to indicate that the module supports check mode, while the second is to detect when check mode is active and return data before execution.

## Supporting check mode

To indicate that a module supports check mode, an argument has to be set when creating the module object. This can be done before or after the `argument_spec` variable is defined in the module object; here, we will do it after it has been defined:

```
module = AnsibleModule(
    argument_spec = dict(
        source=dict(required=True, type='str'),
        dest=dict(required=True, type='str')
    ),
    supports_check_mode=True
)
```

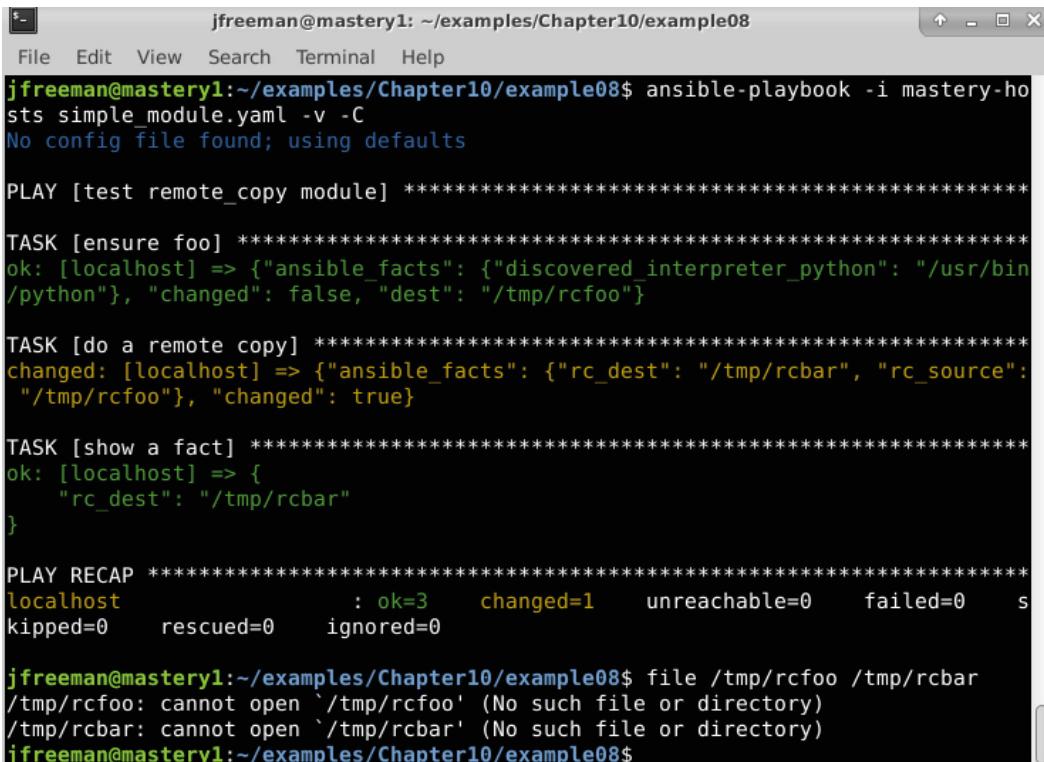
If you're modifying your existing code, don't forget to add the comma after the `argument_spec` dictionary definition, as shown in the preceding code.

## Handling check mode

Detecting when check mode is active is very easy. The module object will have a `check_mode` attribute, which will be set to a Boolean value of `true` when check mode is active. In our module, we want to detect whether check mode is active before performing the copy. We can simply move the copy action into an `if` statement to avoid copying when check mode is active. No further changes to the module are necessary beyond this:

```
if not module.check_mode:
    shutil.copy(module.params['source'],
                module.params['dest'])
```

Now, we can run our playbook and add the `-C` argument to our execution. This argument engages check mode. We'll also test to ensure that the playbook did not create and copy the files. The following screenshot shows this in action:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter10/example08". The command run is `ansible-playbook -i mastery-hosts simple_module.yaml -v -C`. The output shows the playbook running through three tasks: ensuring a file exists at /tmp/rcfoo, doing a remote copy from /tmp/rcfoo to /tmp/rcbar, and showing a fact about the rc\_dest path. The output indicates that no files were actually copied or created, as expected for check mode.

```
jfreeman@mastery1:~/examples/Chapter10/example08$ ansible-playbook -i mastery-hosts simple_module.yaml -v -C
No config file found; using defaults

PLAY [test remote_copy module] ****
TASK [ensure foo] ****
ok: [localhost] => {"ansible_facts": {"discovered_interpreter_python": "/usr/bin/python"}, "changed": false, "dest": "/tmp/rcfoo"}

TASK [do a remote copy] ****
changed: [localhost] => {"ansible_facts": {"rc_dest": "/tmp/rcbar", "rc_source": "/tmp/rcfoo"}, "changed": true}

TASK [show a fact] ****
ok: [localhost] => {
    "rc_dest": "/tmp/rcbar"
}

PLAY RECAP ****
localhost                  : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

jfreeman@mastery1:~/examples/Chapter10/example08$ file /tmp/rcfoo /tmp/rcbar
/tmp/rcfoo: cannot open '/tmp/rcfoo' (No such file or directory)
/tmp/rcbar: cannot open '/tmp/rcbar' (No such file or directory)
jfreeman@mastery1:~/examples/Chapter10/example08$
```

Figure 10.7 – Adding check mode support to our Ansible module

Although the module's output looks as though it created and copied files, we can see that the files referenced did not exist before execution and still do not exist after execution, a clear indication that our simple module was run in check mode.

Now that we've looked at our simple example module, we'll explore how to extend the functionality of Ansible through another important item – plugins.

## Developing plugins

Plugins are another way of extending or modifying the functionality of Ansible. While modules are executed as tasks, plugins are utilized in a variety of other places. Plugins are broken down into a few types, based on where they would plug into the Ansible execution. Ansible ships some plugins for each of these areas, and end users can create their own to extend the functionality of these specific areas.

## Connection-type plugins

Any time Ansible makes a connection to a host to perform a task, a connection plugin is used. Ansible ships with a few connection plugins, including `ssh`, `community.docker`, `docker`, `local`, and `winrm`. Additional connection mechanisms can be utilized by Ansible to connect to remote systems by creating a connection plugin, which may be useful if you must connect to some new type of system, such as a network switch, or perhaps your refrigerator someday. To create a new connection plugin, we would have to understand and work with an underlying communication protocol, which in itself could have a book devoted to it; as such, we won't attempt to create one here. However, the easiest way to get started is to read through the existing plugins that ship with Ansible and pick one to modify as necessary. The existing plugins can be found in `plugins/connection/` wherever the Ansible Python libraries are installed on your system, such as `/usr/local/lib/python3.8/dist-packages/ansible/plugins/connection/` on my system. You can also view them on GitHub – for example, if you wanted to look up the files relevant to the 2.11.1 release of `ansible-core`, you could look here: <https://github.com/ansible/ansible/tree/v2.11.1/lib/ansible/plugins/connection>.

## Shell plugins

Much like connection plugins, Ansible makes use of **shell plugins** to execute things in a shell environment. Each shell has subtle differences that Ansible cares about to properly execute commands, redirect output, discover errors, and other such interactions. Ansible supports several shells, including `sh`, `ansible.posix.csh`, `ansible.posix.fish`, and `powershell`. We can add more shells by implementing a new shell plugin. You can view the code for them (for the 2.11.1 release of `ansible-core`) here: <https://github.com/ansible/ansible/tree/v2.11.1/lib/ansible/plugins/shell>.

## Lookup plugins

**Lookup plugins** are how Ansible accesses outside data sources from the host system and implements language features, such as looping constructs (`loop` or `with_*`). A lookup plugin can be created to access data from an existing data store or to create a new looping mechanism. The existing lookup plugins can be found in `plugins/lookup/` or on GitHub here: <https://github.com/ansible/ansible/tree/v2.11.1/lib/ansible/plugins/lookup>. Lookup plugins can be added to introduce new ways of looping over content, or for looking up resources in external systems.

## Vars plugins

Constructs to inject variable data exist in the form of **vars plugins**. Data such as `host_vars` and `group_vars` are implemented via plugins. While it's possible to create new variable plugins, most often, it is better to create a custom inventory source or a fact module instead.

## Fact-caching plugins

Ansible can cache facts between playbook runs. Where facts are cached depends on the configured cache plugin that is used. Ansible includes plugins to cache facts in memory (they're not cached between runs as this is not persistent), `community.general.memcached`, `community.general.redis`, and `jsonfile`. Creating a **fact-caching plugin** can enable additional caching mechanisms.

## Filter plugins

While Jinja2 includes several filters, Ansible has made filters pluggable to extend the Jinja2 functionality. Ansible includes several filters that are useful for Ansible operations, and users of Ansible can add more. Existing plugins can be found in `plugins/filter/`.

To demonstrate the development of a filter plugin, we will create a simple filter plugin to do a silly thing to text strings. We will create a filter that will replace any occurrence of the words `the cloud` with the string `somebody else's computer`. We'll define our filter in a file within a new directory, `filter_plugins/`, in our existing working directory. The name of the file doesn't matter, as we'll define the name of the filter within the file; so, let's name our file `filter_plugins/sample_filter.py`.

First, we need to define the function that will perform the translation and provide the code to translate the strings:

```
def cloud_truth(a):
    return a.replace("the cloud", "somebody else's computer")
```

Next, we'll need to construct a `FilterModule` object and define our filter within it. This object is what Ansible will load, and Ansible expects there to be a `filters` function within the object that returns a set of filter names to functions within the file:

```
class FilterModule(object):
    '''Cloud truth filters'''
    def filters(self):
        return {'cloud_truth': cloud_truth}
```

Now, we can use this filter in a playbook, which we'll call `simple_filter.yaml`:

```
---
- name: test cloud_truth filter
  hosts: localhost
  gather_facts: false
  vars:
    statement: "I store my files in the cloud"
  tasks:
    - name: make a statement
      ansible.builtin.debug:
        msg: "{{ statement | cloud_truth }}"
```

Now, let's run our playbook and see our filter in action:

```
jfreeman@mastery1: ~/examples/Chapter10/example09$ ansible-playbook -i mastery-ho sts simple_filter.yaml

PLAY [test cloud_truth filter] ****
TASK [make a statement] ****
ok: [localhost] => {
    "msg": "I store my files in somebody else's computer"
}

PLAY RECAP ****
localhost                  : ok=1      changed=0      unreachable=0      failed=0      s
kipped=0      rescued=0      ignored=0
```

Figure 10.8 – Executing a playbook to test our new filter plugin

Our filter worked, and it turned the words `the cloud` into `somebody else's computer`. This is a silly example without any error handling, but it demonstrates our capability to extend Ansible and Jinja2's filter capabilities.

#### Important Note

Although the name of the file that contains a filter definition can be whatever the developer wants, a best practice is to name it after the filter itself so that it can easily be found in the future, potentially by other collaborators. This example did not follow this, to demonstrate that the filename is not attached to the filter name.

## Callback plugins

**Callbacks** are places in Ansible execution that can be plugged into for added functionality. There are expected callback points that can be registered against to trigger custom actions at those points. Here is a list of possible points that can be used to trigger functionality at the time of writing:

- `v2_on_any`
- `v2_runner_on_failed`
- `v2_runner_on_ok`

- v2\_runner\_on\_skipped
- v2\_runner\_on\_unreachable
- v2\_runner\_on\_async\_poll
- v2\_runner\_on\_async\_ok
- v2\_runner\_on\_async\_failed
- v2\_runner\_on\_start
- v2\_playbook\_on\_start
- v2\_playbook\_on\_notify
- v2\_playbook\_on\_no\_hosts\_matched
- v2\_playbook\_on\_no\_hosts\_remaining
- v2\_playbook\_on\_task\_start
- v2\_playbook\_on\_cleanup\_task\_start
- v2\_playbook\_on\_handler\_task\_start
- v2\_playbook\_on\_vars\_prompt
- v2\_playbook\_on\_import\_for\_host
- v2\_playbook\_on\_not\_import\_for\_host
- v2\_playbook\_on\_play\_start
- v2\_playbook\_on\_stats
- v2\_on\_file\_diff
- v2\_playbook\_on\_include
- v2\_runner\_item\_on\_ok
- v2\_runner\_item\_on\_failed
- v2\_runner\_item\_on\_skipped
- v2\_runner\_retry

As an Ansible run reaches each of these states, any plugins that have code to run at these points will be executed. This provides the tremendous ability to extend Ansible without having to modify the base code.

Callbacks can be utilized in a variety of ways: to change how things are displayed on the screen, to update a central status system of progress, to implement a global locking system, or nearly anything imaginable. It's the most powerful way to extend the functionality of Ansible. However, you will note that the previously listed items do not appear on the official Ansible documentation website (<https://docs.ansible.com>), nor are they listed by the `ansible-doc` command. A great place to go to look up these callbacks to learn more about them is the `plugins/callback/__init__.py` file, under your `ansible-core` installation directory. For example, on my system, where Ansible was installed with pip, the full path is `/usr/local/lib/python3.8/dist-packages/ansible/plugins/callback/__init__.py` (if you want to look this up on the internet, the file for the 2.11.1 release of `ansible-core` can be found here: [https://github.com/ansible/ansible/blob/v2.11.1/lib/ansible/plugins/callback/\\_\\_init\\_\\_.py](https://github.com/ansible/ansible/blob/v2.11.1/lib/ansible/plugins/callback/__init__.py)).

To demonstrate our ability to develop a callback plugin, we'll create a simple plugin that will print something silly on the screen when the playbook prints the summary of the play at the end:

1. First, we'll need to make a new directory to hold our callback. The location Ansible will look for is `callback_plugins/`. Unlike the `filter` plugin earlier, we do need to name our callback plugin file carefully, as it will also have to be reflected in an `ansible.cfg` file.
2. We'll name ours `callback_plugins/shrug.py`. As Ansible versions greater than 3.0 are being moved toward Python 3 support only (though Python 2.7 is supported still at the time of writing), your plugin code should be written for Python 3. Start by adding the following Python 3 header to your plugin:

```
from __future__ import (absolute_import, division, print_
function)
__metaclass__ = type
```

3. Next up, you will need to add a documentation block, much like we did in the *Developing modules* section of this chapter. In the previous edition of this book, there was no need to do this, but now, you will get a deprecation warning if you don't, and your callback plugin may not work when `ansible-core` 2.14 is released. Our documentation block will look like this:

```
DOCUMENTATION = '''
callback: shrug
type: stdout
short_description: modify Ansible screen output
version_added: 4.0
```

```

description:
    - This modifies the default output callback for
      ansible-playbook.

extends_documentation_fragment:
    - default_callback

requirements:
    - set as stdout in configuration

'''
```

Most of the items in the documentation are self-explanatory, but the `extends_documentation_fragment` item is worth noting. This particular part of the documentation block is the piece that is required for compatibility with `ansible-core` 2.14, and as we are extending the `default_callback` plugin here, we need to tell Ansible that we are extending this piece of documentation.

4. With this complete, we'll need to create a `CallbackModule` class, subclassed from `CallbackModule`, defined in the `default` callback plugin found in `ansible.plugins.callback.default`, since we only need to change one aspect of the normal output.
5. Within this class, we will define variable values to indicate that it is a `2.0` version callback, that it is an `stdout` type of callback, and that it has the name `shrug`.
6. Also, within this class, we must initialize it so that we can define one or more of the callback points listed earlier that we'd like to plug into to make something happen. In our example, we want to modify the display of the playbook summary that is produced at the end of the run, for which we will modify the `v2_playbook_on_stats` callback.
7. To round off our plugin, we must call the original callback module itself. Ansible now only supports one `stdout` plugin at a time, so if we don't call the original plugin, we will find that the output from our plugin is the only output that's produced – all the other information regarding the playbook run will be missing! The final code below the documentation block should look like this:

```

from ansible.plugins.callback.default import
    CallbackModule as CallbackModule_default

class CallbackModule(CallbackModule_default):
    CALLBACK_VERSION = 2.0
    CALLBACK_TYPE = 'stdout'
    CALLBACK_NAME = 'shrug'
```

```
def __init__(self):
    super(CallbackModule, self).__init__()

def v2_playbook_on_stats(self, stats):
    msg = b'\xc2\xaf\\_\(\xe3\x83\x84)_/\xc2\xaf'
    self._display.display(msg.decode('utf-8') * 8)
    super(CallbackModule, self).v2_playbook_on_
stats(stats)
```

- As this callback is `stdout_callback`, we'll need to create an `ansible.cfg` file and, within it, indicate that the `shrub stdout` callback should be used. The `ansible.cfg` file can be found in `/etc/ansible/` or in the same directory as the playbook:

```
[defaults]
stdout callback = shrug
```

- That's all we have to write in our callback. Once it's saved, we can rerun our previous playbook, which exercised our `sample_filter`, but this time, we'll see something different on the screen:

```
jfreeman@mastery1: ~/examples/Chapter10/example10
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter10/example10$ ansible-playbook -i mastery-hosts simple_filter.yaml

PLAY [test cloud_truth filter] ****
TASK [make a statement] ****
ok: [localhost] => {
    "msg": "I store my files in somebody else's computer"
}
PLAY RECAP ****
localhost                  : ok=1    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 10.9 – Adding our shrug plugin to modify the playbook run output

This is very silly, but it demonstrates the ability to plug into various points of a playbook execution. We chose to display a series of shrugs on screen, but we could have just as easily interacted with some internal audit and control system to record actions, or to report progress to an IRC or Slack channel.

## Action plugins

**Action plugins** exist to hook into the task construct without actually causing a module to be executed, or to execute code locally on the Ansible host before executing a module on the remote host. Several action plugins are included with Ansible and they can be found in `plugins/action/`. One such action plugin is the `template` plugin, which can be used in place of a `template` module. When a playbook author writes a `template` task, that task will call the `template` plugin to do the work. The plugin, among other things, will render the template locally before copying the content to the remote host. Because actions have to happen locally, the work is done by an action plugin. Another action plugin we should be familiar with is the `debug` plugin, which we've used heavily in this book to print content. Creating a custom action plugin is useful when we're trying to accomplish both local work and remote work in the same task.

## Distributing plugins

Much like distributing custom modules, there are standard places to store custom plugins alongside playbooks that expect to use plugins. The default locations for plugins are the locations that are shipped with the Ansible code install, subdirectories within `~/.ansible/plugins/`, and subdirectories of the project root (the place where the top-level playbook is stored). Plugins can be distributed within the same subdirectories of a role as well, as well as collections, as we covered in *Chapter 2, Migrating from Earlier Ansible Versions*. To utilize plugins from any other location, we need to define the location to find the plugin for the plugin type in an `ansible.cfg` file or reference the collection, as we demonstrated by loading our example filter module in *Chapter 2, Migrating from Earlier Ansible Versions*.

If you're distributing plugins inside the project root, each plugin type gets its own top-level directory:

- `action_plugins/`
- `cache_plugins/`
- `callback_plugins/`
- `connection_plugins/`
- `shell_plugins/`
- `lookup_plugins/`
- `vars_plugins/`
- `filter_plugins/`

As with other Ansible constructs, the first plugin with a given name that's found will be used, and just as with modules, the paths relative to the project root are checked first, allowing a local override of an existing plugin. Simply place the filter file in the appropriate subdirectory, and it will be automatically used when referenced.

## Developing dynamic inventory plugins

**Inventory plugins** are bits of code that will create inventory data for an Ansible execution. In many environments, the simple `ini` file-style inventory source and variable structure are not sufficient for representing the actual infrastructure being managed. In such cases, a dynamic inventory source is desired, one that will discover the inventory and data at runtime at every execution of Ansible. A number of these dynamic sources ship with Ansible, primarily to operate Ansible with the infrastructure built into one cloud computing platform or another. A short, incomplete list of dynamic inventory plugins that ship with Ansible 4.3 (there are now over 40) includes the following – note from the FQCNs that many of these that were once shipped as part of the Ansible 2.x releases are now being included as part of the wider set of collections that form Ansible 4.3:

- `azure.azcollection.azure_rm`
- `community.general.cobbler`
- `community.digitalocean.digitalocean`
- `community.docker.docker_containers`
- `amazon.aws.aws_ec2`
- `google.cloud.gcp_compute`
- `community.libvirt.libvirt`
- `community.general.linode`
- `kubernetes.core.openshift`
- `openstack.cloud.openstack`
- `community.vmware.vmware_vm_inventory`
- `servicenow.servicenow.now`

An inventory plugin is essentially an executable script. Ansible calls the script with set arguments (`--list` or `--host <hostname>`) and expects JSON formatted output on `STDOUT`. When the `--list` argument is provided, Ansible expects a list of all the groups to be managed. Each group can list host membership, child group membership, and group variable data. When the script is called with the `--host <hostname>` argument, Ansible expects host-specific data to be returned (or an empty JSON dictionary).

Using a dynamic inventory source is easy. A source can be used directly by referring to it with the `-i (--inventory-file)` option to `ansible` and `ansible-playbook`, or by putting the plugin file inside the directory referred to by the inventory path in `ansible.cfg`.

Before creating an inventory plugin, we must understand the expected format for when `--list` or `--host` is used with our script.

## **List**ing hosts

When the `--list` argument is passed to an inventory script, Ansible expects the JSON output data to have a set of top-level keys. These keys are named for the groups in the inventory. Each group gets a key. The structure within a group key varies, depending on what data needs to be represented in the group. If a group just has hosts and no group-level variables, the data within the key can simply be a list of hostnames. If the group has variables or children (a group of groups), then the data needs to be a hash, which can have one or more keys named `hosts`, `vars`, or `children`. The `hosts` and `children` subkeys have list values, a list of the hosts that exist in the group, or a list of the child groups. The `vars` subkey has a hash value, where each variable's name and value is represented by a key and value.

## **List**ing host variables

When the `--host <hostname>` argument is passed to an inventory script, Ansible expects the JSON output data to simply be a hash of the variables, where each variable name and value is represented by a key and a value. If there are no variables for a given host, an empty hash is expected.

## Simple inventory plugin

To demonstrate developing an inventory plugin, we'll create one that simply prints some static inventory host data – it won't be dynamic, but this is a great first step to understanding the basics and the output formats required. This is based on some of the inventories we have used throughout this book, so they may seem familiar in parts. We'll write our inventory plugin to a file in the top level of our project root named `mastery-inventory.py` and make it executable. We'll use Python for this file, to handle execution arguments and JSON formatting with ease, but remember that you can write inventory scripts in any language that you please, so long as they produce the required JSON output:

1. First, we'll need to add a shebang line to indicate that this script is to be executed with Python:

```
#!/usr/bin/env python
#
```

2. Next, we'll need to import a couple of Python modules that we will need later in our plugin:

```
import json
import argparse
```

3. Now, we'll create a Python dictionary to hold all of our groups. Some of our groups just have hosts while others have variables or children. We'll format each group accordingly:

```
inventory = {}
inventory['web'] = {'hosts': ['mastery.example.name'],
                   'vars': {'http_port': 80,
                            'proxy_timeout': 5}}
inventory['dns'] = {'hosts': ['backend.example.name']}
inventory['database'] = {'hosts': ['backend.example.name'],
                        'vars': {'ansible_ssh_user':
                                'database'}}
inventory['frontend'] = {'children': ['web']}
inventory['backend'] = {'children': ['dns', 'database']}
```

```

        'vars': {'ansible_ssh_user':
'blotto'}}
inventory['errors'] = {'hosts': ['scsihost']}
inventory['failtest'] = {'hosts': ["failer%02d" % n for n
in
range(1,11)]}

```

4. To create our `failtest` group (you'll see this in action in the next chapter), which in our inventory file will be represented as `failer[01:10]`, we can use a Python list comprehension to produce the list for us, formatting the items in the list just the same as our ini-formatted inventory file. Every other group entry should be self-explanatory.
5. Our original inventory also had an `all` group variable, which provided a default variable, `ansible_ssh_user`, to all groups (which groups could override), which we'll define here and make use of later in the file:

```
allgroupvars = {'ansible_ssh_user': 'otto'}
```

6. Next, we need to enter the host-specific variables in their own dictionary. Only one node in our original inventory had host-specific variables – we'll also add a new host, `scsihost`, to develop our example further:

```

hostvars = {}
hostvars['mastery.example.name'] = {'ansible_ssh_host':
'192.168.10.25'}
hostvars['scsihost'] = {'ansible_ssh_user': 'jfreeman'}

```

7. With all our data defined, we can now move on to the code that will handle argument parsing. This can be done via the `argparse` module we imported earlier in the file:

```

parser = argparse.ArgumentParser(description='Simple
Inventory')

parser.add_argument('--list', action='store_true',
help='List all hosts')

parser.add_argument('--host', help='List details of a
host')

args = parser.parse_args()

```

8. After parsing the arguments, we can deal with either the `--list` or `--host` actions. If a list is requested, we simply print a JSON representation of our inventory. This is where we'll take the `allgroupvars` data into account; the default `ansible_ssh_user` for each group. We'll loop through each group, create a copy of the `allgroupvars` data, update that data with any data that may already exist in the group, then replace the group's variable data with the newly updated copy. Finally, we'll print out the result:

```
if args.list:  
    for group in inventory:  
        ag = allgroupvars.copy()  
        ag.update(inventory[group].get('vars', {}))  
        inventory[group]['vars'] = ag  
    print(json.dumps(inventory))
```

9. Finally, we'll handle the `--host` action by building up a dictionary of all the variables that can be applied to the host that is passed to this script. We'll do this using an approximation of the precedence order that's used in Ansible when parsing an ini format inventory. This code is iterative, and the nested loops would not be efficient in a production environment, but for this example, it serves us well. The output is the JSON formatted variable data for the provided host, or an empty hash if there is no host-specific variable data for the provided host:

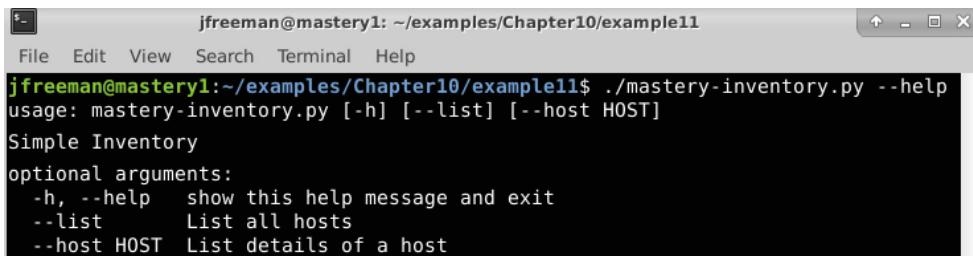
```
elif args.host:  
    hostfound = False  
    agghostvars = allgroupvars.copy()  
    for group in inventory:  
        if args.host in inventory[group].get('hosts', {}):  
            hostfound = True  
            for childgroup in inventory:  
                if group in inventory[childgroup].  
                    get('children', {}):  
                    agghostvars.  
                    update(inventory[childgroup].get('vars', {}))  
            for group in inventory:  
                if args.host in inventory[group].get('hosts', {}):  
                    hostfound = True
```

```

        agghostvars.update(inventory[group].
get('vars', {}))
        if hostvars.get(args.host, {}):
            hostfound = True
            agghostvars.update(hostvars.get(args.host, {}))
        if not hostfound:
            agghostvars = {}
print(json.dumps(agghostvars))

```

Now, our inventory is ready to be tested! We can execute it directly and pass the `--help` argument we get for free using argparse. This will show us the usage of our script based on the argparse data we provided earlier in the file:



```
jfreeman@mastery1:~/examples/Chapter10/example11$ ./mastery-inventory.py --help
usage: mastery-inventory.py [-h] [--list] [--host HOST]

Simple Inventory

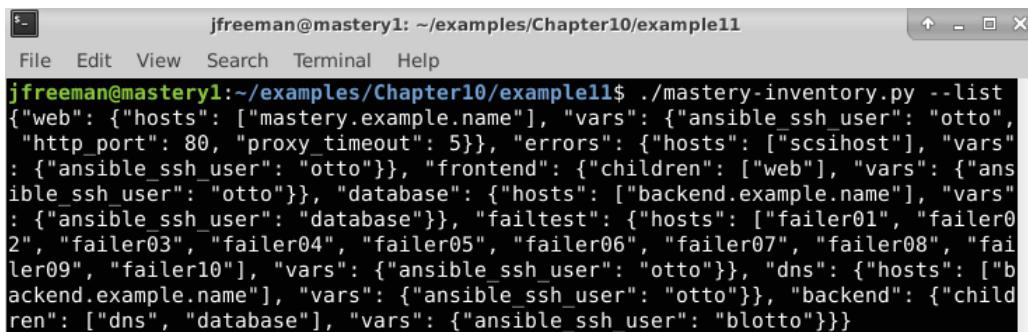
optional arguments:
  -h, --help    show this help message and exit
  --list        List all hosts
  --host HOST  List details of a host
```

Figure 10.10 – Testing the built-in help function of our dynamic inventory script

#### Important Note

Don't forget to make the dynamic inventory script executable; for example,  
`chmod +x mastery-inventory.py`.

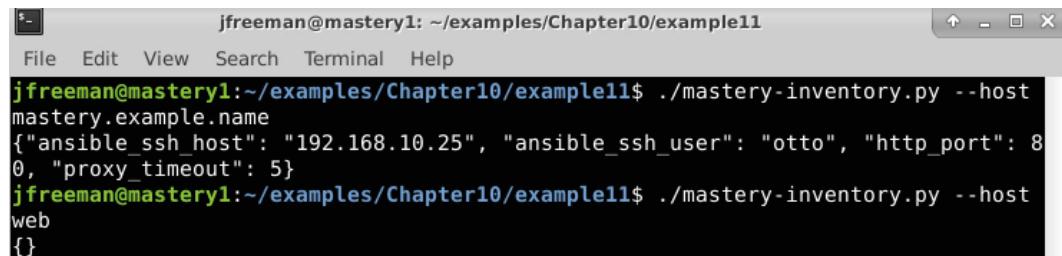
If we pass `--list`, we'll get the output of all our groups, along with all the hosts in each group and all the associated inventory variables:



```
jfreeman@mastery1:~/examples/Chapter10/example11$ ./mastery-inventory.py --list
{"web": {"hosts": ["mastery.example.name"], "vars": {"ansible_ssh_user": "otto", "http_port": 80, "proxy_timeout": 5}}, "errors": {"hosts": ["scsihost"], "vars": {"ansible_ssh_user": "otto"}}, "frontend": {"children": ["web"], "vars": {"ansible_ssh_user": "otto"}}, "database": {"hosts": ["backend.example.name"], "vars": {"ansible_ssh_user": "database"}}, "failtest": {"hosts": ["failer01", "failer02", "failer03", "failer04", "failer05", "failer06", "failer07", "failer08", "failer09", "failer10"], "vars": {"ansible_ssh_user": "otto"}}, "dns": {"hosts": ["backend.example.name"], "vars": {"ansible_ssh_user": "otto"}}, "backend": {"children": ["dns", "database"], "vars": {"ansible_ssh_user": "blotto"}}}
```

Figure 10.11 – Displaying the JSON output produced by the `--list` parameter of our dynamic inventory script

Similarly, if we run this Python script with the `--host` argument and a hostname we know is in the inventory, we'll see the host variables for the hostname that was passed. If we pass a group name, nothing should be returned, as the script only returns data for valid individual hostnames:



```
jfreeman@mastery1: ~/examples/Chapter10/example11$ ./mastery-inventory.py --host mastery.example.name
{"ansible_ssh_host": "192.168.10.25", "ansible_ssh_user": "otto", "http_port": 80, "proxy_timeout": 5}
jfreeman@mastery1:~/examples/Chapter10/example11$ ./mastery-inventory.py --host web
{}
```

Figure 10.12 – Displaying the JSON output produced by the `--list` parameter of our dynamic inventory script

Now, we're ready to use our inventory file with Ansible. Let's make a new playbook (`inventory_test.yaml`) to display the hostname and the ssh username data:

```
---
- name: test the inventory
  hosts: all
  gather_facts: false

  tasks:
  - name: hello world
    ansible.builtin.debug:
      msg: "Hello world, I am {{ inventory_hostname }}.
            My username is {{ ansible_ssh_user }}"
```

There is one more thing we have to do before we can use our new inventory plugin. By default (and as a security feature), most of Ansible's inventory plugins are disabled. To ensure our dynamic inventory script will run, open the applicable `ansible.cfg` file in an editor and look for the `enable_plugins` line in the `[inventory]` section. At a minimum, it should look like this (though you may choose to enable more plugins if you wish):

```
[inventory]
enable_plugins = ini, script
```

To use our new inventory plugin with this playbook, we can simply refer to the plugin file with the `-i` argument. Because we are using the `all` hosts group in our playbook, we'll also limit the run to a few groups to save screen space. We'll also time the execution, which will become important in the next section, so run the following command to execute the playbook:

```
time ansible-playbook -i mastery-inventory.py inventory_test.yaml --limit backend,frontend,errors
```

The output from this run should look as follows:

```
jfreeman@mastery1: ~/examples/Chapter10/example11$ time ansible-playbook -i mastery-inventory.py inventory_test.yaml --limit backend,frontend,errors

PLAY [test the inventory] ****
TASK [hello world] ****
ok: [scsihost] => {
    "msg": "Hello world, I am scsihost. My username is jfreeman"
}
ok: [mastery.example.name] => {
    "msg": "Hello world, I am mastery.example.name. My username is otto"
}
ok: [backend.example.name] => {
    "msg": "Hello world, I am backend.example.name. My username is database"
}

PLAY RECAP ****
backend.example.name      : ok=1    changed=0    unreachable=0    failed=0    s
mastery.example.name     : ok=1    changed=0    unreachable=0    failed=0    s
scsihost                  : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
kipped=0    rescued=0    ignored=0
kipped=0    rescued=0    ignored=0

real    0m1.570s
user    0m1.251s
sys     0m0.388s
jfreeman@mastery1:~/examples/Chapter10/example11$
```

Figure 10.13 – Running a test playbook against our dynamic inventory script

As you can see, we get the hosts we expect, and we get the default ssh user for `mastery.example.name`. `backend.example.name` and `scsihost` each show their host-specific ssh usernames.

## Optimizing script performance

With this inventory script, when Ansible starts, it will execute the script once with `--list` to gather the group data. Then, Ansible will execute the script again with `--host <hostname>` for each host it discovered in the first call. With our script, this takes very little time as there are very few hosts and our execution is very fast. However, in an environment with a large number of hosts or a plugin that takes a while to run, gathering the inventory data can be a lengthy process. Fortunately, there is an optimization that can be made in the return data from a `--list` call that will prevent Ansible from rerunning the script for every host. The host-specific data can be returned all at once inside the group data return, inside a top-level key named `_meta`, which has a subkey named `hostvars` that contains a hash of all the hosts that have host variables and the variable data itself. When Ansible encounters a `_meta` key in the `--list` return, it'll skip the `--host` calls and assume that all of the host-specific data was already returned, potentially saving a significant amount of time! Let's modify our inventory script to return host variables inside `_meta`, and then create an error condition inside the `--host` option to show that `--host` is not being called:

1. First, we'll add the `_meta` key to the inventory dictionary once all of `hostvars` has been built up using the same algorithm as before, and just before argument parsing:

```
hostvars['scsihost'] = {'ansible_ssh_user': 'jfreeman'}
```

```
agghostvars = dict()
for outergroup in inventory:
    for grouphost in inventory[outergroup].get('hosts',
{}):
        agghostvars[grouphost] = allgroupvars.copy()
        for group in inventory:
            if grouphost in inventory[group].get('hosts',
{}):
                for childgroup in inventory:
                    if group in inventory[childgroup].
get('children', {}):
                        agghostvars[grouphost].
update(inventory[childgroup].get('vars', {}))
                for group in inventory:
                    if grouphost in inventory[group].get('hosts',
{}):
```

```

        agghostvars[grouphost].
update(inventory[group].get('vars', {}))
        agghostvars[grouphost].update(hostvars.
get(grouphost, {}))

inventory['_meta'] = {'hostvars': agghostvars}

parser = argparse.ArgumentParser(description='Simple
Inventory')

```

Next, we'll change the --host handling to raise an exception:

```

elif args.host:
    raise StandardError("You've been a bad person")

```

2. Now, we'll rerun the inventory\_test.yaml playbook using the same command as we did previously to ensure that we're still getting the right data:

```

jfreeman@mastery1:~/examples/Chapter10/example12
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter10/example12$ time ansible-playbook -i mastery-inventory.py inventory_test.yaml --limit backend,frontend,errors

PLAY [test the inventory] ****
TASK [hello world] ****
ok: [scsihost] => {
    "msg": "Hello world, I am scsihost. My username is jfreeman"
}
ok: [mastery.example.name] => {
    "msg": "Hello world, I am mastery.example.name. My username is otto"
}
ok: [backend.example.name] => {
    "msg": "Hello world, I am backend.example.name. My username is database"
}

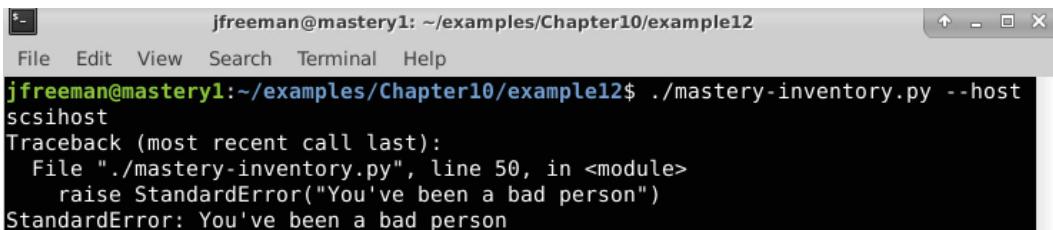
PLAY RECAP ****
backend.example.name      : ok=1    changed=0    unreachable=0    failed=0    s
skipped=0    rescued=0    ignored=0
mastery.example.name     : ok=1    changed=0    unreachable=0    failed=0    s
skipped=0    rescued=0    ignored=0
scsihost          : ok=1    changed=0    unreachable=0    failed=0    s
skipped=0    rescued=0    ignored=0

real    0m1.248s
user    0m1.015s
sys     0m0.286s
jfreeman@mastery1:~/examples/Chapter10/example12$

```

Figure 10.14 – Running our optimized dynamic inventory script

3. Just to be sure, we'll manually run the inventory plugin with the `--host` argument to show the exception:



```
jfreeman@mastery1: ~/examples/Chapter10/example12$ ./mastery-inventory.py --host scsihost
Traceback (most recent call last):
  File "./mastery-inventory.py", line 50, in <module>
    raise StandardError("You've been a bad person")
StandardError: You've been a bad person
```

Figure 10.15 – Demonstrating that the `--host` parameter does not work on our newly optimized script

With this optimization, our simple playbook, which is using our inventory module, now runs a good few percent faster because of the gained efficiency in inventory parsing. This might not seem like much here, but when scaled up to a more complex inventory, this would be significant.

## Contributing to the Ansible project

Not all modifications need to be for local site requirements. Ansible users will often identify an enhancement that could be made to the project that others would benefit from. These enhancements can be contributed via a collection, and in the new structure of Ansible that proceeds from version 3.0, this is likely to be the most suitable route for most people. In this case, you will be able to follow the guidance given in *Chapter 2, Migrating from Earlier Ansible Versions*, to build and release a collection. However, what if you develop the next killer plugin or filter that should be added to the `ansible-core` project itself? In this section, we'll look at how you can do this. Contributions could be in the form of updates to an existing built-in module or core Ansible code, updates to documentation, new filters or plugins, or simply testing proposed contributions from other community members.

## Contribution submissions

The Ansible project uses GitHub (<https://github.com>) to manage code repositories, issues, and other aspects of the project. The Ansible organization (<https://github.com/ansible>) is where the code repositories can be found. The main repository is the `ansible` repository (which now houses the `ansible-core` code) and for legacy reasons, it is located here: <https://github.com/ansible/ansible>. This is where the `ansible-core` code, the built-in modules, and the documentation can be found. This is the repository that should be forked to develop a contribution.

**Important Note**

The Ansible project uses a development branch named `devel` instead of the traditional name of `master`. Most contributions target the `devel` branch or a stable release branch.

## The Ansible repository

The Ansible repository has several files and folders at its root. These files are mostly high-level documentation files, code licenses, or continuous integration test platform configurations.

Of the directories, a few are worth noting:

- `bin`: Source for the various ansible core executables
- `docs`: Source for the API documentation, the <https://docs.ansible.com> website, and the main pages
- `hacking`: Guides and utilities for hacking on the Ansible source
- `lib/ansible`: The core Ansible source code
- `test`: Unit and integration testing code

Contributions to Ansible will likely occur in one of those key folders.

## Executing tests

Before any submission can be accepted by Ansible, the change must pass tests. These tests fall into three categories: unit tests, integration tests, and code-style tests. Unit tests cover very narrow aspects of source code functions, while integration tests take a more holistic approach and ensure the desired functionality happens. Code-style tests examine the syntax used, as well as whitespace and other style aspects.

Before any tests can be executed, the shell environment must be prepared to work with the Ansible code checkout. A shell environment file exists to set the required variables, which can be activated with this command:

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

Ensuring tests are passing before modifications are made can save a lot of debugging time later, as the `devel` branch is bleeding edge and there are possibilities that code that has been committed to this branch does not pass all tests.

## Unit tests

All of the unit tests are located within the directory tree starting at `test/units`. These tests should all be self-contained and do not require access to external resources. Running the tests is as simple as executing `make tests` from the root of the Ansible source checkout. This will test much of the code base, including the module code.

### Important Note

Executing the tests may require installing additional software. When using a Python virtualenv to manage Python software installations, it's best to create a new venv to use for testing Ansible – one that does not have Ansible installed in it.

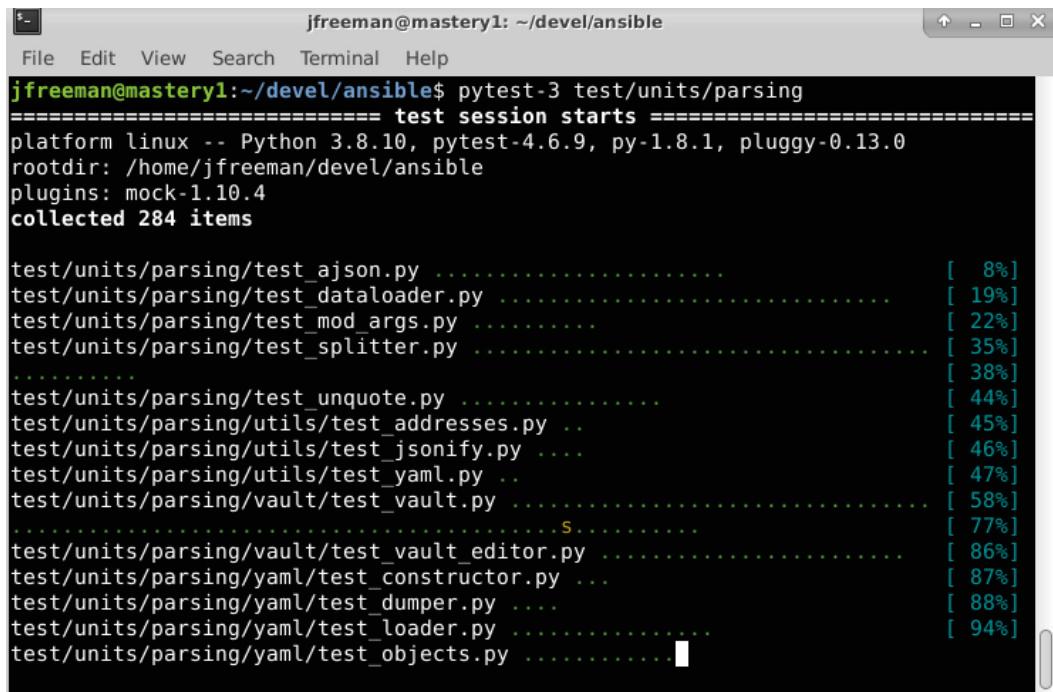
To target a specific set of tests to run, the `pytest` (sometimes accessed as `py.test`) utility can be called directly, with a path provided to a directory or a specific file to test. On Ubuntu Server 20.04, you can install this tool with the following command:

```
sudo apt install python3-pytest
```

Assuming you have checked out the `ansible-core` repository code, you could run just the parsing unit tests with the following commands. Note that some of the tests require that you install additional Python modules and that Ansible runs under Python 3 now by default, so you should always ensure you are installing and working with Python 3-based modules and tools. The following commands may not be sufficient for running all tests, but they are sufficient for running the parsing tests and give you an idea of the kinds of things you need to do to prepare for running the included test suite:

```
sudo apt install python3-pytest python3-tz python3-pytest-mock
cd ansible
source ./hacking/env-setup
pytest-3 test/units/parsing
```

The output should look as follows if all goes well, with any warnings and/or errors displayed, along with a summary at the end:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/devel/ansible". The terminal shows the command "pytest-3 test/units/parsing" being run. The output indicates a "test session starts" with platform details: "platform linux -- Python 3.8.10, pytest-4.6.9, py-1.8.1, pluggy-0.13.0", root directory "/home/jfreeman/devel/ansible", and plugins "mock-1.10.4". It then lists "collected 284 items". The subsequent output shows the progress of running various test files, with completion percentages in brackets: test/units/parsing/test\_ajson.py [ 8%], test/units/parsing/test\_dataloader.py [ 19%], test/units/parsing/test\_mod\_args.py [ 22%], test/units/parsing/test\_splitter.py [ 35%], test/units/parsing/test\_unquote.py [ 38%], test/units/parsing/utils/test\_addresses.py [ 44%], test/units/parsing/utils/test\_jsonify.py [ 45%], test/units/parsing/utils/test\_yaml.py [ 46%], test/units/parsing/vault/test\_vault.py [ 47%], test/units/parsing/vault/test\_vault\_editor.py [ 58%], test/units/parsing/yaml/test\_constructor.py [ 77%], test/units/parsing/yaml/test\_dumper.py [ 86%], test/units/parsing/yaml/test\_loader.py [ 87%], test/units/parsing/yaml/test\_objects.py [ 88%], and test/units/parsing/yaml/test\_objects.py [ 94%].

Figure 10.16 – Using the `pytest-3` utility for Python 3 to run the parsing unit tests included with the `ansible-core` source code

As you can see, the `pytest-3` utility is running through the defined unit tests and will report any errors it finds, aiding you greatly in checking any code you might be planning to submit. Everything seems to be going well in the preceding screenshot!

## Integration tests

Ansible integration tests are tests designed to validate playbook functionality. Testing is executed by playbooks as well, making things a bit recursive. The tests are broken down into a few main categories:

- Non-destructive
- Destructive
- Legacy Cloud
- Windows
- Network

A more detailed explanation of these test categories can be found here: [https://docs.ansible.com/ansible/latest/dev\\_guide/testing\\_integration.html](https://docs.ansible.com/ansible/latest/dev_guide/testing_integration.html).

#### Important Note

Many of the integration tests require ssh to localhost to be functional. Be sure that ssh works, ideally without a password prompt. Remote hosts can be used by altering the inventory file included with specific integration tests that require them. For example, if you are running the connection\_ssh integration tests, be sure to look in test/integration/targets/connection\_ssh/test\_connection.inventory and update it as required. It is left as an exercise for you to explore this directory tree and locate the appropriate inventory files that you might need to update.

As with unit tests, individual integration tests can be executed by using the ansible-test utility located at bin/ansible-test. Many of the integration tests require external resources, such as computer cloud accounts, and again, you will need to explore the documentation and directory tree to establish what you need to configure to run these tests in your environment. Each directory in test/integration/targets is a target that can be tested individually. Let's choose a simple example for testing ping functionality with the ping target. This can be done with the following commands:

```
source ./hacking/env-setup
ansible-test integration --python 3.8 ping
```

Note that we have specifically set the Python environment to test against here. This is important as my Ubuntu Server 20.04 test machine has some Python 2.7 installed and Ansible has been installed and configured to use Python 3.8 (which is also present). If the ansible-test tool makes use of the Python 2.7 environment, it might find that modules are missing and the tests will fail, but not because of anything that's wrong with our code – rather because we have failed to set the environment up correctly.

When you run ansible-test, ensure you know which Python environment you are using and set it accordingly in the command. If you want to test against another Python version, you will need to ensure that all the prerequisite Python modules that Ansible depends upon (such as Jinja2) are installed under that Python environment.

A successful test run should look as follows:

```
jfreeman@mastery1: ~/devel/ansible
File Edit View Search Terminal Help
ok: [testhost]

TASK [ping : assert the ping worked with data] *****
ok: [testhost] => {
    "changed": false,
    "msg": "All assertions passed"
}

TASK [ping : ping with data=crash] *****
An exception occurred during task execution. To see the full traceback, use -vvv
. The error was: Exception: boom
fatal: [testhost]: FAILED! => {"changed": false, "module_stderr": "Traceback (most recent call last):\n  File \"/home/jfreeman/.ansible/tmp/ansible-tmp-1626455041.0660515-7910-251618093354542/AnsiballZ_ping.py\", line 114, in <module>\n    _ansiballz_main()\n  File \"/home/jfreeman/.ansible/tmp/ansible-tmp-1626455041.0660515-7910-251618093354542/AnsiballZ_ping.py\", line 106, in _ansiballz_main\n      invoke_module(zipped_mod, temp_path, ANSIBALLZ_PARAMS)\n  File \"/home/jfreeman/.ansible/tmp/ansible-tmp-1626455041.0660515-7910-251618093354542/AnsiballZ_ping.py\", line 54, in invoke_module\n      runpy.run_module(mod_name='ansible.modules.ping', init_globals=dict(_module_fqn='ansible.modules.ping', _modlib_path=modlib_path),\n  File \"/usr/lib/python3.8/runpy.py\", line 207, in run_module\n      return _run_module_code(code, init_globals, run_name, mod_spec)\n  File \"/usr/lib/python3.8/runpy.py\", line 97, in _run_module_code\n      _run_code(code, mod_globals, init_globals,\n  File \"/usr/lib/python3.8/runpy.py\", line 87, in _run_code\n      exec(code, run_globals)\n  File \"/tmp/ansible_ping_payload_9uj19ulc/ansible_ping_payload.zip/ansible/modules/ping.py\", line 83, in <module>\n  File \"/tmp/ansible_ping_payload_9uj19ulc/ansible_ping_payload.zip/ansible/modules/ping.py\", line 73, in main\nException: boom\nSee stdout/stderr for the exact error", "rc": 1}
...ignoring

TASK [ping : assert the ping failed with data=boom] *****
ok: [testhost] => {
    "changed": false,
    "msg": "All assertions passed"
}

PLAY RECAP *****
testhost : ok=7    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=1

WARNING: Reviewing previous 1 warning(s):
WARNING: A Python pip was found at "/usr/bin/pip", but it uses interpreter "/usr/bin/python3.8" instead of "/usr/bin/python2.7".
jfreeman@mastery1:~/devel/ansible$
```

Figure 10.17 – Running the Ansible ping integration test against a Python 3.8 environment

Note that there is even a test in this suite that's designed to fail – and that, in the end, we will see `ok=7` and `failed=0`, meaning all the tests passed. A large set of POSIX-compatible non-destructive integration tests run by continuous integration systems on proposed changes to Ansible can be executed with the following command:

```
ansible-test integration shippable/ --docker fedora32
```

#### Important Note

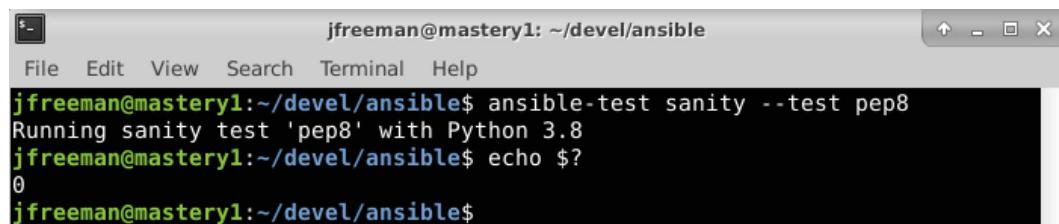
To ensure a consistent and stable testing environment, these tests are run in a local Fedora 32 container. You will need to ensure that Docker is set up and accessible on your test host for this command to work.

## Code-style tests

The third category of Ansible tests is the code-style category. These tests examine the syntax used in the Python files, ensuring a cohesive look across the code base. The code style that's enforced is defined by PEP8, a style guide for Python. More information is available here: [https://docs.ansible.com/ansible/latest/dev\\_guide/testing/sanity/pep8.html](https://docs.ansible.com/ansible/latest/dev_guide/testing/sanity/pep8.html). This style is enforced via the `pep8` sanity test target. For this test to run, you must have installed the `pycodestyle` module for Python 3. So, your commands from the root of your Ansible source directory might be as follows:

```
sudo apt install python3-pycodestyle
source ./hacking/env-setup
ansible-test sanity --test pep8
echo $?
```

If there are no errors, this target does not output any text; however, the return code can be verified. An exit code of 0 means there were no errors, as shown in the following screenshot:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/devel/ansible". The window shows the following command being run and its output:

```
jfreeman@mastery1:~/devel/ansible$ ansible-test sanity --test pep8
Running sanity test 'pep8' with Python 3.8
jfreeman@mastery1:~/devel/ansible$ echo $?
0
jfreeman@mastery1:~/devel/ansible$
```

Figure 10.18 – A successful run of the `pep8` Python code style test

**Important Note**

As you have seen already, additional Python modules might be required to run any of the Ansible tests – the method for installing them will vary from system to system, and the modules required will vary from test to test. These could typically be installed by using the pip3 tool or local operating system packages, as we have done here.

If a Python file does have a pep8 violation, the output will reflect the violation – for example, we'll deliberately edit the code for the ansible.builtin.file module, which can be found in lib/ansible/modules/file.py under the source code root. We'll introduce several deliberate errors such as blank lines with whitespace, and we'll replace some of the all-important indentation spaces with tabs, then rerun the test just as we did previously. We won't need to reinstall the Python module or set up the environment again; the output of this test will show us exactly where the errors exist, as shown in the following screenshot:

```
jfreeman@mastery1:~/devel/ansible
File Edit View Search Terminal Help
jfreeman@mastery1:~/devel/ansible$ vi lib/ansible/modules/file.py
jfreeman@mastery1:~/devel/ansible$ ansible-test sanity --test pep8
Running sanity test 'pep8' with Python 3.8
ERROR: Found 7 pep8 issue(s) which need to be resolved:
ERROR: lib/ansible/modules/file.py:11:1: W293: blank line contains whitespace
ERROR: lib/ansible/modules/file.py:278:1: E305: expected 2 blank lines after class or function definition, found 0
ERROR: lib/ansible/modules/file.py:279:1: E101: indentation contains mixed spaces and tabs
ERROR: lib/ansible/modules/file.py:279:1: W191: indentation contains tabs
ERROR: lib/ansible/modules/file.py:279:2: E117: over-indented
ERROR: lib/ansible/modules/file.py:281:1: E101: indentation contains mixed spaces and tabs
ERROR: lib/ansible/modules/file.py:291:5: E901: IndentationError: unindent does not match any outer indentation level
See documentation for help: https://docs.ansible.com/ansible/dev_guide/testing/sanity/pep8.html
ERROR: The 1 sanity test(s) listed below (out of 1) failed. See error output above for details.
pep8
jfreeman@mastery1:~/devel/ansible$ echo $?
1
jfreeman@mastery1:~/devel/ansible$
```

Figure 10.19 – Rerunning the pep8 sanity test with deliberate coding style errors introduced to the file module

pep8 errors will indicate an error code, which can be looked up for detailed explanations and guidance, along with a location and a filename, and even a line and column number to help you rapidly locate and rectify the issue.

## Making a pull request

Once all the tests have passed, a submission can be made. The Ansible project uses GitHub pull requests to manage submissions. To create a pull request, your changes must be committed and pushed to GitHub. Developers use a fork of the Ansible repository under their account to push proposed changes.

Once pushed, a pull request can be opened using the GitHub website. This will create the pull request, which will start continuous integration tests and notify reviewers of a new submission. Further information about GitHub pull requests can be found at <https://docs.github.com/en/github/collaborating-with-pull-requests>.

Once the pull request is open, reviewers will comment on the pull request, either asking for more information, suggesting changes, or approving the change. For new module submissions, you are encouraged to go down the collections route, but if you wish to explore this further, there is a great deal of valuable information available to would-be developers here: [https://docs.ansible.com/ansible/latest/dev\\_guide/index.html](https://docs.ansible.com/ansible/latest/dev_guide/index.html).

Submissions that are found acceptable and merged will be made generally available in the next release of Ansible. That concludes our look at contributing code back to the Ansible project and this chapter on extending Ansible. Hopefully, this chapter has given you some ideas and inspiration on how to build on the excellent foundation that Ansible provides to solve your automation challenges.

## Summary

Ansible is a great tool; however, sometimes, it doesn't offer all the functionality you might desire. Not every bit of functionality is appropriate to submit to the `ansible-core` project, nor is it possible to provide bespoke integration with custom proprietary data sources, as these would be different in every case. As a result, there are many facilities within Ansible to extend its functionality. Creating and using custom modules is made easy by the shared module base code. Many different types of plugins can be created and used with Ansible to affect operations in a variety of ways. Inventory sources beyond those provided with the Ansible release collections can still be used with relative ease and efficiency.

In this chapter, you learned about developing modules and including them in your playbooks. You then learned about extending Ansible through plugins, and we went into specific details about creating dynamic inventory plugins. Finally, you learned how to contribute code back to the Ansible project to enhance the code for everyone in the community. In summary, you learned that, in all cases, there are mechanisms to provide modules, plugins, and inventory sources alongside the playbooks and roles that depend on the enhanced functionality, making it seamless to distribute. This enables an almost infinite amount of expansion or customization of Ansible to your requirements, and the ability to easily contribute to the wider community if desired.

In *Chapter 12, Infrastructure Provisioning*, we will explore the use of Ansible in creating the infrastructure to be managed.

## Questions

1. For Ansible releases after 3.0, you would almost always develop a new module and distribute it via which of the following?
  - a) The `ansible-core` project.
  - b) Your collection.
  - c) An existing collection with overlapping functionality, with the project maintainer's approval.
  - d) A role.
  - e) b, c, and maybe d only
2. The easiest way to develop a custom module is to write it in what language?
  - a) Bash
  - b) Perl
  - c) Python
  - d) C++
3. Providing facts from a custom module does what?
  - a) Saves you from needing to register the output to a variable and then using `set_fact`.
  - b) Gives your code greater capabilities.
  - c) Helps you debug your code.
  - d) Shows you how the module is running.

4. Callback plugins allow you to do what?
  - a) Help you call other playbooks.
  - b) Easily alter the behavior of Ansible at key operational points without having to alter the `ansible-core` code.
  - c) Provide an efficient means of altering the code's state.
  - d) Help you to call back to your playbook during runtime.
5. To distribute plugins, where would you place them?
  - a) In a specially named directory relevant to their function (for example, callback plugins would go in the `callback_plugins/` directory).
  - b) In the Ansible installation directory.
  - c) Under `~/.ansible/plugins`.
  - d) It doesn't matter where, provided you specify them in `ansible.cfg`.
6. Dynamic inventory plugins should be written in what language?
  - a) Python.
  - b) Bash.
  - c) C++.
  - d) Any language, provided the output is returned in the correct JSON data structure.
7. Dynamic inventory plugins should parse which two command-line arguments?
  - a) `--list` and `--hostname`
  - b) `--list` and `--host`
  - c) `--list-all` and `--hosts`
  - d) `--list` and `--server`
8. Dynamic inventory performance can be improved by doing what?
  - a) Returning all host-specific data under a `_meta` key when the `--list` parameter is passed.
  - b) Returning all host-specific data, regardless of which arguments are passed.
  - c) Caching the output of the script's run.
  - d) Compressing your output data to reduce transmission times.

9. If you wish to contribute code to the `ansible-core` project, you should submit it via which of the following methods?
  - a) A ticket raised against the project detailing your changes
  - b) Submitting a support ticket to Red Hat
  - c) A GitHub pull request once your code is passing all included tests
  - d) Complaining loudly on Twitter
10. Which utility is used to launch and control most of the Ansible code tests?
  - a) `test-runner`
  - b) `integration-test`
  - c) Jenkins
  - d) `ansible-test`

# Section 3: Orchestration with Ansible

In this section, we will understand the real-world usage of Ansible to coordinate and manage systems and services, whether on-premise or in the cloud.

The following chapters are included in this section:

- *Chapter 11, Minimizing Downtime with Rolling Deployments*
- *Chapter 12, Infrastructure Provisioning*
- *Chapter 13, Network Automation*



# 11

# Minimizing Downtime with Rolling Deployments

Ansible is well suited to the task of upgrading or deploying applications in a live service environment. Of course, application deployments and upgrades can be approached with a variety of different strategies. The best approach depends on the application itself, the capabilities of the infrastructure the application runs on, and any promised **service-level agreements (SLAs)** with the users of the application. Whatever the approach, it is vital that the application deployment or upgrade is controlled, predictable, and repeatable in order to ensure that users experience a stable service while automated deployments occur in the background. The last thing anyone wants is an outage caused by unexpected behavior from their automation tool; an automation tool should be trustworthy and not an additional risk factor.

Although there is a myriad of choices, some deployment strategies are more common than others, and in this chapter, we'll walk through a couple of the more common ones. In doing so, we will showcase the Ansible features that will be useful within those strategies. We'll also discuss a couple of other deployment considerations that are common across both deployment strategies. To achieve this, we will delve into the details of the following subjects, in the context of a rolling Ansible deployment:

- In-place upgrades
- Expanding and contracting
- Failing fast
- Minimizing disruptions
- Serializing single tasks

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running **Ansible 4.3** or a newer version. Almost any flavor of Linux should do—for those interested in specifics, all the code presented in this chapter was tested on **Ubuntu Server 20.04 Long-Term Support (LTS)** unless stated otherwise, and on Ansible 4.3. The example code that accompanies this chapter can be downloaded from GitHub at <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter11>.

Check out the following video to see the code in action: <https://bit.ly/3lZ6Y9W>

## In-place upgrades

The first type of deployment that we'll cover is in-place upgrades. This style of deployment operates on an infrastructure that already exists, in order to upgrade the existing application. This model is a traditional model that was used when the creation of new infrastructure was a costly endeavor, in terms of both time and money.

A general design pattern to minimize the downtime during this type of upgrade is to deploy the application across multiple hosts, behind a load balancer. The load balancer will act as a gateway between users of the application and the servers that run the application. Requests for the application will come to the load balancer, and, depending on the configuration, the load balancer will decide which backend server to direct the requests to.

To perform a rolling in-place upgrade of an application deployed with this pattern, each server (or a small subset of the servers) will be disabled at the load balancer, upgraded, and then re-enabled to take on new requests. This process will be repeated for the remaining servers in the pool until all servers have been upgraded. As only a portion of the available application servers is taken offline to be upgraded, the application as a whole remains available for requests. Of course, this assumes that an application can perform well with mixed versions running at the same time.

Let's build a playbook to upgrade a fictional application. Our fictional application will run on servers `foo-app01` through `foo-app08`, which exist in the `foo-app` group. These servers will have a simple website that's served via the `nginx` web server, with the content coming from a `foo-app` Git repository, defined by the `foo-app.repo` variable. A load-balancer server, `foo-1b`, running the `haproxy` software, will front these app servers.

In order to operate on a subset of our `foo-app` servers, we need to employ the `serial` mode. This mode changes how Ansible will execute a play. By default, Ansible will execute the tasks of a play across each host in the order that the tasks are listed. Ansible executes each task of the play on every host before it moves on to the next task in the play. If we were to use the default method, our first task would remove every server from the load balancer, which would result in the complete outage of our application. Instead, the `serial` mode lets us operate on a subset so that the application as a whole stays available, even if some of the members are offline. In our example, we'll use a serial count of 2 in order to keep the majority of the application members online:

```
---  
- name: Upgrade foo-app in place  
  hosts: foo-app  
  serial: 2
```

### Important Note

Ansible 2.2 introduced the concept of serial batches: a list of numbers that can increase the number of hosts addressed serially each time through the play. This allows the size of the hosts addressed to increase as confidence increases. Where a batch of numbers is provided to the `serial` keyword, the last number provided will be the size of any remaining batch, until all hosts in the inventory have been completed.

Now, we can start to create our tasks. The first task will be to disable the host from the load balancer. The load balancer runs on the `foo-1b` host; however, we're operating on the `foo-app` hosts. Therefore, we need to delegate the task by using the `delegate_to` task operator. This operator redirects where Ansible will connect to in order to execute the task, but it keeps all of the variable contexts of the original host. We'll use the `community.general.haproxy` module to disable the current host from the `foo-app` backend pool. The code is illustrated in the following snippet:

```
tasks:
  - name: disable member in balancer
    community.general.haproxy:
      backend: foo-app
      host: "{{ inventory_hostname }}"
      state: disabled
      delegate_to: foo-1b
```

With the host disabled, we can now update the `foo-app` content. We'll use the `ansible.builtin.git` module to update the content path with the desired version, defined as `foo-version`. We'll add a `notify` handler to this task to reload the `nginx` server if the content update results in a change. This restart can be done every time, but we're also using this as an example usage of `notify`. You can view the code in the following snippet:

```
- name: pull stable foo-app
  ansible.builtin.git:
    repo: "{{ foo-app.repo }}"
    dest: /srv/foo-app/
    version: "{{ foo-version }}"
    notify:
      - reload nginx
```

Our next step would be to re-enable the host in the load balancer; however, if we did that task next, we'd put the old version back in place, as our notified handler hasn't run yet. So, we need to trigger our handlers early, by way of the `meta: flush_handlers` call, which you learned about in *Chapter 10, Extending Ansible*. You can see this again here:

```
- meta: flush_handlers
```

Now, we can re-enable the host in the load balancer. We can just enable it right away and rely on the load balancer to wait until the host is healthy before sending requests to it. However, because we are running with a reduced number of available hosts, we need to ensure that all of the remaining hosts are healthy. We can make use of an `ansible.builtin.wait_for` task to wait until the `nginx` service is once again serving connections. The `ansible.builtin.wait_for` module will wait for a condition on either a port or a file path. In the following example, we will wait for port 80 and the condition that the port should be in. If it is started (the default), that means it is accepting connections:

```
- name: ensure healthy service
  ansible.builtin.wait_for:
    port: 80
```

Finally, we can re-enable the member within haproxy. Once again, we'll delegate the task to `foo-lb`, as illustrated in the following code snippet:

```
- name: enable member in balancer
  community.general.haproxy:
    backend: foo-app
    host: "{{ inventory_hostname }}"
    state: enabled
  delegate_to: foo-lb
```

Of course, we still need to define our `reload nginx` handler. We can do this by running the following code:

```
handlers:
- name: reload nginx
  ansible.builtin.service:
    name: nginx
    state: restarted
```

This playbook, when run, will now perform a rolling in-place upgrade of our application. Of course, it's not always desirable to run an upgrade in place—there's always the chance that this could be service-impacting, especially if the service comes under unexpected load. An alternate strategy that prevents this, expanding and contracting, is explored in the next section.

## Expanding and contracting

An alternative to the in-place upgrade strategy is the **expand and contract** strategy. This strategy has become popular of late, thanks to the self-service nature of on-demand infrastructures, such as cloud computing or virtualization pools. The ability to create new servers on-demand from a large pool of available resources means that every deployment of an application can happen on brand new systems. This strategy avoids a host of issues, such as a buildup of cruft on long-running systems, such as the following:

- Configuration files that are no longer managed by Ansible being left behind
- Runaway processes consuming resources in the background
- Changes being made to the server manually by human beings without updating the Ansible playbooks

Starting fresh each time also removes the differences between initial deployment and an upgrade. The same code path can be used, reducing the risk of surprises when upgrading an application. This type of installation can also make it extremely easy to roll back if the new version does not perform as expected. In addition to this, as new systems are created to replace old systems, the application does not need to go into a degraded state during the upgrade.

Let's re-approach our previous upgrade playbook with the expand and contract strategy. Our pattern will be to create new servers, deploy our application, verify our application, add new servers to the load balancer, and remove old servers from the load balancer. Let's start by creating new servers. For this example, we'll make use of an OpenStack compute cloud to launch new instances:

```
---  
- name: Create new foo servers  
  hosts: localhost  
  
  tasks:  
    - name: launch instances  
      openstack.cloud.os_server:  
        name: foo-appv{{ version }}-{{ item }}  
        image: foo-appv{{ version }}  
        flavor: 4
```

```
key_name: ansible-prod
security_groups: foo-app
auto_floating_ip: false
state: present
auth:
  auth_url: https://me.openstack.blueboxgrid.com:5001/
v2.0
  username: jlk
  password: FAKEPASSWORD
  project_name: mastery
register: launch
loop: "{{ range(1, 8 + 1, 1)|list }}"
```

In this task, we're looping over a count of 8, using the new `loop` with `range` syntax that was introduced in Ansible 2.5. For each iteration of the loop, the `item` variable will be replaced by a number. This allows us to create eight new server instances with names based on the version of our application and the number of the loop. We're also assuming a prebuilt image to us so that we do not need to do any further configuration on the instance. In order to use the servers in future plays, we need to add their details to the inventory. To accomplish this, we register the results of the run in the `launch` variable, which we'll use to create runtime inventory entries. The code is illustrated in the following snippet:

```
- name: add hosts
  ansible.builtin.add_host:
    name: "{{ item.openstack.name }}"
    ansible_ssh_host: "{{ item.openstack.private_v4 }}"
    groups: new-foo-app
  loop: launch.results
```

This task will create new inventory items with the same names as those of our server instance. To help Ansible know how to connect, we'll set `ansible_ssh_host` to the **Internet Protocol (IP)** address that our cloud provider assigned to the instance (this is assuming that the address is reachable by the host running Ansible). Finally, we'll add the hosts to the `new-foo-app` group. As our `launch` variable comes from a task with a `loop`, we need to iterate over the results of that loop by accessing the `results` key. This allows us to loop over each `launch` action to access the data specific to that task.

Next, we'll operate on the servers to ensure that the new service is ready for use. We'll use `ansible.builtin.wait_for` again, just as we did earlier, as a part of a new play operating on our `new-foo-app` group. The code is illustrated in the following snippet:

```
- name: Ensure new app
hosts: new-foo-app
tasks:
  - name: ensure healthy service
    ansible.builtin.wait_for:
      port: 80
```

Once they're all ready to go, we can reconfigure the load balancer to make use of our new servers. For the sake of simplicity, we will assume a template for the haproxy configuration that expects hosts in a `new-foo-app` group, and the end result will be a configuration that knows all about our new hosts and forgets about our old hosts. This means that we can simply call an `ansible.builtin.template` task on the load-balancer system itself, rather than attempting to manipulate the running state of the balancer. The code is illustrated in the following snippet:

```
- name: Configure load balancer
hosts: foo-lb
tasks:
  - name: haproxy config
    ansible.builtin.template:
      dest: /etc/haproxy/haproxy.cfg
      src: templates/etc/haproxy/haproxy.cfg

  - name: reload haproxy
    ansible.builtin.service:
      name: haproxy
      state: reloaded
```

Once the new configuration file is in place, we can issue a reload of the `haproxy` service. This will parse the new configuration file and start a new listening process for new incoming connections. The existing connections will eventually close, and the old processes will terminate. All new connections will be routed to the new servers running our new application version.

This playbook can be extended to decommission the old version of the servers, or that action may happen at a different time when it has been decided that a rollback to the old-version capability is no longer necessary.

The expand and contract strategy can involve more tasks, and even separate playbooks for creating a golden image set, but the benefits of a fresh infrastructure for every release far outweigh the extra tasks or added complexity of creation followed by deletion.

## Failing fast

When performing an upgrade of an application, it may be desirable to fully stop the deployment at any sign of an error. A partially upgraded system with mixed versions may not work at all, so continuing with part of the infrastructure while leaving the failed systems behind can lead to big problems. Fortunately, Ansible provides a mechanism to decide when to reach a fatal-error scenario.

By default, when Ansible is running through a playbook and encounters an error, it will remove the failed host from the list of play hosts and continue with the tasks or plays. Ansible will stop executing either when all the requested hosts for a play have failed or when all the plays have been completed. To change this behavior, there are a couple of play controls that can be employed. Those controls are `any_errors_fatal`, `max_fail_percentage`, and `force_handlers`, and these are discussed next.

### The `any_errors_fatal` option

This setting instructs Ansible to consider the entire operation fatal and to stop executing immediately if any host encounters an error. To demonstrate this, we'll edit our `mastery-hosts` inventory, defining a pattern that will expand up to 10 new hosts, as illustrated in the following code snippet:

```
[failtest]
failer[01:10]
```

Then, we'll create a play on this group with `any_errors_fatal` set to `true`. We'll also turn off fact-gathering since these hosts do not exist. The code is illustrated in the following snippet:

```
---
- name: any errors fatal
  hosts: failtest
  gather_facts: false
  any_errors_fatal: true
```

We want a task that will fail for one of the hosts but not the others. Then, we'll want a second task as well, just to demonstrate how it will not run. Here's the code we need to execute:

```
tasks:
- name: fail last host
  ansible.builtin.fail:
    msg: "I am last"
    when: inventory_hostname == play_hosts[-1]
- name: never run
  ansible.builtin.debug:
    msg: "I should never be run"
    when: inventory_hostname == play_hosts[-1]
```

We now execute the playbook using the following command:

```
ansible-playbook -i mastery-hosts failtest.yaml
```

When we do this, we'll see one host fail, but the entire play will stop after the first task, and the `ansible.builtin.debug` task is never attempted, as illustrated in the following screenshot:

```
jfreeman@mastery1: ~/examples/Chapter11/example03
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter11/example03$ ansible-playbook -i mastery-hosts failtest.yaml

PLAY [any errors fatal] ****
TASK [fail last host] ****
skipping: [failer01]
skipping: [failer02]
skipping: [failer03]
skipping: [failer04]
skipping: [failer05]
skipping: [failer06]
skipping: [failer07]
skipping: [failer08]
skipping: [failer09]
fatal: [failer10]: FAILED! => {"changed": false, "msg": "I am last"}

NO MORE HOSTS LEFT ****

PLAY RECAP ****
failer01          : ok=0    changed=0      unreachable=0    failed=0    s
kipped=1  rescued=0  ignored=0
failer02          : ok=0    changed=0      unreachable=0    failed=0    s
kipped=1  rescued=0  ignored=0
failer03          : ok=0    changed=0      unreachable=0    failed=0    s
kipped=1  rescued=0  ignored=0
failer04          : ok=0    changed=0      unreachable=0    failed=0    s
kipped=1  rescued=0  ignored=0
failer05          : ok=0    changed=0      unreachable=0    failed=0    s
kipped=1  rescued=0  ignored=0
failer06          : ok=0    changed=0      unreachable=0    failed=0    s
kipped=1  rescued=0  ignored=0
failer07          : ok=0    changed=0      unreachable=0    failed=0    s
kipped=1  rescued=0  ignored=0
failer08          : ok=0    changed=0      unreachable=0    failed=0    s
kipped=1  rescued=0  ignored=0
failer09          : ok=0    changed=0      unreachable=0    failed=0    s
kipped=1  rescued=0  ignored=0
failer10          : ok=0    changed=0      unreachable=0    failed=1    s
kipped=0  rescued=0  ignored=0
```

Figure 11.1 – Failing an entire playbook early when just one host in the inventory fails

We can see that just one host failed; however, Ansible reported NO MORE HOSTS LEFT (implying that all hosts failed) and aborted the playbook before getting to the next play.

## The max\_fail\_percentage option

This setting allows play developers to define a percentage of hosts that can fail before the whole operation is aborted. At the end of each task, Ansible will perform a calculation to determine the number of hosts targeted by the play that have reached a failure state, and if that number is greater than the number allowed, Ansible will abort the playbook. This is similar to any\_errors\_fatal; in fact, any\_errors\_fatal just internally expresses a max\_fail\_percentage parameter of 0, where any failure is considered fatal. Let's edit our play from the preceding section and remove any\_errors\_fatal, replacing it with the max\_fail\_percentage parameter set to 20, as follows:

```
---
- name: any errors fatal
  hosts: failtest
  gather_facts: false
  max_fail_percentage: 20
```

By making that change and running our playbook with the same command as we used previously, our play should complete both tasks without aborting, as the following screenshot shows:

```
jfreeman@mastery1: ~/examples/Chapter11/example04
File Edit View Search Terminal Help
skipping: [failer06]
skipping: [failer07]
skipping: [failer08]
skipping: [failer09]
fatal: [failer10]: FAILED! => {"changed": false, "msg": "I am last"}

TASK [never run] *****
skipping: [failer01]
skipping: [failer02]
skipping: [failer03]
skipping: [failer04]
skipping: [failer05]
skipping: [failer06]
skipping: [failer07]
skipping: [failer08]
ok: [failer09] => {
    "msg": "I should never be run"
}

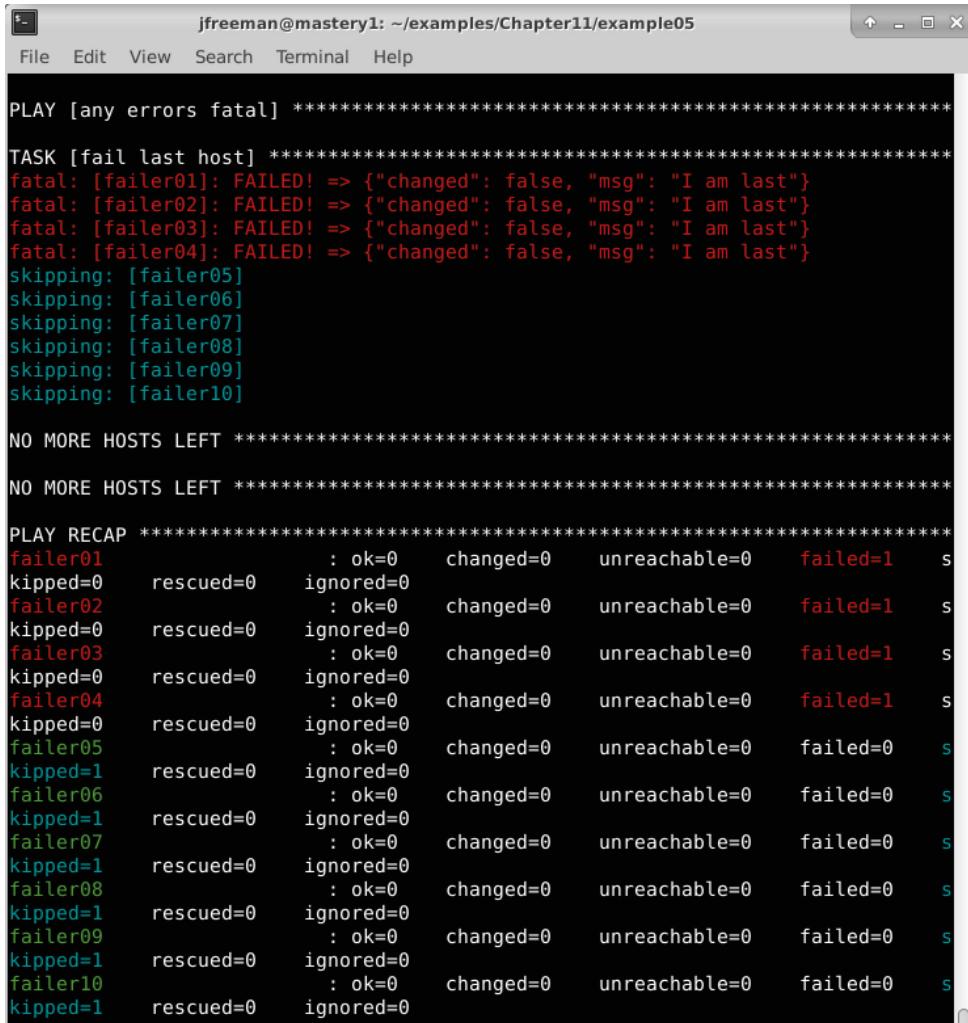
PLAY RECAP *****
failer01          : ok=0      changed=0      unreachable=0      failed=0      S
kipped=2  rescued=0  ignored=0
failer02          : ok=0      changed=0      unreachable=0      failed=0      S
kipped=2  rescued=0  ignored=0
failer03          : ok=0      changed=0      unreachable=0      failed=0      S
kipped=2  rescued=0  ignored=0
failer04          : ok=0      changed=0      unreachable=0      failed=0      S
kipped=2  rescued=0  ignored=0
failer05          : ok=0      changed=0      unreachable=0      failed=0      S
kipped=2  rescued=0  ignored=0
failer06          : ok=0      changed=0      unreachable=0      failed=0      S
kipped=2  rescued=0  ignored=0
failer07          : ok=0      changed=0      unreachable=0      failed=0      S
kipped=2  rescued=0  ignored=0
failer08          : ok=0      changed=0      unreachable=0      failed=0      S
kipped=2  rescued=0  ignored=0
failer09          : ok=1      changed=0      unreachable=0      failed=0      S
kipped=1  rescued=0  ignored=0
failer10          : ok=0      changed=0      unreachable=0      failed=1      S
kipped=0  rescued=0  ignored=0
```

Figure 11.2 – Demonstrating our previous failure-test playbook proceeding with fewer than 20 percent failed hosts

Now, if we change the condition on our first task so that we deliberately fail on over 20 percent of the hosts, we'll see the playbook abort early:

```
- name: fail last host
  ansible.builtin.fail:
    msg: "I am last"
  when: inventory_hostname in play_hosts[0:3]
```

We're deliberately setting up three hosts to fail, which will give us a failure rate of greater than 20 percent. The `max_fail_percentage` setting is the maximum allowed, so our setting of 20 would allow two out of the ten hosts to fail. With three hosts failing, we will see a fatal error before the second task is allowed to execute, as the following screenshot illustrates:



```
jfreeman@mastery1: ~/examples/Chapter11/example05
File Edit View Search Terminal Help

PLAY [any errors fatal] ****
TASK [fail last host] ****
fatal: [failer01]: FAILED! => {"changed": false, "msg": "I am last"}
fatal: [failer02]: FAILED! => {"changed": false, "msg": "I am last"}
fatal: [failer03]: FAILED! => {"changed": false, "msg": "I am last"}
fatal: [failer04]: FAILED! => {"changed": false, "msg": "I am last"}
skipping: [failer05]
skipping: [failer06]
skipping: [failer07]
skipping: [failer08]
skipping: [failer09]
skipping: [failer10]

NO MORE HOSTS LEFT ****
NO MORE HOSTS LEFT ****

PLAY RECAP ****
failer01          : ok=0    changed=0    unreachable=0   failed=1    s
kipped=0  rescued=0 ignored=0
failer02          : ok=0    changed=0    unreachable=0   failed=1    s
kipped=0  rescued=0 ignored=0
failer03          : ok=0    changed=0    unreachable=0   failed=1    s
kipped=0  rescued=0 ignored=0
failer04          : ok=0    changed=0    unreachable=0   failed=1    s
kipped=0  rescued=0 ignored=0
failer05          : ok=0    changed=0    unreachable=0   failed=0    s
kipped=1  rescued=0 ignored=0
failer06          : ok=0    changed=0    unreachable=0   failed=0    s
kipped=1  rescued=0 ignored=0
failer07          : ok=0    changed=0    unreachable=0   failed=0    s
kipped=1  rescued=0 ignored=0
failer08          : ok=0    changed=0    unreachable=0   failed=0    s
kipped=1  rescued=0 ignored=0
failer09          : ok=0    changed=0    unreachable=0   failed=0    s
kipped=1  rescued=0 ignored=0
failer10          : ok=0    changed=0    unreachable=0   failed=0    s
kipped=1  rescued=0 ignored=0
```

Figure 11.3 – Demonstrating the `max_fail_percentage` operation failing a play when the percentage is exceeded

With this combination of parameters, we can easily set up and control **fail-fast** conditions on a group of hosts, which is incredibly valuable if our goal is to maintain the integrity of an environment during an Ansible deployment.

## Forcing handlers

Normally, when Ansible fails a host, it stops executing anything on that host. This means that any pending handlers will not be run. This can be undesirable, and there is a play control that will force Ansible to process pending handlers for failed hosts. This play control is `force_handlers`, which must be set to the `true` Boolean value.

Let's modify our preceding example a little in order to demonstrate this functionality. We'll remove our `max_fail_percentage` parameter and add a new first task. We need to create a task that will return successfully with a change. This is possible with the `ansible.builtin.debug` module, using the `changed_when` task control, as this module will never register a change otherwise. We'll revert our `ansible.builtin.fail` task conditional to our original one, as well. The code is illustrated in the following snippet:

```
---
```

```
- name: any errors fatal
  hosts: failtest
  gather_facts: false
```

```

tasks:
  - name: run first
    ansible.builtin.debug:
      msg: "I am a change"
      changed_when: true
      when: inventory_hostname == play_hosts[-1]
      notify: critical handler
  - name: change a host
    ansible.builtin.fail:
      msg: "I am last"
      when: inventory_hostname == play_hosts[-1]
```

Our third task remains unchanged, but we will define our critical handler, as follows:

```
- name: never run
  ansible.builtin.debug:
    msg: "I should never be run"
    when: inventory_hostname == play_hosts[-1]
```

```

handlers:
  - name: critical handler
```

```
ansible.builtin.debug:
  msg: "I really need to run"
```

Let's run this new play to show the default behavior of the handler not being executed. In the interest of reduced output, we'll limit execution to just one of the hosts with the following command:

```
ansible-playbook -i mastery-hosts failtest.yaml --limit
failer01:failer01
```

Note that although the handler is referenced in the play output, it is not actually run, as evidenced by the lack of any debug message, which the following screenshot clearly shows:

```
jfreeman@mastery1:~/examples/Chapter11/example06$ ansible-playbook -i mastery-hosts failtest.yaml --limit failer01:failer01

PLAY [any errors fatal] ****
TASK [run first] ****
changed: [failer01] => {
    "msg": "I am a change"
}

TASK [change a host] ****
fatal: [failer01]: FAILED! => {"changed": false, "msg": "I am last"}

RUNNING HANDLER [critical handler] ****

PLAY RECAP ****
failer01                  : ok=1      changed=1      unreachable=0      failed=1      s
kipped=0      rescued=0      ignored=0
```

Figure 11.4 – Demonstrating how handlers are not run even when notified if a play fails

Now, we add the `force_handlers` play control and set it to `true`, as follows:

```
---
- name: any errors fatal
  hosts: failtest
  gather_facts: false
  force_handlers: true
```

This time, when we run the playbook (using the same command as before), we should see the handler run even for the failed hosts, as demonstrated in the following screenshot:

```
jfreeman@mastery1:~/examples/Chapter11/example07$ ansible-playbook -i mastery-hosts failtest.yaml --limit failer01:failer01

PLAY [any errors fatal] ****
TASK [run first] ****
changed: [failer01] => {
    "msg": "I am a change"
}

TASK [change a host] ****
fatal: [failer01]: FAILED! => {"changed": false, "msg": "I am last"}

RUNNING HANDLER [critical handler] ****
ok: [failer01] => {
    "msg": "I really need to run"
}

PLAY RECAP ****
failer01 : ok=2    changed=1    unreachable=0    failed=1    s
kipped=0    rescued=0    ignored=0
```

Figure 11.5 – Demonstrating that handlers can be forced to run, even for failed hosts in a failed play

#### Important Note

Forcing handlers can be a runtime decision as well, using the `--force-handlers` command-line argument on `ansible-playbook`. It can also be set globally, as a parameter in `ansible.cfg`.

Forcing handlers to run can be really useful for repeated playbook runs. The first run may result in some changes, but if a fatal error is encountered before the handlers are flushed, those handler calls will be lost. Repeated runs will not result in the same changes, so the handler will never run without manual interaction. Forcing handlers ensures that those handler calls are not lost, and so the handlers are always run regardless of the task outcomes. Of course, the whole objective of any upgrade strategy is to keep the impact on any given service as low as possible—can you imagine your favorite retail site going down for someone to upgrade software? It is unthinkable in this day and age! In the next section, we explore ways to minimize potentially disruptive actions using Ansible.

## Minimizing disruptions

During deployment, there are often tasks that can be considered disruptive or destructive. These tasks may include restarting services, performing database migrations, and so on. Disruptive tasks should be clustered together to minimize the overall impact on an application, while destructive tasks should only be performed once. The next two subsections explore how you can meet both these targets using Ansible.

### Delaying a disruption

Restarting services for a new configuration or code version is a very common requirement. When viewed in isolation, a single service can be restarted whenever the code and configuration for the application have changed, without concern for the overall distributed system health. Typically, a distributed system will have roles for each part of the system, and each role will essentially operate in isolation on the hosts targeted to perform those roles. When deploying an application for the first time, there is no existing uptime of the whole system to worry about, so services can be restarted at will. However, during an upgrade, it may be desirable to delay all service restarts until every service is ready, to minimize interruptions.

The reuse of role code is strongly encouraged, as opposed to designing a completely separate upgrade code path. To accommodate a coordinated reboot, the role code for a particular service needs protection around the service restart. A common pattern is to put a conditional statement on the disruptive tasks that check a variable's value. When performing an upgrade, the variable can be defined at runtime to trigger this alternative behavior. This variable can also trigger a coordinated restart of services at the end of the main playbook once all of the roles have been completed, in order to cluster the disruption and minimize the total outage.

Let's create a fictional application upgrade that involves two roles with simulated service restarts. We'll call these roles `microA` and `microB`. The code is illustrated in the following snippet:

```
roles/microA
|   └── handlers
|       └── main.yaml
|   └── tasks
|       └── main.yaml
roles/microB
|   └── handlers
|       └── main.yaml
```

```

└── tasks
    └── main.yaml

```

For both roles, we'll have a simple debug task that simulates the installation of a package. We'll notify a handler to simulate the restart of a service, and to ensure that the handler will trigger, we'll force the task to always register as changed. The following code snippet shows the content of `roles/microA/tasks/main.yaml`:

```

---
- name: install microA package
  ansible.builtin.debug:
    msg: "This is installing A"
  changed_when: true
  notify: restart microA

```

The content of `roles/microB/tasks/main.yaml` is shown here:

```

---
- name: install microB package
  ansible.builtin.debug:
    msg: "This is installing B"
  changed_when: true
  notify: restart microB

```

The handlers for these roles will be debug actions as well, and we'll attach a conditional statement to the handler task to only restart if the `upgrade` variable evaluates to the `false` Boolean value. We'll also use the `default` filter to give this variable a default value of `false`. The content of `roles/microA/handlers/main.yaml` is shown here:

```

---
- name: restart microA
  ansible.builtin.debug:
    msg: "microA is restarting"
  when: not upgrade | default(false) | bool

```

The content of `roles/microB/handlers/main.yaml` is shown here:

```

---
- name: restart microB
  ansible.builtin.debug:

```

```
msg: "microB is restarting"
when: not upgrade | default(false) | bool
```

For our top-level playbook, we'll create four plays (remember that a playbook can consist of one or more plays). The first two plays will apply each of the micro roles, and the last two plays will do the restarts. The last two plays will only be executed if performing an upgrade; so, they will make use of the upgrade variable as a condition. Let's take a look at the following code snippet (called `micro.yaml`):

```
---
- name: apply microA
  hosts: localhost
  gather_facts: false

  roles:
    - role: microA

- name: apply microB
  hosts: localhost
  gather_facts: false

  roles:
    - role: microB

- name: restart microA
  hosts: localhost
  gather_facts: false

  tasks:
    - name: restart microA for upgrade
      ansible.builtin.debug:
        msg: "microA is restarting"
      when: upgrade | default(false) | bool

- name: restart microB
  hosts: localhost
  gather_facts: false
```

```

tasks:
  - name: restart microB for upgrade
    ansible.builtin.debug:
      msg: "microB is restarting"
    when: upgrade | default(false) | bool

```

We execute this playbook without defining the upgrade variable, using the following command:

```
ansible-playbook -i mastery-hosts micro.yaml
```

When we do this, we will see the execution of each role, and the handlers within. The final two plays will have skipped tasks, as the following screenshot shows:

```

jfreeman@mastery1:~/examples/Chapter11/example08$ ansible-playbook -i mastery-hosts micro.yaml

PLAY [apply microA] ****
TASK [microA : install microA package] ****
changed: [localhost] => {
    "msg": "This is installing A"
}

RUNNING HANDLER [microA : restart microA] ****
ok: [localhost] => {
    "msg": "microA is restarting"
}

PLAY [apply microB] ****
TASK [microB : install microB package] ****
changed: [localhost] => {
    "msg": "This is installing B"
}

RUNNING HANDLER [microB : restart microB] ****
ok: [localhost] => {
    "msg": "microB is restarting"
}

PLAY [restart microA] ****
TASK [restart microA for upgrade] ****
skipping: [localhost]

PLAY [restart microB] ****
TASK [restart microB for upgrade] ****
skipping: [localhost]

PLAY RECAP ****
localhost          : ok=4    changed=2    unreachable=0    failed=0    s
kiped=2    rescued=0    ignored=0

```

Figure 11.6 – Demonstrating a role-based playbook for installing a microservice architecture

Now, let's execute the playbook again; this time, we'll define the upgrade variable as true at runtime, using the -e flag as follows:

```
ansible-playbook -i mastery-hosts micro.yaml -e upgrade=true
```

This time, the results should look like this:

```
jfreeman@mastery1: ~/examples/Chapter11/example08
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter11/example08$ ansible-playbook -i mastery-hosts micro.yaml -e upgrade=true

PLAY [apply microA] ****
TASK [microA : install microA package] ****
changed: [localhost] => {
    "msg": "This is installing A"
}

RUNNING HANDLER [microA : restart microA] ****
skipping: [localhost]

PLAY [apply microB] ****
TASK [microB : install microB package] ****
changed: [localhost] => {
    "msg": "This is installing B"
}

RUNNING HANDLER [microB : restart microB] ****
skipping: [localhost]

PLAY [restart microA] ****
TASK [restart microA for upgrade] ****
ok: [localhost] => {
    "msg": "microA is restarting"
}

PLAY [restart microB] ****
TASK [restart microB for upgrade] ****
ok: [localhost] => {
    "msg": "microB is restarting"
}

PLAY RECAP ****
localhost                  : ok=4    changed=2    unreachable=0    failed=0    s
kiped=2      rescued=0    ignored=0
```

Figure 11.7 – Demonstrating the same playbook, but in an upgrade scenario  
with all restarts batched at the end

This time, we can see that our handlers are skipped, but the final two plays have tasks that execute. In a real-world scenario, where many more things are happening in the `microA` and `microB` roles (and, potentially, other microservice roles on other hosts), the difference could be of many minutes or more. Clustering the restarts at the end can reduce the interruption period significantly.

## Running destructive tasks only once

Destructive tasks come in many flavors. They can be one-way tasks that are extremely difficult to roll back, one-time tasks that cannot be rerun easily, or race-condition tasks that, if performed in parallel, would result in catastrophic failure. For these reasons and more, it is essential that these tasks be performed only once, from a single host. Ansible provides a mechanism to accomplish this by way of the `run_once` task control.

The `run_once` task control will ensure that the task only executes a single time from a single host, regardless of how many hosts happen to be in a play. While there are other methods to accomplish this goal, such as using a conditional statement to make the task execute only on the first host of a play, the `run_once` control is the most simple and direct way to express this wish. Additionally, any variable data registered from a task controlled by `run_once` will be made available to all hosts of the play, not just the host that was selected by Ansible to perform the action. This can simplify later retrieval of the variable data.

Let's create an example playbook to demonstrate this functionality. We'll reuse our `failtest` hosts that were created in an earlier example, in order to have a pool of hosts, and we'll select two of them by using a host pattern. We'll create an `ansible.builtin.debug` task set to `run_once` and register the results, then we'll access the results in a different task with a different host. The code is as follows:

```
---
- name: run once test
  hosts: failtest[0:1]
  gather_facts: false

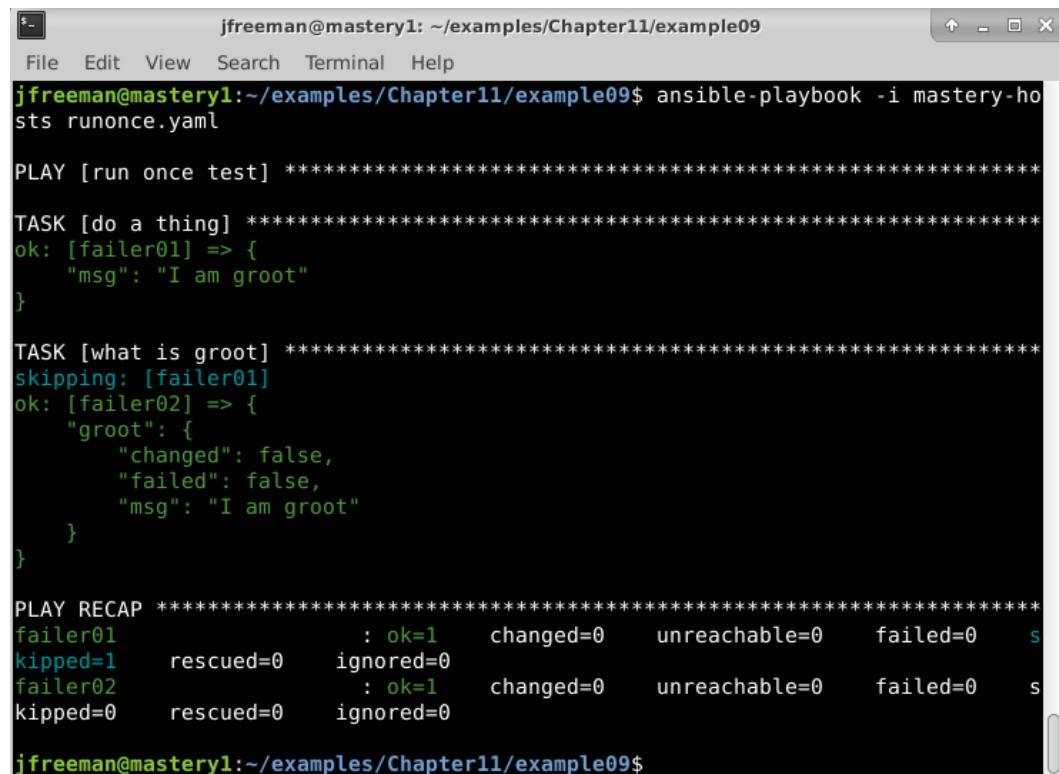
  tasks:
    - name: do a thing
      ansible.builtin.debug:
        msg: "I am groot"
      register: groot
      run_once: true
```

```
- name: what is groot
  ansible.builtin.debug:
    var: groot
    when: inventory_hostname == play_hosts[-1]
```

We run this play with the following command:

```
ansible-playbook -i mastery-hosts runonce.yaml
```

When we do this, we'll pay special attention to the hostnames listed for each task operation shown in the following screenshot:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter11/example09". The terminal displays the output of an Ansible playbook named "runonce.yaml". The playbook contains two tasks: one that fails and one that succeeds. The failed task outputs "I am groot". The successful task outputs "I am groot" and includes variable data from the failed task. The final PLAY RECAP shows the results for both hosts.

```
jfreeman@mastery1:~/examples/Chapter11/example09$ ansible-playbook -i mastery-hosts runonce.yaml

PLAY [run once test] ****

TASK [do a thing] ****
ok: [failer01] => {
    "msg": "I am groot"
}

TASK [what is groot] ****
skipping: [failer01]
ok: [failer02] => {
    "groot": {
        "changed": false,
        "failed": false,
        "msg": "I am groot"
    }
}

PLAY RECAP ****
failer01              : ok=1      changed=0      unreachable=0      failed=0      s
kipper=1   rescued=0   ignored=0
failer02              : ok=1      changed=0      unreachable=0      failed=0      s
kipped=0   rescued=0   ignored=0

jfreeman@mastery1:~/examples/Chapter11/example09$
```

Figure 11.8 – Demonstrating the use of the `run_once` task parameter, and the availability of variable data from that task on other hosts in the play

We can see that the `do_a_thing` task is executed on the `failer01` host, while the `what_is_groot` task, which examines the data from the `do_a_thing` task, operates on the `failer02` host. Of course, while you can reduce the risk of disruption to your production services using the techniques we have discussed here, there is even more we can do, such as limiting the number of times a task is run or the number of hosts it is run against. We will explore this very topic in the next section of this chapter.

## Serializing single tasks

Certain applications that run multiple copies of a service may not react well to all of those services being restarted at once. Typically, when upgrading this type of application, a `serial` play is used. However, if the application is of a large enough scale, serializing the entire play may be wildly inefficient. A different approach can be used, which is to serialize only the sensitive tasks (often the handlers to restart services).

To serialize a specific handler task, we can make use of a built-in variable, `play_hosts`. This variable holds a list of hosts that should be used for a given task as a part of the play. It is kept up to date with hosts that have failed or are unreachable. Using this variable, we can construct a loop to iterate over each host that could potentially run a handler task. Instead of using the `item` value in the module arguments, we'll use the `item` value in a `when` conditional and a `delegate_to` directive. In this manner, handler tasks that get notified within the playbook can be delegated to a host in the aforementioned loop, rather than the original host. However, if we just use this as the list for a `loop` directive, we'll end up executing the task for every host that triggers a handler. That's obviously unwanted, so we can use a task directive, `run_once`, to change the behavior. The `run_once` directive instructs Ansible to only execute the task for one host, instead of for every host that it would normally target. Combining `run_once` and our loop of `play_hosts` creates a scenario where Ansible will run through the loop only once. Finally, we want to wait a small amount of time between each loop so that the restarted service can become functional before we restart the next one. We can make use of a `loop_control` parameter called `pause` (introduced in Ansible version 2.2) to insert a pause between each iteration of the loop.

To demonstrate how this serialization will work, we'll write a play using a few hosts from our `failtest` group, with a task that creates a change and registers the output so that we can check this output in the handler task we notify, called `restart groot`. We then create a serialized handler task itself at the bottom of the playbook. The code is illustrated as follows:

```
---
- name: parallel and serial
  hosts: failtest[0:3]
  gather_facts: false

  tasks:
    - name: do a thing
      ansible.builtin.debug:
        msg: "I am groot"
      changed_when: inventory_hostname in play_hosts[0:2]
      register: groot
      notify: restart groot

  handlers:
    - name: restart groot
      debug:
        msg: "I am groot?"
      loop: "{{ play_hosts }}"
      delegate_to: "{{ item }}"
      run_once: true
      when: hostvars[item]['groot']['changed'] | bool
      loop_control:
        pause: 2
```

Upon execution of this playbook, we can see the handler notification (thanks to double verbosity using the following command):

```
ansible-playbook -i mastery-hosts forserial.yaml -vv
```

In the handler task, we can see the loop, conditional, and delegation, as the following screenshot shows:

```
jfreeman@mastery1: ~/examples/Chapter11/example10
File Edit View Search Terminal Help

TASK [do a thing] *****
task path: /home/jfreeman/examples/Chapter11/example10/forserial.yaml:7
NOTIFIED HANDLER restart groot for failer01
changed: [failer01] => {
    "msg": "I am groot"
}
NOTIFIED HANDLER restart groot for failer02
changed: [failer02] => {
    "msg": "I am groot"
}
ok: [failer03] => {
    "msg": "I am groot"
}
ok: [failer04] => {
    "msg": "I am groot"
}

RUNNING HANDLER [restart groot] *****
task path: /home/jfreeman/examples/Chapter11/example10/forserial.yaml:15
ok: [failer01 -> failer01] => (item=failer01) => {
    "msg": "I am groot?"
}
ok: [failer01 -> failer02] => (item=failer02) => {
    "msg": "I am groot?"
}
skipping: [failer01] => (item=failer03)  => {"ansible_loop_var": "item", "item": "failer03"}
skipping: [failer01] => (item=failer04)  => {"ansible_loop_var": "item", "item": "failer04"}
META: ran handlers
META: ran handlers

PLAY RECAP *****
failer01      : ok=2    changed=1    unreachable=0    failed=0    s
kipped=0    rescued=0   ignored=0
failer02      : ok=1    changed=1    unreachable=0    failed=0    s
kipped=0    rescued=0   ignored=0
failer03      : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0   ignored=0
failer04      : ok=1    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0   ignored=0
```

Figure 11.9 – A playbook with a serialized handler routing for the restart of services

If you have tried this code out for yourself, you will notice the delay between each handler run, just as we specified in the `loop_control` part of the task. Using these techniques, you can confidently roll out updates and upgrades to your environment while keeping disruption to a minimum. It is hoped that this chapter has given you the tools and techniques to confidently perform such actions on your environment.

## Summary

Deployment and upgrade strategies are a matter of taste. Each strategy comes with distinct advantages and disadvantages. Ansible does not declare an opinion about which is better, and therefore it is well suited to perform deployments and upgrades regardless of the strategy. Ansible provides features and design patterns that facilitate a variety of styles with ease. Understanding the nature of each strategy and how Ansible can be tuned for that strategy will empower you to decide on and design deployments for each of your applications. Task controls and built-in variables provide methods to efficiently upgrade large-scale applications while treating specific tasks carefully.

In this chapter, you learned how to use Ansible to perform in-place upgrades and some different methodologies for these, including techniques such as expanding and contracting an environment. You learned about failing fast to ensure that playbooks don't cause extensive damage if an early part of a play goes wrong, and how to minimize both disruptive and destructive actions. Finally, you learned about serializing single tasks to minimize disruption to running services by taking nodes out of service in a minimal controlled manner. This ensures that services remain operational while maintenance work (such as an upgrade) occurs behind the scenes.

In the next chapter, we'll go into detail about using Ansible to work with cloud infrastructure providers and container systems in order to create an infrastructure to manage.

## Questions

1. What is a valid strategy for minimizing disruption when in-place upgrades are performed?
  - a) Use the `serial` mode to alter how many hosts Ansible performs the upgrade on at one time.
  - b) Use the `limit` parameter to alter how many hosts Ansible performs the upgrade on at one time.
  - c) Have lots of small inventories, each with just a few hosts in.
  - d) Revoke access to the hosts by Ansible.

2. What is a key benefit of expanding and contracting as an upgrade strategy?
  - a) Reduced cloud operating costs.
  - b) It fits well with a **development-operations (DevOps)** culture.
  - c) All hosts are newly built for each application deployment or upgrade, reducing the possibility of stale libraries and configurations.
  - d) It provides flexibility in your approach to upgrades.
3. Why would you want to fail fast?
  - a) So that you know about your playbook errors as soon as possible.
  - b) So that you minimize the damage or disruption caused by a failed play.
  - c) So that you can debug your code.
  - d) So that you can be agile in your deployments.
4. Which Ansible play option would you use to ensure that your play stops executing early in the event of errors on any single host?
  - a) `ansible.builtin.fail`
  - b) `any_errors_fatal`
  - c) `when: failed`
  - d) `max_fail_percentage: 50`
5. Which Ansible play option would you use to ensure that your play stops executing early in the event of errors on more than 30 percent of the hosts in your inventory?
  - a) `any_errors_fatal`
  - b) `max_fail_percentage: 30%`
  - c) `max_fail_percentage: 30`
  - d) `max_fail: 30%`
6. Which play-level option can you specify to ensure that your handlers are run even if your play fails?
  - a) `handlers_on_fail`
  - b) `handlers_on_failure`
  - c) `always_handlers`
  - d) `force_handlers`

7. Why might you want to delay the running of handlers to the end of your play?
  - a) It could save time on the execution of the play as a whole.
  - b) It makes the operation more predictable.
  - c) It reduces the risk of downtime.
  - d) It might help increase your chances of a successful upgrade.
8. Which task-level parameter can you use to ensure that a task does not get executed more than once, even when you have multiple hosts in your inventory?
  - a) `task_once`
  - b) `run_once`
  - c) `limit: 1`
  - d) `run: once`
9. Which `loop_control` parameter can insert a delay between iterations of a loop in Ansible?
  - a) `pause`
  - b) `sleep`
  - c) `delay`
  - d) `wait_for`
10. Which task conditional could you use to ensure you only run a task on the first four hosts in an inventory?
  - a) `when: inventory_hostname in play_hosts[0:3]`
  - b) `when: inventory_hostname in play_hosts[1:4]`
  - c) `when: inventory_hostname[0:3]`
  - d) `when: play_hosts[0:3]`

# 12

# Infrastructure Provisioning

Almost everything in data centers is becoming software-defined, from networks to the server infrastructure on which our software runs. **Infrastructure as a Service (IaaS)** providers offer APIs for programmatically managing images, servers, networks, and storage components. These resources are often expected to be created just-in-time, in order to reduce costs and increase efficiency.

As a result, a great deal of effort has gone into the cloud provisioning aspect of Ansible over the years, with more than 30 infrastructure providers catered for in the official Ansible release. These range from open source solutions such as OpenStack and oVirt to proprietary providers such as VMware and cloud providers such as AWS, GCP, and Azure.

There are more use cases than we can cover in this chapter, but nonetheless, we will explore the following ways in which Ansible can interact with a variety of these services:

- Managing an on-premise cloud infrastructure
- Managing a public cloud infrastructure
- Interacting with Docker containers
- Building containers with Ansible

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running **Ansible 4.3** or newer. Almost any flavor of Linux should do – for those interested in specifics, all the code presented in this chapter was tested on Ubuntu Server 20.04 LTS unless stated otherwise, and on Ansible 4.3. The example code that accompanies this chapter can be downloaded from GitHub at this URL: <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter12>.

Check out the following video to see the Code in Action:  
<https://bit.ly/3BU6My2>

## Managing an on-premise cloud infrastructure

The cloud is a popular but vague term, used to describe IaaS. There are many types of resources that can be provided by a cloud, although the most commonly discussed are compute and storage. Ansible is capable of interacting with numerous cloud providers in order to discover, create, or otherwise manage resources within them. Note that although we will focus on the compute and storage resources in this chapter, Ansible has a module for interacting with many more cloud resource types, such as load balancers, and even cloud role-based access controls.

One such cloud provider that Ansible can interact with is OpenStack (an open source cloud operating system), and this is a likely solution for those with a need for on-premise IaaS functionality. A suite of services provides interfaces to manage compute, storage, and networking services, plus many other supportive services. There is not a single provider of OpenStack; instead, many public and private cloud providers build their products with OpenStack, and thus although the providers may themselves be disparate, they provide the same APIs and software interfaces so that Ansible can automate tasks with ease in these environments.

Ansible has supported OpenStack services since very early in the project, and this support can now be found as part of the `OpenStack.cloud` collection. That initial support has grown to include over 70 modules, with support for managing the following:

- Compute
- Bare-metal compute
- Compute images
- Authentication accounts

- Networks
- Object storage
- Block storage

In addition to performing **create, read, update, and delete (CRUD)** actions on the preceding types of resources, Ansible also includes the ability to use OpenStack (and other clouds) as an inventory source, and we touched on this earlier, in *Chapter 1, The System Architecture and Design of Ansible*. Again, the dynamic inventory provider maybe found in the `OpenStack.cloud` collection. Each execution of `ansible` or `ansible-playbook` that utilizes an OpenStack cloud as an inventory source will get on-demand information about what compute resources exist, and various facts about those compute resources. Since the cloud service is already tracking these details, this can reduce overheads by eliminating the manual tracking of resources.

To demonstrate Ansible's ability to manage and interact with cloud resources, we'll walk through two scenarios: a scenario to create and then interact with new compute resources and a scenario that will demonstrate using OpenStack as an inventory source.

## Creating servers

The OpenStack compute service provides an API for creating, reading, updating, and deleting virtual machine servers. Through this API, we'll be able to create the server for our demonstration. After accessing and modifying the server through SSH, we'll also use the API to delete the server. This self-service ability is a key feature of cloud computing.

Ansible can be used to manage these servers by using the various `openstack.cloud` modules:

- `openstack.cloud.server`: This module is used to create and delete virtual servers.
- `openstack.cloud.server_info`: This module is used to gather information about a server – in Ansible 2.9 and earlier, it returned these as facts, but this is no longer the case.
- `openstack.cloud.server_action`: This module is used to perform various actions on a server.
- `openstack.cloud.server_group`: This module is used to create and delete server groups.

- `openstack.cloud.server_volume`: This module is used to attach or detach block storage volumes from a server.
- `openstack.cloud.server_metadata`: This module is used to create, update, and delete metadata for virtual servers.

## Booting virtual servers

For our demonstration, we will use `openstack.cloud.server`. We'll need to provide authentication details about our cloud, such as the auth URL and our login credentials. In addition to this, we will need to set up our Ansible host with the correct prerequisite software for this module to function. As we discussed earlier in the book when addressing dynamic inventories, Ansible sometimes requires additional software or libraries on the host in order to function. In fact, it is a policy of the Ansible developers to not ship cloud libraries with Ansible itself, as they would rapidly become out of date, and different operating systems would require different versions – even the advent of collections has not changed this.

You can always find the software dependencies in the Ansible documentation for each module, so it is worth checking this when using a module for the first time (especially a cloud provider module). The Ansible host used for the demos throughout this book is based on Ubuntu Server 20.04 and in order for the `openstack.cloud.server` module to function, I had to run the following command first:

```
sudo apt install python3-openstacksdk
```

The exact software and version will depend on our host operating system and may change with newer Ansible releases. There may be native packages available for your operating system, or you could install this Python module with `pip`. It is worth spending a few minutes checking the best approach for your operating system before proceeding.

Once the prerequisite modules are in place, we can proceed with the server creation. For this, we'll need a flavor, an image, a network, and a name. You will also need a key, and this will need to be defined in the OpenStack GUI (or CLI) before proceeding. Naturally, these details may be different for each OpenStack cloud. For this demo, I am using a single, all-in-one VM based on **DevStack**, and I am using defaults as much as possible, to make it easy to follow. You can download DevStack and learn about getting started quickly here: <https://docs.openstack.org/devstack/latest/>.

I'll name our playbook `boot-server.yaml`. Our play starts with a name and uses `localhost` as the host pattern as the module we are calling talks to the OpenStack API from the local Ansible machine directly. As we do not rely on any local facts, I'll turn fact-gathering off as well:

```
---
```

```
- name: boot server
```

```
  hosts: localhost
```

```
  gather_facts: false
```

To create the server, I'll use the `openstack.cloud.server` module and provide the auth details relevant to an OpenStack cloud that I have access to, as well as a flavor, image, network, and name. Note the `key_name`, which indicates the SSH public key from the keypair you would have created for yourself in OpenStack prior to writing this playbook (as discussed previously in this chapter). This SSH public key is integrated into the `Fedora34` image we are using when it is first booted on OpenStack so that we can subsequently gain access to it over SSH. I also uploaded a `Fedora34` image for demonstration purposes in this chapter, as it allows greater manipulation than the default Cirros image that is included with OpenStack distributions. These images can be freely downloaded, ready-made, from <https://alt.fedoraproject.org/cloud/>. Finally, as you'd expect, I've obfuscated my password:

```
tasks:
```

```
  - name: boot the server
```

```
    openstack.cloud.server:
```

```
      auth:
```

```
        auth_url: "http://10.0.50.32/identity/v3"
```

```
        username: "demo"
```

```
        password: "password"
```

```
        project_name: "demo"
```

```
        project_domain_name: "default"
```

```
        user_domain_name: "default"
```

```
      flavor: "ds1G"
```

```
      image: "Fedora34"
```

```
      key_name: "mastery-key"
```

```
      network: "private"
```

```
      name: "mastery1"
```

**Important Note**

Authentication details can be written to an external file, which will be read by the underlying module code. This module code uses `openstacksdk`, a standard library for managing OpenStack credentials. Alternatively, they can be stored in an Ansible vault, as we described in *Chapter 3, Protecting Your Secrets with Ansible*, and then passed to the module as variables.

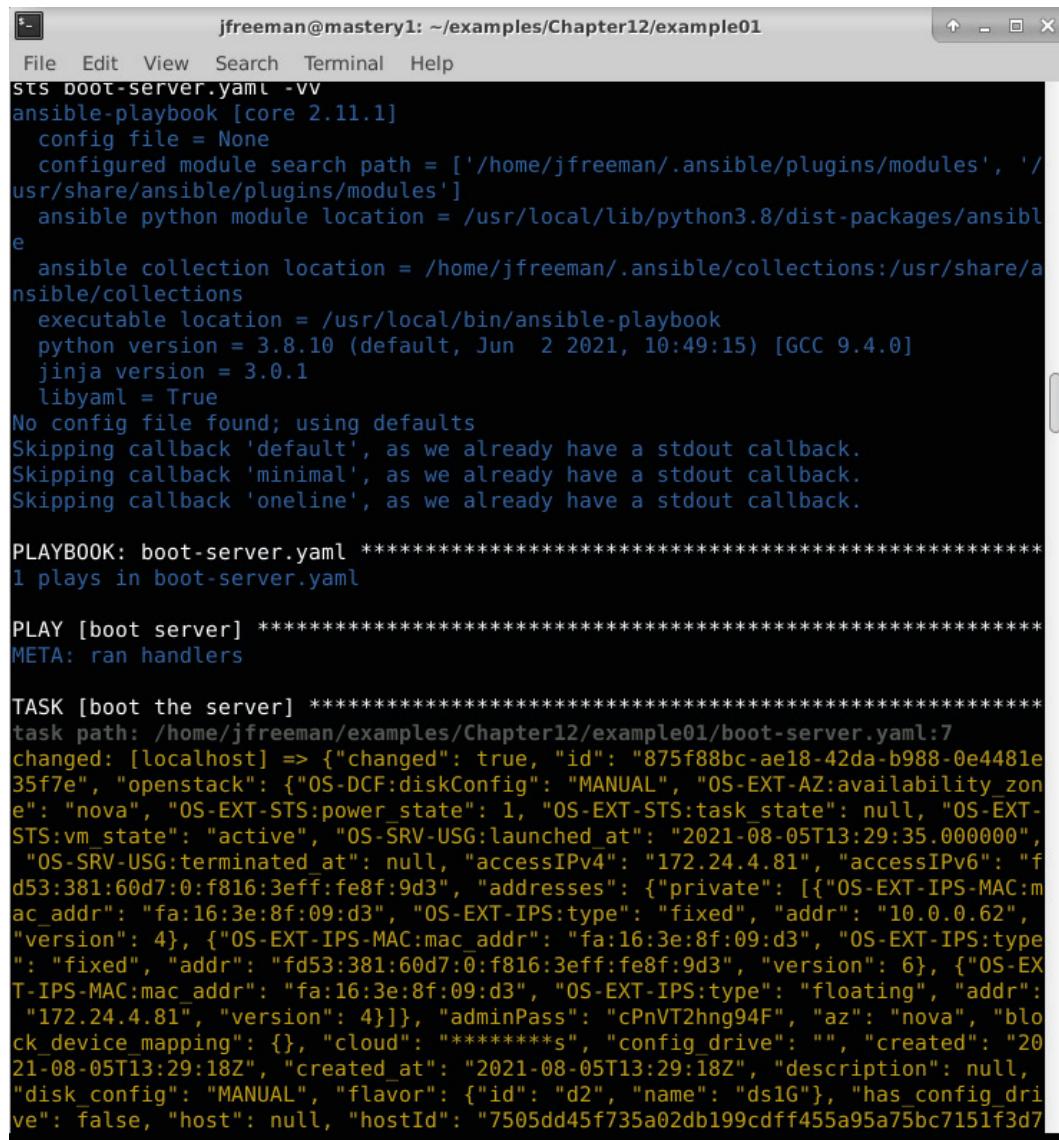
Running this play as is will simply create the server, and nothing more. To test this out (assuming you have access to a suitable OpenStack environment), run the playbook with the following commands:

```
export ANSIBLE PYTHON INTERPRETER=$(which python3)
ansible-playbook -i mastery-hosts boot-server.yaml -vv
```

**Ensuring the Correct Python Environment Is Used**

Note that on Ubuntu Server 20.04, Ansible runs by default under Python 2.7 – this is not a problem and we have ignored this so far in this book – however, in this particular instance, we have installed the `openstacksdk` module only on Python 3, and as a result, we must tell Ansible to use the Python 3 environment. We do this here by setting an environment variable, but you could just as easily do this via an `ansible.cfg` file – this is left as an exercise for you to explore.

A successful run of the playbook should yield output similar to that shown in *Figure 12.1*:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter12/example01". The terminal displays the output of running the "boot-server.yaml" playbook with the command "sts boot-server.yaml -vv". The output includes configuration details like module search paths, Python version (3.8.10), Jinja version (3.0.1), and libyaml (True). It also shows that no config file was found and default callbacks were skipped. The PLAYBOOK section starts with "PLAYBOOK: boot-server.yaml \*\*\*\*\*" followed by "1 plays in boot-server.yaml". The PLAY section for "[boot server]" starts with "PLAY [boot server] \*\*\*\*\*". The META section shows "ran handlers". Finally, the TASK section for "[boot the server]" starts with "TASK [boot the server] \*\*\*\*\*". The task path is "/home/jfreeman/examples/Chapter12/example01/boot-server.yaml:7". The task details a complex dictionary representing a virtual machine configuration, including fields like "id", "az", "nova", "power\_state", "task\_state", "vm\_state", "launched\_at", "terminated\_at", "accessIPv4", "accessIPv6", "addresses", "mac\_addr", "version", "OS-EXT-IPS:type", "OS-EXT-IPS-MAC:mac\_addr", "OS-EXT-IPS-MAC:fixed", "OS-EXT-IPS-MAC:floating", "adminPass", "az", "nova", "cloud", "config\_drive", "created", "disk\_config", "flavor", "hostId", and "hostId". The "version" field is explicitly noted as being 4.

```
jfreeman@mastery1: ~/examples/Chapter12/example01
File Edit View Search Terminal Help
sts boot-server.yaml -vv
ansible-playbook [core 2.11.1]
  config file = None
  configured module search path = ['/home/jfreeman/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
    ansible python module location = /usr/local/lib/python3.8/dist-packages/ansible
e
  ansible collection location = /home/jfreeman/.ansible/collections:/usr/share/ansible/collections
    executable location = /usr/local/bin/ansible-playbook
    python version = 3.8.10 (default, Jun 2 2021, 10:49:15) [GCC 9.4.0]
      jinja version = 3.0.1
      libyaml = True
No config file found; using defaults
Skipping callback 'default', as we already have a stdout callback.
Skipping callback 'minimal', as we already have a stdout callback.
Skipping callback 'oneline', as we already have a stdout callback.

PLAYBOOK: boot-server.yaml *****
1 plays in boot-server.yaml

PLAY [boot server] *****
META: ran handlers

TASK [boot the server] *****
task path: /home/jfreeman/examples/Chapter12/example01/boot-server.yaml:7
changed: [localhost] => {"changed": true, "id": "875f88bc-ae18-42da-b988-0e4481e35f7e", "openstack": {"OS-DCF:diskConfig": "MANUAL", "OS-EXT-AZ:availability_zone": "nova", "OS-EXT-STS:power_state": 1, "OS-EXT-STS:task_state": null, "OS-EXT-STS:vm_state": "active", "OS-SRV-USG:launched_at": "2021-08-05T13:29:35.000000", "OS-SRV-USG:terminated_at": null, "accessIPv4": "172.24.4.81", "accessIPv6": "fd53:381:60d7:0:f816:3eff:fe8f:9d3", "addresses": {"private": [{"OS-EXT-IPS-MAC:mac_addr": "fa:16:3e:8f:09:d3", "OS-EXT-IPS:type": "fixed", "addr": "10.0.0.62", "version": 4}, {"OS-EXT-IPS-MAC:mac_addr": "fa:16:3e:8f:09:d3", "OS-EXT-IPS:type": "fixed", "addr": "fd53:381:60d7:0:f816:3eff:fe8f:9d3", "version": 6}], "OS-EXT-IPS-MAC:mac_addr": "fa:16:3e:8f:09:d3", "OS-EXT-IPS:type": "floating", "addr": "172.24.4.81", "version": 4}], "adminPass": "cPnVT2hng94F", "az": "nova", "block_device_mapping": {}, "cloud": "*****s", "config_drive": "", "created": "2021-08-05T13:29:18Z", "created_at": "2021-08-05T13:29:18Z", "description": null, "disk_config": "MANUAL", "flavor": {"id": "d2", "name": "ds1G"}, "has_config_drive": false, "host": null, "hostId": "7505dd45f735a02db199cdff455a95a75bc7151f3d7"}}
```

Figure 12.1 – Creating a virtual instance in OpenStack with Ansible

I've truncated the output, as there is a lot of data returned from the module. Most importantly, we get data regarding the IP addresses of the host. This particular cloud uses a floating IP to provide public access to the server instance, which we can see the value of by registering the output and then debug printing the value of `openstack.accessIPv4`:

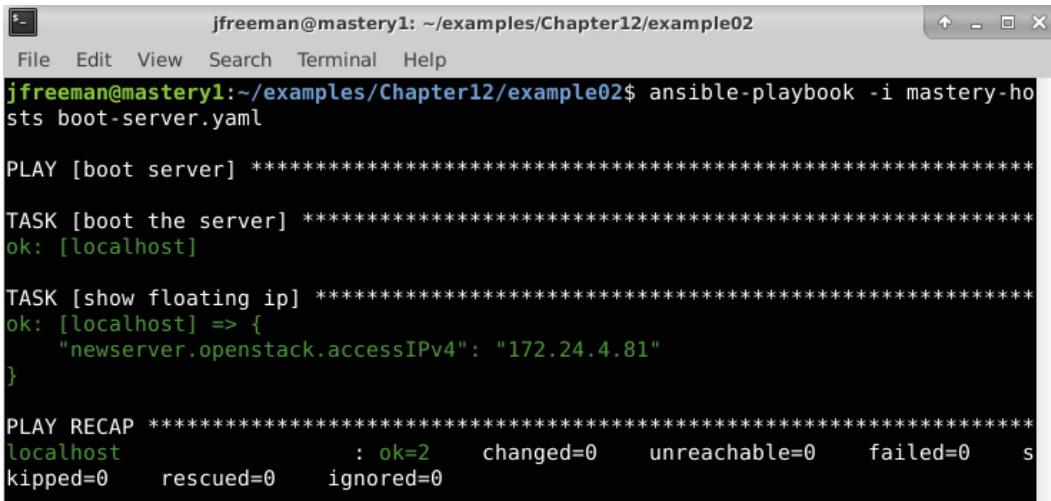
```
tasks:
  - name: boot the server
    openstack.cloud.server:
      auth:
        auth_url: "http://10.0.50.32/identity/v3"
        username: "demo"
        password: "password"
        project_name: "demo"
        project_domain_name: "default"
        user_domain_name: "default"
      flavor: "ds1G"
      image: "Fedora34"
      key_name: "mastery-key"
      network: "private"
      name: "mastery1"
      register: newserver

  - name: show floating ip
    ansible.builtin.debug:
      var: newserver.openstack.accessIPv4
```

Execute this playbook using a command similar to the preceding command (but without the added verbosity):

```
export ANSIBLE PYTHON INTERPRETER=$(which python3)
ansible-playbook -i mastery-hosts boot-server.yaml
```

This time, the first task does not result in a change, as the server that we want already exists – however, it still retrieves the information about the server, enabling us to discover its IP address:



```
jfreeman@mastery1: ~/examples/Chapter12/example02
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter12/example02$ ansible-playbook -i mastery-hosts boot-server.yaml

PLAY [boot server] ****
TASK [boot the server] ****
ok: [localhost]

TASK [show floating ip] ****
ok: [localhost] => {
    "newserver.openstack.accessIPv4": "172.24.4.81"
}

PLAY RECAP ****
localhost : ok=2    changed=0    unreachable=0    failed=0    s
kipped=0   rescued=0   ignored=0
```

Figure 12.2 – Using Ansible to retrieve the IP address of the OpenStack virtual machine we booted in the previous example

The output shows an IP address of 172.24.4.81. I can use that information to connect to my newly created cloud server.

## Adding to runtime inventory

Booting a server isn't all that useful by itself. The server exists to be used and will likely need some configuration to become useful. While it's possible to have one playbook to create resources and a completely different playbook to manage configuration, we can also do it all from the same playbook. Ansible provides a facility to add hosts to the inventory as a part of a play, which will allow for the use of those hosts in subsequent plays.

Working from the previous example, we have enough information to add the new host to the runtime inventory, by way of the `ansible.builtin.add_host` module:

```
- name: add new server
  ansible.builtin.add_host:
    name: "mastery1"
    ansible_ssh_host: "{{ newserver.openstack.accessIPv4 }}"
    ansible_ssh_user: "fedora"
```

I know that this image has a default user of `fedora`, so I set a host variable accordingly, along with setting the IP address as the connection address.

**Important Note**

This example is also glossing over any required security group configuration in OpenStack, and any accepting of the SSH host key. Additional tasks can be added to manage these things, or you might pre-configure them as I have done in my environment.

With the server added to our inventory, we can do something with it. Let's imagine a scenario in which we want to use this cloud resource to convert an image file, using ImageMagick software. To accomplish this, we'll need a new play to make use of the new host. I know that this particular Fedora image does not contain Python, so we need to add Python and the Python bindings for `dnf` (so we can use the `ansible.builtin.dnf` module) as our first task, using the `ansible.builtin.raw` module:

```
- name: configure server
  hosts: mastery1
  gather_facts: false

  tasks:
    - name: install python
      ansible.builtin.raw: "sudo dnf install -y python python-dnf"
```

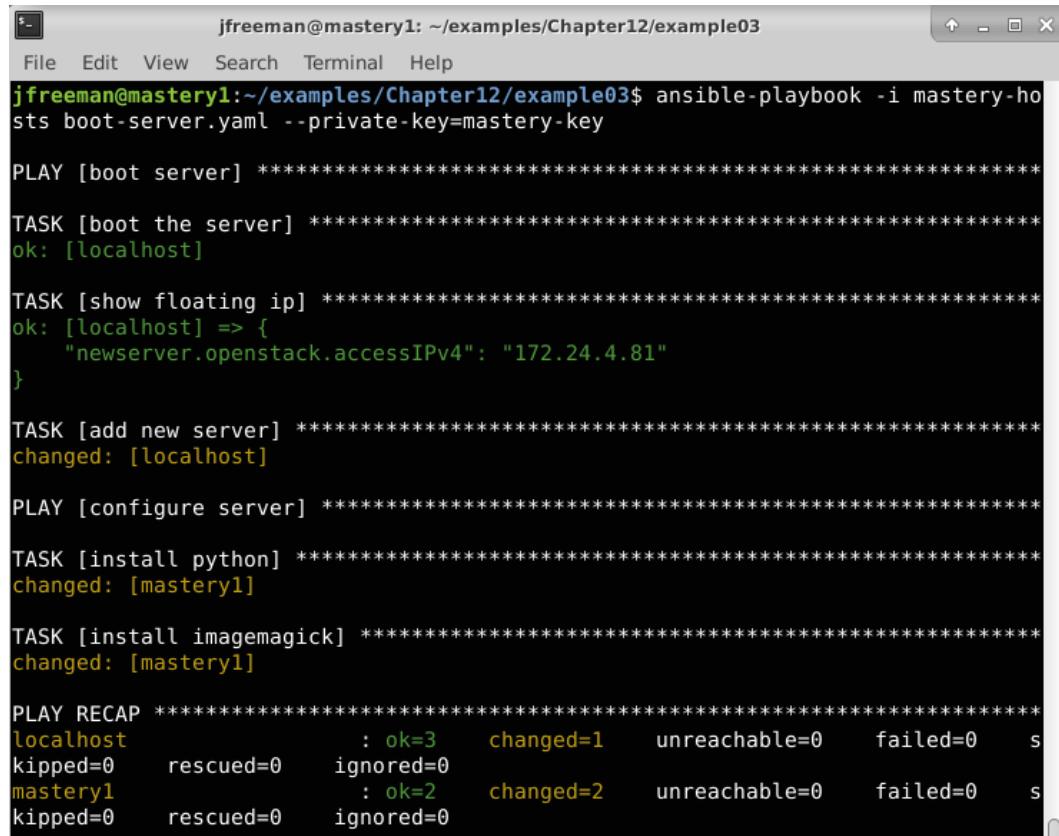
Next, we'll need the ImageMagick software, which we can install by using the `dnf` module:

```
- name: install imagemagick
  ansible.builtin.dnf:
    name: "ImageMagick"
    become: "yes"
```

Running the playbook at this point will show the changed tasks for our new host; note that this time, we must give `ansible-playbook` the location of our private key file from OpenStack, so that it can authenticate to the Fedora image, using the following command:

```
export ANSIBLE_PYTHON_INTERPRETER=$(which python3)
ansible-playbook -i mastery-hosts boot-server.yaml --private-key=mastery-key
```

A successful run of the playbook should yield output like that shown in *Figure 12.3*:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter12/example03". The terminal displays the output of an Ansible playbook run. The output shows the following tasks being executed:

- PLAY [boot server]
- TASK [boot the server] (ok: [localhost])
- TASK [show floating ip] (ok: [localhost] => { "newserver.openstack.accessIPv4": "172.24.4.81" })
- TASK [add new server] (changed: [localhost])
- PLAY [configure server]
- TASK [install python] (changed: [mastery1])
- TASK [install imagemagick] (changed: [mastery1])
- PLAY RECAP

For the host `mastery1`, the summary statistics are:

Host	Status	Ok	Changed	Unreachable	Failed	Skip
localhost		3	1	0	0	0
mastery1		2	2	0	0	0
All hosts		5	3	0	0	0

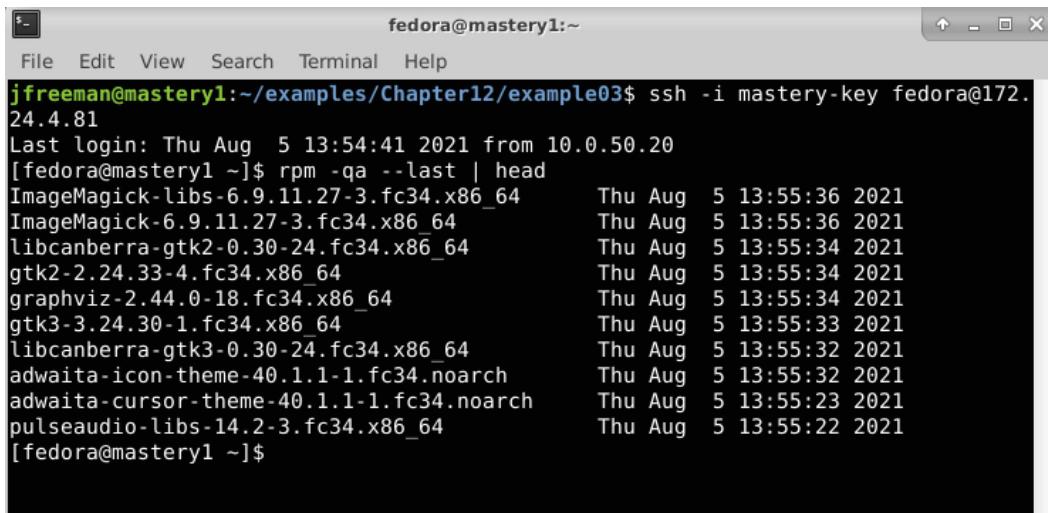
Figure 12.3 – Performing post instantiation configuration on our OpenStack virtual machine using Ansible

We can see Ansible reporting two changed tasks on the host `mastery1`, which we just created in the first play. This host does not exist in the `mastery-hosts` inventory file.

We have turned off verbose reporting here, too, as the output would otherwise be very cumbersome to wade through; however, given that we have the private key file for our OpenStack instance, we can manually log in and check the results of our playbook, for example, using a command like the following:

```
rpm -qa --last | head
```

This command queries the RPM package database and displays a short list of the most recently installed ones. The output might look something like that shown in *Figure 12.4*, though dates will undoubtedly vary:



The screenshot shows a terminal window titled "fedora@mastery1:~". The window contains the following command and its output:

```
jfreeman@mastery1:~/examples/Chapter12/example03$ ssh -i mastery-key fedora@172.24.4.81
Last login: Thu Aug  5 13:54:41 2021 from 10.0.50.20
[fedora@mastery1 ~]$ rpm -qa --last | head
ImageMagick-libs-6.9.11.27-3.fc34.x86_64      Thu Aug  5 13:55:36 2021
ImageMagick-6.9.11.27-3.fc34.x86_64          Thu Aug  5 13:55:36 2021
libcanberra-gtk2-0.30-24.fc34.x86_64          Thu Aug  5 13:55:34 2021
gtk2-2.24.33-4.fc34.x86_64                  Thu Aug  5 13:55:34 2021
graphviz-2.44.0-18.fc34.x86_64                Thu Aug  5 13:55:34 2021
gtk3-2.24.30-1.fc34.x86_64                  Thu Aug  5 13:55:33 2021
libcanberra-gtk3-0.30-24.fc34.x86_64          Thu Aug  5 13:55:32 2021
adwaiata-icon-theme-40.1.1-1.fc34.noarch       Thu Aug  5 13:55:32 2021
adwaiata-cursor-theme-40.1.1-1.fc34.noarch     Thu Aug  5 13:55:23 2021
pulseaudio-libs-14.2-3.fc34.x86_64            Thu Aug  5 13:55:22 2021
[fedora@mastery1 ~]$
```

Figure 12.4 – Checking the success of our playbook on our OpenStack VM

From here, we could extend our second play to upload a source image file by using `ansible.builtin.copy`, then perform a command by using `ImageMagick` on the host to convert the image. Another task can be added to fetch the converted file back down by using the `ansible.builtin.slurp` module or the modified file can be uploaded to a cloud-based object store. Finally, a last play can be added to delete the server itself.

The entire lifespan of the server, from creation to configuration to use, and finally, to removal, can all be managed with a single playbook. The playbook can be made dynamic by reading runtime variable data, in order to define what file should be uploaded/modified and where it should be stored, essentially turning the playbook into a reusable program. Although somewhat simplistic, hopefully, this gives you a clear idea of how powerful Ansible is for working with infrastructure service providers.

## Using OpenStack inventory sources

Our previous example showed a single-use, short-lived cloud server. What if we want to create and use long-lived cloud servers, instead? Walking through the tasks of creating them and adding them to the temporary inventory each time we want to touch them seems inefficient. Manually recording the server details in a static inventory also seems inefficient, and also error-prone. Thankfully, there is a better way: using the cloud itself as a dynamic inventory source.

Ansible ships with a number of dynamic inventory scripts for cloud providers, as we discussed in *Chapter 1, The System Architecture and Design of Ansible*. We'll continue our examples here with OpenStack. To recap, the `openstack.cloud` collection provides the dynamic inventory script that we need. To make use of this script, we need to create a YAML file that tells Ansible to utilize this inventory script – this file must be named `openstack.yaml` or `openstack.yml`. It should contain code that looks something like the following:

```
# file must be named openstack.yaml or openstack.yml
plugin: openstack.cloud.openstack
expand_hostvars: yes
fail_on_errors: yes
all_projects: yes
```

The configuration file needs a bit more consideration. This file holds authentication details for the OpenStack cloud(s) to connect to. That makes this file sensitive, and it should only be made visible to the users that require access to this information. In addition, the inventory script will attempt to load the configuration from the standard paths used by `os-client-config` (<https://docs.openstack.org/os-client-config/latest/user/configuration.html#config-files>), the underlying authentication code. This means that the configuration for this inventory source can live in the following:

- `clouds.yaml` (in the current working directory when executing the inventory script)
- `~/.config/openstack/clouds.yaml`
- `/etc/openstack/clouds.yaml`

The first file that's found will be used. You can override this by adding the `clouds_yaml_path` to the `openstack.yaml` we created earlier in this section. For our example, I'll use a `clouds.yaml` file in the playbook directory alongside the script itself, in order to isolate configuration from any other paths.

Your `clouds.yaml` file will look very similar to the `auth:` section of the parameters to the `openstack.cloud.server` module we used in our earlier examples. There is one key difference though – in our earlier examples, we used the `demo` account and limited ourselves to the `demo` project in OpenStack. For us to query all instances across all projects (which we want to do to demonstrate some functionality), we need an account with administrator privileges rather than the `demo` account. For this part of the chapter, my `clouds.yaml` file contains the following:

```
clouds:  
  mastery_cloud:  
    auth:  
      auth_url: "http://10.0.50.32/identity/v3"  
      username: "admin"  
      password: "password"  
      project_name: "demo"  
      project_domain_name: "default"  
      user_domain_name: "default"
```

The actual dynamic inventory script has a built-in help function, which you can also use to learn more about it. If you can locate it on your system, you can run this – on my system I used this command:

```
python3 /usr/local/lib/python3.8/dist-packages/ansible_  
collections/openstack/cloud/scripts/inventory/openstack_  
inventory.py --help
```

There is one final thing to know before we get started: If you are using the Ansible 4.0 release, this ships with version 1.4.0 of the `openstack.cloud` collection. This has a bug in it that renders the dynamic inventory script inoperable. You can query your installed collection version using this command:

```
ansible-galaxy collection list | grep openstack.cloud
```

If you need to install a newer version, you can install it using this command:

```
ansible-galaxy collection install openstack.cloud
```

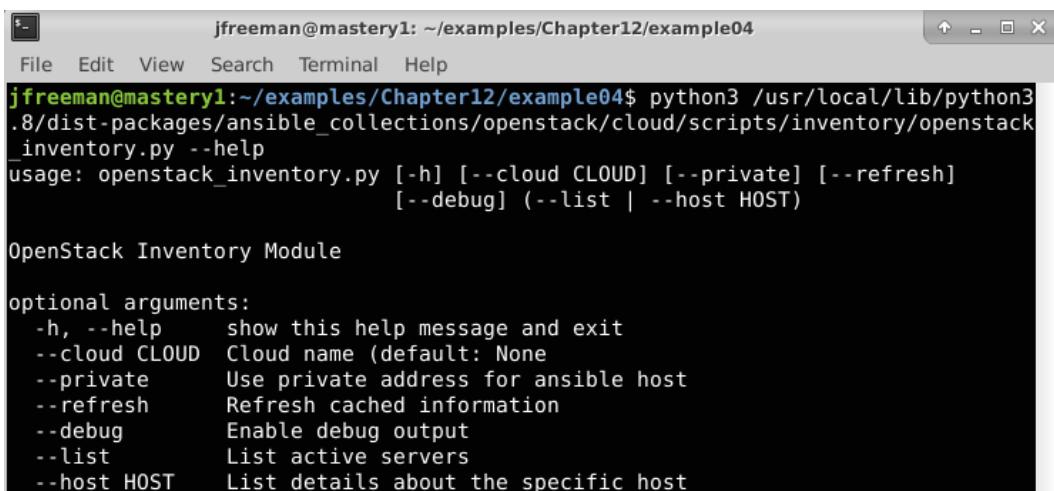
This will install the collection in a hidden directory within your home directory, so if you are using the local copy, don't use this command:

```
/usr/local/lib/python3.8/dist-packages/ansible_collections/
openstack/cloud/scripts/inventory/openstack_inventory.py
```

Use this instead:

```
~/.ansible/collections/ansible_collections/openstack/cloud/
scripts/inventory/openstack_inventory.py
```

The help output for the script shows a few possible arguments; however, the ones that Ansible will use are `--list` and `--host`, as *Figure 12.5* shows:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter12/example04". The window contains the following text:

```
jfreeman@mastery1:~/examples/Chapter12/example04$ python3 /usr/local/lib/python3.8/dist-packages/ansible_collections/openstack/cloud/scripts/inventory/openstack_inventory.py --help
usage: openstack_inventory.py [-h] [--cloud CLOUD] [--private] [--refresh]
                             [--debug] (--list | --host HOST)

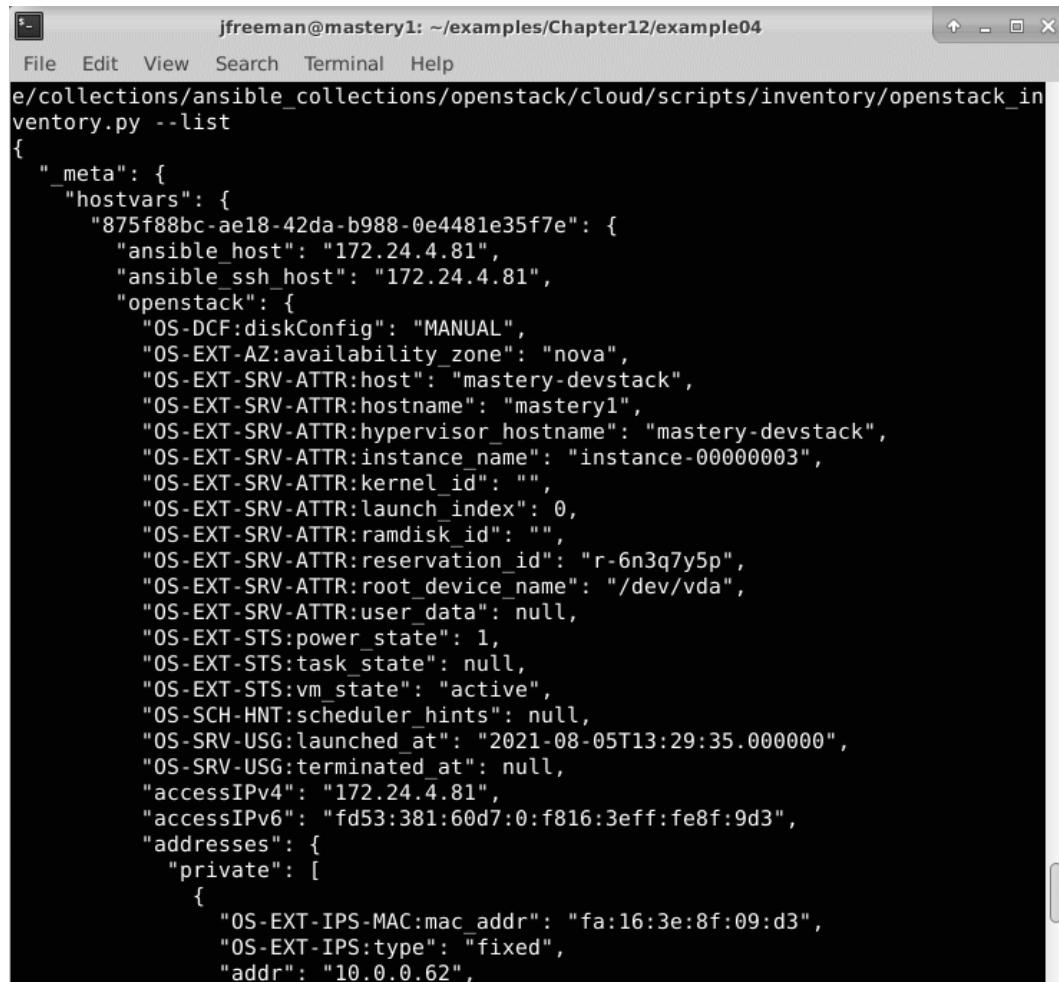
OpenStack Inventory Module

optional arguments:
  -h, --help            show this help message and exit
  --cloud CLOUD        Cloud name (default: None)
  --private             Use private address for ansible host
  --refresh             Refresh cached information
  --debug               Enable debug output
  --list                List active servers
  --host HOST          List details about the specific host
```

Figure 12.5 – Demonstrating the help function of the `openstack_inventory.py` script

The first is used to get a list of all of the servers visible to the account used, and the second would be used to get host variable data from each, except that this inventory script returns all of the host variables with the `--list` call. Returning the data with the host list is a performance enhancement, as we discussed earlier in the book, eliminating the need to call the OpenStack APIs for each and every host returned.

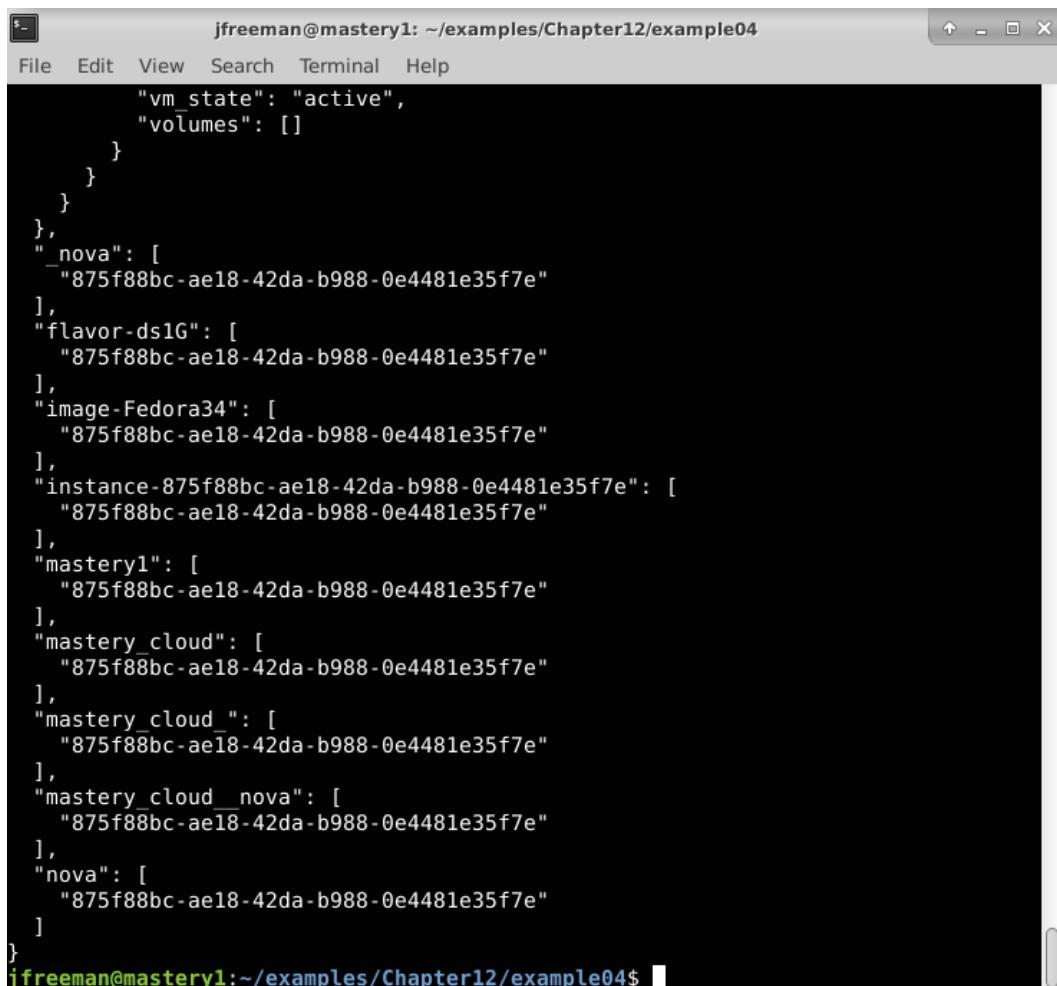
The output from `--list` is quite long; here are the first few lines:



```
jfreeman@mastery1: ~/examples/Chapter12/example04
File Edit View Search Terminal Help
e/collections/ansible_collections/openstack/cloud/scripts/inventory/openstack_inventory.py --list
{
  "_meta": {
    "hostvars": {
      "875f88bc-ae18-42da-b988-0e4481e35f7e": {
        "ansible_host": "172.24.4.81",
        "ansible_ssh_host": "172.24.4.81",
        "openstack": {
          "OS-DCF:diskConfig": "MANUAL",
          "OS-EXT-AZ:availability_zone": "nova",
          "OS-EXT-SRV-ATTR:host": "mastery-devstack",
          "OS-EXT-SRV-ATTR:hostname": "mastery1",
          "OS-EXT-SRV-ATTR:hypervisor_hostname": "mastery-devstack",
          "OS-EXT-SRV-ATTR:instance_name": "instance-00000003",
          "OS-EXT-SRV-ATTR:kernel_id": "",
          "OS-EXT-SRV-ATTR:launch_index": 0,
          "OS-EXT-SRV-ATTR:ramdisk_id": "",
          "OS-EXT-SRV-ATTR:reservation_id": "r-6n3q7y5p",
          "OS-EXT-SRV-ATTR:root_device_name": "/dev/vda",
          "OS-EXT-SRV-ATTR:user_data": null,
          "OS-EXT-STS:power_state": 1,
          "OS-EXT-STS:task_state": null,
          "OS-EXT-STS:vm_state": "active",
          "OS-SCH-HNT:scheduler_hints": null,
          "OS-SRV-USG:launched_at": "2021-08-05T13:29:35.000000",
          "OS-SRV-USG:terminated_at": null,
          "accessIPv4": "172.24.4.81",
          "accessIPv6": "fd53:381:60d7:0:f816:3eff:fe8f:9d3",
          "addresses": {
            "private": [
              {
                "OS-EXT-IPS-MAC:mac_addr": "fa:16:3e:8f:09:d3",
                "OS-EXT-IPS:type": "fixed",
                "addr": "10.0.0.62"
              }
            ]
          }
        }
      }
    }
  }
}
```

Figure 12.6 – Demonstrating the data returned by the `openstack_inventory.py` dynamic inventory

The configured account only has one visible server, which has a UUID of `875f88bc-ae18-42da-b988-0e4481e35f7e`, the instance that we booted in a previous example. We see this instance listed in the `flavor-ds1G` and `image-Fedora34` groups, for example. The first group is for all of the servers running with the `ds1G` flavor, and the second is for all servers running from our `Fedora34` image. These groupings happen automatically within the inventory plugin and may vary according to the OpenStack setup that you use. The tail end of the output will show the other groups provided by the plugin:



```
jfreeman@mastery1: ~/examples/Chapter12/example04$ ./openstack_inventory.py
{
    "group": "mastery1",
    "hosts": [
        "875f88bc-ae18-42da-b988-0e4481e35f7e"
    ],
    "group": "mastery_cloud",
    "hosts": [
        "875f88bc-ae18-42da-b988-0e4481e35f7e"
    ],
    "group": "mastery_cloud_nova",
    "hosts": [
        "875f88bc-ae18-42da-b988-0e4481e35f7e"
    ],
    "group": "nova",
    "hosts": [
        "875f88bc-ae18-42da-b988-0e4481e35f7e"
    ],
    "group": "flavor-ds1G",
    "hosts": [
        "875f88bc-ae18-42da-b988-0e4481e35f7e"
    ],
    "group": "image-Fedora34",
    "hosts": [
        "875f88bc-ae18-42da-b988-0e4481e35f7e"
    ],
    "group": "instance-875f88bc-ae18-42da-b988-0e4481e35f7e",
    "hosts": [
        "875f88bc-ae18-42da-b988-0e4481e35f7e"
    ],
    "group": "vm_state-active",
    "hosts": [
        "875f88bc-ae18-42da-b988-0e4481e35f7e"
    ]
}
jfreeman@mastery1:~/examples/Chapter12/example04$
```

Figure 12.7 – Demonstrating more data returned by the `openstack_inventory.py` dynamic inventory

**Important Note**

Note that for the preceding groupings to appear, `expand_hostvars: True` must be set in the `openstack.yaml` file.

Some of the additional groups are as follows:

- `mastery_cloud`: All servers running on our `mastery_cloud` instance, as specified in our `clouds.yaml` file
- `flavor-ds1G`: All servers that use the `ds1G` flavor
- `image-Fedora 29`: All servers that use the `Fedora 29` image

- `instance-875f88bc-ae18-42da-b988-0e4481e35f7e`: A group named after the instance itself
- `nova`: All servers running under the `nova` service

There are many groups provided, each with a potentially different slice of the servers found by the inventory script. These groups make it easy to target just the right instances with plays. The hosts are defined as the UUIDs of the servers. As these are unique by nature, and also quite long, they are unwieldy as a target within a play. This makes groups all the more important.

To demonstrate using this script as an inventory source, we'll recreate the previous example, skipping over the creation of the server and just writing the second play by using an appropriate group target. We'll name this playbook `configure-server.yaml`:

```
---
- name: configure server
  hosts: all
  gather_facts: false
  remote_user: fedora

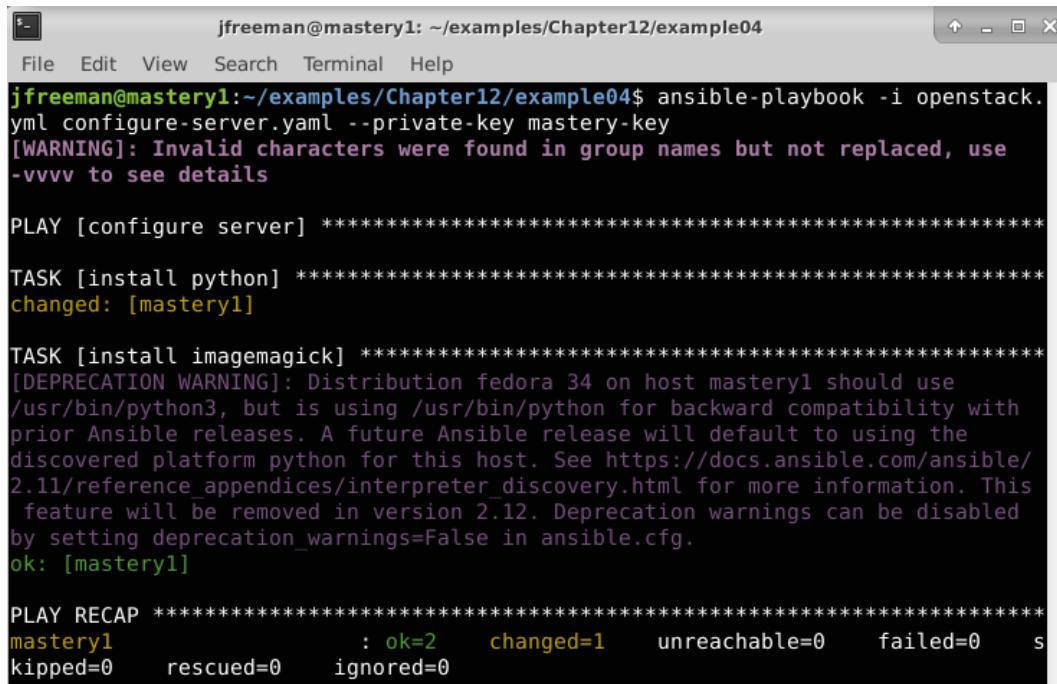
  tasks:
    - name: install python
      ansible.builtin.raw: "sudo dnf install -y python python-dnf"

    - name: install imagemagick
      ansible.builtin.dnf:
        name: "ImageMagick"
      become: "yes"
```

The default user of this image is `fedora`; however, that information isn't readily available via the OpenStack APIs, and thus, it is not reflected in the data that our inventory script provides. We can simply define the user to use at the play level.

This time, the host pattern is set to `all`, as we only have one host on our demo OpenStack server at this time; however, in real life, it's unlikely that you would be so open in your host targeting in Ansible.

The rest of the play is unchanged, and the output should look similar to previous executions:

A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter12/example04". The window shows the command "ansible-playbook -i openstack.yml configure-server.yaml --private-key mastery-key" being run. The output includes a warning about invalid characters in group names, followed by the execution of tasks for "configure server", "install python", and "install imagemagick". The "install imagemagick" task triggers a deprecation warning about Python version compatibility. The final PLAY RECAP summary shows 2 OK, 1 changed, 0 unreachable, 0 failed, 0 skipped, 0 rescued, and 0 ignored tasks.

```
jfreeman@mastery1:~/examples/Chapter12/example04$ ansible-playbook -i openstack.yml configure-server.yaml --private-key mastery-key
[WARNING]: Invalid characters were found in group names but not replaced, use -vvvv to see details

PLAY [configure server] ****
TASK [install python] ****
changed: [mastery1]

TASK [install imagemagick] ****
[DEPRECATION WARNING]: Distribution fedora 34 on host mastery1 should use /usr/bin/python3, but is using /usr/bin/python for backward compatibility with prior Ansible releases. A future Ansible release will default to using the discovered platform python for this host. See https://docs.ansible.com/ansible/2.11/reference_appendices/interpreter_discovery.html for more information. This feature will be removed in version 2.12. Deprecation warnings can be disabled by setting deprecation_warnings=False in ansible.cfg.
ok: [mastery1]

PLAY RECAP ****
mastery1 : ok=2    changed=1    unreachable=0    failed=0    s
kipped=0   rescued=0   ignored=0
```

Figure 12.8 – Reconfiguring our virtual instance via a dynamic inventory plugin

This output differs from the last time that the `boot-server.yaml` playbook was executed in only a few ways. First, the `mastery1` instance is not created or booted. We're assuming that the servers we want to interact with have already been created and are running. Secondly, we have pulled the inventory for this playbook run directly from the OpenStack server itself, using a dynamic inventory plugin, rather than creating one in the playbook using `add_host`. Otherwise, the output is the same, barring two deprecation warnings. The warning regarding group names comes up because the dynamic inventory script provides automatically created group names that need to be sanitized – I imagine this will be fixed in a future release of the plugin. In addition, the Python deprecation warning is common to see right now during this transitional phase as Ansible moves over completely to Python 3, and provided you are not missing any modules from your Python 2 environment, is benign.

As servers get added or removed over time, each execution of the inventory plugin will discover what servers are there at the moment of playbook execution. This can save a significant amount of time spent attempting to maintain an accurate list of servers in static inventory files.

# Managing a public cloud infrastructure

The management of public cloud infrastructures with Ansible is no more difficult than the management of OpenStack with it, as we covered earlier. In general, for any IaaS provider supported by Ansible, there is a three-step process to getting it working:

1. Establish the Ansible collections, modules, and inventory plugins available to support the cloud provider.
  2. Install any prerequisite software or libraries on the Ansible host.
  3. Define the playbook and run it against the infrastructure provider.

There are dynamic inventory plugins readily available for most providers too, and we have already demonstrated two in this book:

- `amazon.aws.aws_ec2` was discussed in *Chapter 1, The System Architecture and Design of Ansible*.
  - `openstack.cloud.openstack` was demonstrated earlier in this chapter.

Let's take a look at **Amazon Web Services (AWS)**, and specifically, the EC2 offering. We can boot up a new server from an image of our choosing, using exactly the same high-level process that we did with OpenStack earlier. However, as I'm sure you will have guessed by now, we have to use an Ansible module that offers specific EC2 support. Let's build up the playbook. First of all, our initial play will once again run from the local host, as this will be making the calls to EC2 to boot up our new server:

```
---  
- name: boot server  
  hosts: localhost  
  gather_facts: false
```

Next, we will use the `community.aws.ec2_instance` module in place of the `openstack.cloud.server` module to boot up our desired server. This code is really just an example to show you how to use the modules; normally, just like with our `openstack.cloud.server` example, you would not include the secret keys in the playbook, but would store them in a vault somewhere:

```
security_group: default
instance_type: t2.micro
image_id: "ami-04d4a52790edc7894"
region: eu-west-2
tags: "{'ansible_group':'mastery_server',
'Name':'mastery1'}"
wait: true
user_data: |
  #!/bin/bash
  sudo dnf install -y python python-dnf
register: newserver
```

#### Important Note

The `community.aws.ec2_instance` module requires the Python `boto3` library to be installed on the Ansible host; the method for this will vary between operating systems, but on our Ubuntu Server 20.04 demo host, it was installed using the `sudo apt install python3-boto3` command. Also, if you are installing this module under Python 3, be sure that your Ansible installation uses Python 3 by setting the `ANSIBLE PYTHON_INTERPRETER` variable.

The preceding code is intended to perform the same job as our `openstack.cloud.server` example, and while it looks similar at a high level, there are many differences. Hence, it is essential to read the module documentation whenever working with a new module, in order to understand precisely how to use it. Of specific interest, do note that the `user_data` field can be used to send post-creation scripts to the new VM; this is incredibly useful when the initial configuration is needed immediately, lending itself to `ansible.builtin.raw` commands. In this case, we use it to install the Python 3 prerequisites required to install ImageMagick with Ansible later on.

Next, we can obtain the public IP address of our newly created server by using the `newserver` variable that we registered in the preceding task. However, note the different variable structure, as compared to the way that we accessed this information when using the `openstack.cloud.server` module (again, always refer to the documentation):

```
- name: show floating ip
  ansible.builtin.debug:
    var: newserver.instances[0].public_ip_address
```

Another key difference between the `community.aws.ec2_instance` module and the `openstack.cloud.server` one is that `community.aws.ec2_instance` does not necessarily wait for SSH connectivity to become available before completing – this can be set using the `wait` parameter; thus, it is good practice to define a task specifically for this purpose to ensure that our playbook doesn't fail later on due to a lack of connectivity:

```
- name: Wait for SSH to come up
  ansible.builtin.wait_for_connection:
    delay: 5
    timeout: 320
```

Once this task has been completed, we will know that our host is alive and responding to SSH, so we can proceed to use `ansible.builtin.add_host` to add this new host to the inventory, and then install ImageMagick just like we did before (the image used here is the same Fedora 34 cloud-based image used in the OpenStack example):

```
- name: add new server
  ansible.builtin.add_host:
    name: "mastery1"
    ansible_ssh_host: "{{ newserver.instances[0].public_ip_address }}"
    ansible_ssh_user: "fedora"

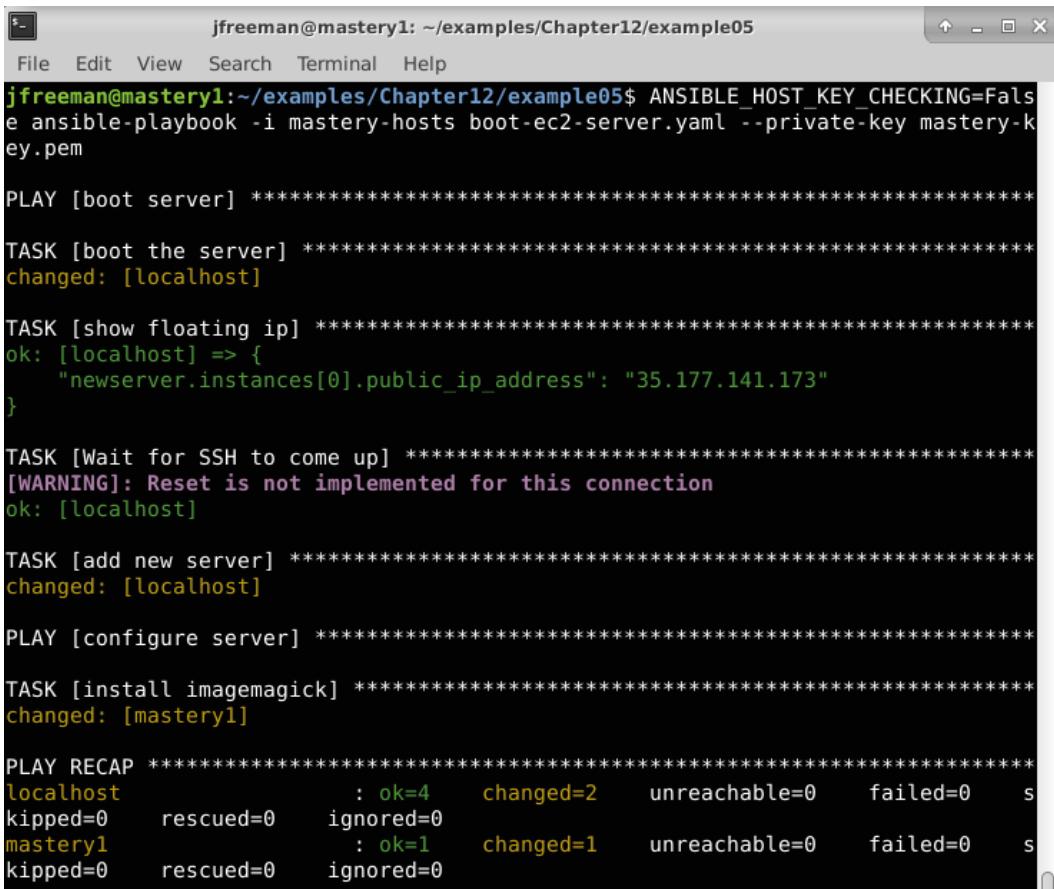
- name: configure server
  hosts: mastery1
  gather_facts: false

  tasks:
    - name: install imagemagick
      ansible.builtin.dnf:
        name: "ImageMagick"
      become: "yes"
```

Putting all of this together and running the playbook should result in something like the following screenshot. Note that I have turned SSH host key checking off, to prevent the SSH transport agent from asking about adding the host key on the first run, which would cause the playbook to hang and wait for user intervention, using this command:

```
export ANSIBLE_PYTHON_INTERPRETER=$(which python3)
ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook -i mastery-
hosts boot-ec2-server.yaml --private-key mastery-key.pem
```

You will also note that I have saved my private SSH key from the keypair I generated on my AWS account as `mastery-key.pem` in the same directory as the playbook – you will need to save your own key in this location and reference it in the command line accordingly. A successful run should look something like the output shown in *Figure 12.9*:

A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter12/example05". The window shows the execution of an Ansible playbook named "boot-ec2-server.yaml" with the command "ANSIBLE\_HOST\_KEY\_CHECKING=False". The output details the playbook's tasks: booting a server, showing floating IP, waiting for SSH, adding a new server, and configuring the server. It includes status messages like "changed" and "ok" for each task, and a final "PLAY RECAP" showing statistics for hosts "localhost" and "mastery1".

```
jfreeman@mastery1:~/examples/Chapter12/example05$ ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook -i mastery-hosts boot-ec2-server.yaml --private-key mastery-key.pem

PLAY [boot server] *****
TASK [boot the server] *****
changed: [localhost]

TASK [show floating ip] *****
ok: [localhost] => {
    "newserver.instances[0].public_ip_address": "35.177.141.173"
}

TASK [Wait for SSH to come up] *****
[WARNING]: Reset is not implemented for this connection
ok: [localhost]

TASK [add new server] *****
changed: [localhost]

PLAY [configure server] *****
TASK [install imagemagick] *****
changed: [mastery1]

PLAY RECAP *****
localhost              : ok=4    changed=2    unreachable=0    failed=0    s
kipped=0   rescued=0   ignored=0
mastery1             : ok=1    changed=1    unreachable=0    failed=0    s
kipped=0   rescued=0   ignored=0
```

Figure 12.9 – Booting and setting up an Amazon EC2 instance using Ansible

As we have seen here, we can achieve the same result on a different cloud provider, using only a subtly different playbook. The key here is to read the documentation that comes with each module and ensure that both the parameters and return values are correctly referenced.

We could apply this methodology to Azure, Google Cloud, or any of the other cloud providers that Ansible ships with support for. If we wanted to repeat this example on Azure, then we would need to use the `azure.azure_rm_virtualmachine` module. The documentation for this module states that we need Python 2.7 or newer (this is already a part of our Ubuntu Server 20.04 demo machine), and a whole suite of Python modules, the names of which along with required versions can be found in a file named `requirements-azure.txt`, which is included with the collection. The expectation is that you will install these requirements with `pip`, and you can do this by locating the aforementioned file on your filesystem and then installing the required modules. On my demo system, I achieved this with the following commands:

```
locate requirements-azure.txt
sudo pip3 install -r /usr/local/lib/python3.8/dist-packages/
ansible_collections/azure/azcollection/requirements-azure.txt
```

With these prerequisites satisfied, we can build up our playbook again. Note that with Azure, multiple authentication methods are possible. For the sake of simplicity, I am using the Azure Active Directory credentials that I created for this demo; however, to enable this, I had to also install the official Azure CLI utility (following the instructions available here: <https://docs.microsoft.com/en-gb/cli/azure/install-azure-cli-linux?pivot=apt>), and log in using the following:

```
az login
```

This ensures that your Ansible host is trusted by Azure. In practice, you would set up a **service principal** that removes the need for this, and you are encouraged to explore this option by yourself. To continue with the current simple example, we set up the header of our playbook like before:

```
---
- name: boot server
  hosts: localhost
  gather_facts: false
  vars:
    vm_password: Password123!
```

Note that this time, we will store a password for our new VM in a variable; normally, we would do this in a vault, but that is again left as an exercise for the reader. From here, we use the `azure.azcollection.azure_rm_virtualmachine` module to boot up our new VM. To make use of a Fedora 34 image for continuity with the previous examples, I've had to go to the image marketplace on Azure, which requires some additional parameters, such as `plan`, to be defined. To enable the use of this image with Ansible, I first had to find it, then accept the terms of the author to enable its use, using the `az` command-line utility with these commands:

```
az vm image list --offer fedora --all --output table
az vm image show --urn tunnelbiz:fedora:fedoraupdate:34.0.1
az vm image terms accept --urn
tunnelbiz:fedora:fedoraupdate:34.0.1
```

I also had to create the resource group and network that the VM would use; these are very much Azure-specific steps and are well documented (and considered *bread and butter* if you are familiar with Azure). Once all of the prerequisites were completed, I was then able to write the following playbook code to boot up our Azure-based Fedora 34 image:

```
tasks:
  - name: boot the server
    azure.azcollection.azure_rm_virtualmachine:
      ad_user: masteryadmin@example.com
      password: < insert your ad password here >
      subscription_id: xxxxxxxx-xxxxxx-xxxxxx-xxxxxxx
      resource_group: mastery
      name: mastery1
      admin_username: fedora
      admin_password: "{{ vm_password }}"
      vm_size: Standard_B1s
      managed_disk_type: "Standard_LRS"
      image:
        offer: fedora
        publisher: tunnelbiz
        sku: fedoraupdate
        version: 34.0.1
      plan:
```

```
name: fedoraupdate
product: fedora
publisher : tunnelbiz
register: newserver
```

As with the previous examples, we obtain the public IP address of our image (note the complex variable required to access this), ensure that SSH access is working, and then use `ansible.builtin.add_host` to add the new VM to our runtime inventory:

```
- name: show floating ip
  ansible.builtin.debug:
    var: newserver.ansible_facts.azure_vm.properties.
networkProfile.networkInterfaces[0].properties.
ipConfigurations[0].properties.publicIPAddress.properties.
ipAddress

- name: Wait for SSH to come up
  ansible.builtin.wait_for_connection:
    delay: 1
    timeout: 320

- name: add new server
  ansible.builtin.add_host:
    name: "mastery1"
    ansible_ssh_host: "{{ newserver.ansible_facts.azure_
vm.properties.networkProfile.networkInterfaces[0].properties.
ipConfigurations[0].properties.publicIPAddress.properties.
ipAddress }}"
    ansible_ssh_user: "fedora"
    ansible_ssh_pass: "{{ vm_password }}"
    ansible_become_pass: "{{ vm_password }}"
```

Azure allows for either password- or key-based authentication for SSH on Linux VMs; we're using password-based here for simplicity. Also, note the newly utilized `ansible_become_pass` connection variable, as the Fedora 34 image that we are using will prompt for a password when `sudo` is used, potentially blocking execution. Finally, with this work complete, we install ImageMagick, like before:

```
- name: configure server
  hosts: mastery1
  gather_facts: false

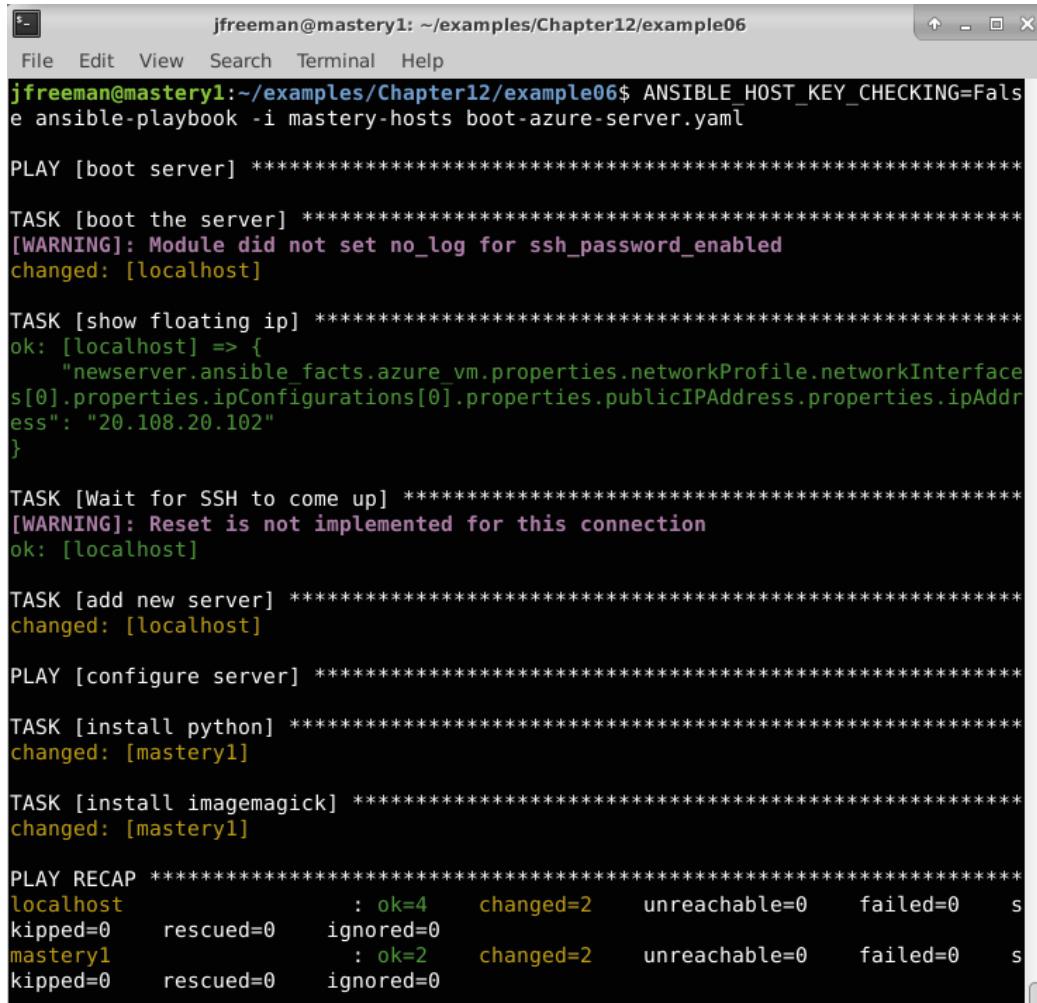
  tasks:
    - name: install python
      ansible.builtin.raw: "dnf install -y python python-dnf"
      become: "yes"

    - name: install imagemagick
      ansible.builtin.dnf:
        name: "ImageMagick"
      become: "yes"
```

With the code complete, run it with the following commands (setting your Python environment as necessary for your system):

```
export ANSIBLE PYTHON INTERPRETER=$(which python3)
ANSIBLE HOST KEY CHECKING=False ansible-playbook -i mastery-
hosts boot-azure-server.yaml
```

Let's take a look at this in action:



```
jfreeman@mastery1: ~/examples/Chapter12/example06
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter12/example06$ ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook -i mastery-hosts boot-azure-server.yaml

PLAY [boot server] ****
TASK [boot the server] ****
[WARNING]: Module did not set no_log for ssh_password_enabled
changed: [localhost]

TASK [show floating ip] ****
ok: [localhost] => {
    "newserver.ansible_facts.azure_vm.properties.networkProfile.networkInterfaces[0].properties.ipConfigurations[0].properties.publicIPAddress.properties.ipAddress": "20.108.20.102"
}

TASK [Wait for SSH to come up] ****
[WARNING]: Reset is not implemented for this connection
ok: [localhost]

TASK [add new server] ****
changed: [localhost]

PLAY [configure server] ****
TASK [install python] ****
changed: [mastery1]

TASK [install imagemagick] ****
changed: [mastery1]

PLAY RECAP ****
localhost                  : ok=4    changed=2    unreachable=0    failed=0    s
kipped=0      rescued=0    ignored=0
mastery1                 : ok=2    changed=2    unreachable=0    failed=0    s
kipped=0      rescued=0    ignored=0
```

Figure 12.10 – Creating and configuring an Azure virtual machine using Ansible

The output is very similar to our AWS example, demonstrating that we can very easily perform the same actions across different cloud platforms with just a little effort in terms of learning how the various modules that each cloud provider needs works. This section of the chapter is by no means definitive, given the number of platforms and operations supported by Ansible, but we hope that the information provided gives an idea of the process and steps required for getting Ansible to integrate with a new cloud platform. Next, we will look at using Ansible to interact with Docker containers.

# Interacting with Docker containers

Linux container technologies, especially Docker, have grown in popularity in recent years, and this has continued since the previous edition of this book was published. Containers provide a fast path to resource isolation while maintaining the consistency of the runtime environment. They can be launched quickly and are efficient to run, as there is very little overhead involved. Utilities such as Docker provide a lot of useful tooling for container management, such as a registry of images to use as the filesystem, tooling to build the images themselves, clustering orchestration, and so on. Through its ease of use, Docker has become one of the most popular ways to manage containers, though others, such as Podman and LXC, are becoming much more prevalent. For now, though, we will focus on Docker, given its broad appeal and wide install base.

Ansible can interact with Docker in numerous ways as well. Notably, Ansible can be used to build images, to start or stop containers, to compose multiple container services, to connect to and interact with active containers, and even to discover inventory from containers. Ansible provides a full suite of tools for working with Docker, including relevant modules, a connection plugin, and an inventory script.

To demonstrate working with Docker, we'll explore a few use cases. The first use case is building a new image to use with Docker. The second use case is launching a container from the new image and interacting with it. The last use case is using the inventory plugin to interact with an active container.

## Important Note

Creating a functional Docker installation is very much dependent on your underlying operating system. A great resource to start with is the Docker website, which provides detailed installation and usage instructions, at <https://docs.docker.com>. Ansible works best with Docker on a Linux host, so we will continue with the Ubuntu Server 20.04 LTS demo machine that we have used throughout this book.

## Building images

Docker images are basically filesystems bundled with parameters to use at runtime. The filesystem is usually a small part of a Linux Userland, with enough files to start the desired process. Docker provides tooling to build these images, generally based on very small, preexisting base images. The tooling uses a Dockerfile as the input, which is a plain text file with directives. This file is parsed by the `docker build` command, and we can parse it via the `docker_image` module. The remaining examples will be from an Ubuntu Server 20.04 virtual machine using Docker CE version 20.10.8, with the `cowsay` and `nginx` packages added so that running the container will provide a web server that will display something from `cowsay`.

First, we'll need a Dockerfile. If you've not come across one of these before, they are a set of instructions used for building Docker containers – you can learn more about this here if you wish: <https://docs.docker.com/engine/reference/builder/>. This file needs to live in a path that Ansible can read, and we're going to put it in the same directory as my playbooks. The Dockerfile content will be very simple. We'll need to define a base image, a command to run to install the necessary software, some minimal configuration of software, a port to expose, and a default action for running a container with this image:

```
FROM docker.io/fedora:34

RUN dnf install -y cowsay nginx
RUN echo "daemon off;" >> /etc/nginx/nginx.conf
RUN cowsay boop > /usr/share/nginx/html/index.html

EXPOSE 80

CMD /usr/sbin/nginx
```

The build process performs the following steps:

1. We're using the Fedora 34 image from the fedora repository on the Docker Hub image registry.
2. To install the necessary `cowsay` and `nginx` packages, we're using `dnf`.
3. To run `nginx` directly in the container, we need to turn `daemon mode off` in `nginx.conf`.
4. We use `cowsay` to generate content for the default web page.
5. Then, we're instructing Docker to expose port 80 in the container, where `nginx` will listen for connections.
6. Finally, the default action of this container will be to run `nginx`.

The playbook to build and use the image can live in the same directory. We'll name it `docker-interact.yaml`. This playbook will operate on `localhost` and will have two tasks; one will be to build the image using `community.docker.docker_image`, and the other will be to launch the container using `community.docker.docker_container`:

```
---
- name: build an image
  hosts: localhost
  gather_facts: false

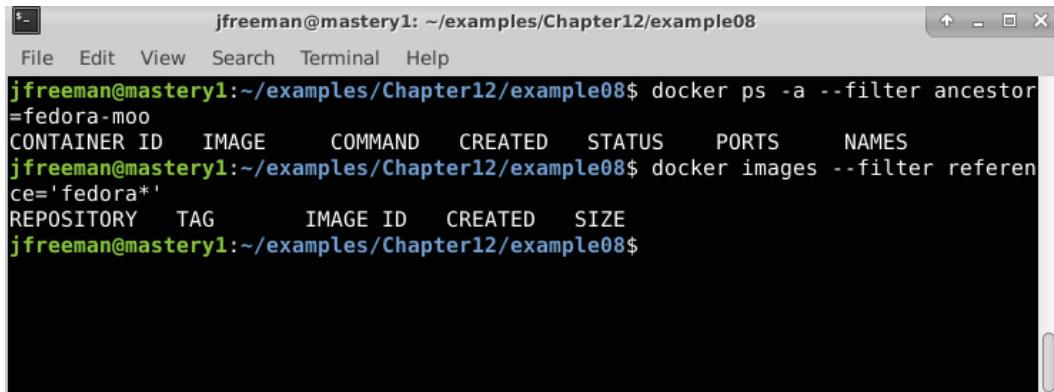
  tasks:
    - name: build that image
      community.docker.docker_image:
        path: .
        state: present
        name: fedora-moo

    - name: start the container
      community.docker.docker_container:
        name: playbook-container
        image: fedora-moo
        ports: 8080:80
        state: started
        container_default_behavior: no_defaults
```

Before we run our playbook, we'll check for any possible container images, or running containers that might match our preceding playbook definitions – this will help us to have confidence that our code is producing the desired results. If you have any additional containers running from previous tests, you can run the following commands to check for `fedora`-based containers that match our specification by running these commands:

```
docker ps -a --filter ancestor=fedora-moo
docker images --filter reference='fedora*' 
```

Unless you have run this code before, you should see that no containers are running, as shown in *Figure 12.11*:

A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter12/example08". The window shows two commands being run: "docker ps -a --filter ancestor=fedora-moo" and "docker images --filter reference='fedora\*'". Both commands return empty results, indicating no containers or images are present.

```
jfreeman@mastery1:~/examples/Chapter12/example08$ docker ps -a --filter ancestor=fedora-moo
CONTAINER ID   IMAGE      COMMAND   CREATED     STATUS      PORTS     NAMES
jfreeman@mastery1:~/examples/Chapter12/example08$ docker images --filter reference='fedora*'
REPOSITORY    TAG      IMAGE ID      CREATED     SIZE
jfreeman@mastery1:~/examples/Chapter12/example08$
```

Figure 12.11 – Checking for the absence of containers before running our playbook

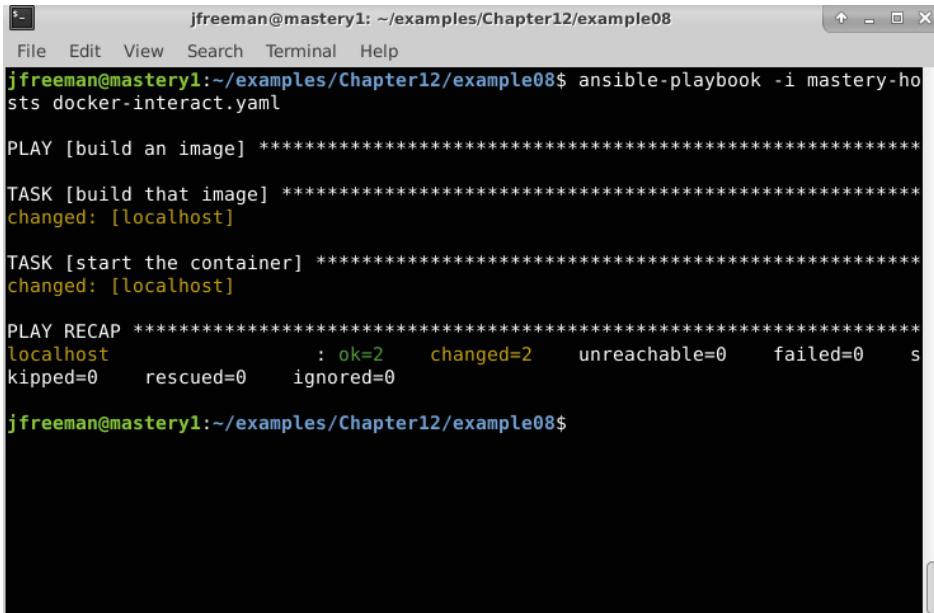
Now, let's run the playbook to build the image and start a container using that image – note that, as is common with many other Ansible modules, you might have to install additional Python modules for your code to work. On my Ubuntu Server 20.04 demo machine, I had to run the following:

```
sudo apt install python3-docker
export ANSIBLE PYTHON INTERPRETER=$(which python3)
```

With the Python support installed, you can then run the playbook with this command:

```
ansible-playbook -i mastery-hosts docker-interact.yaml
```

A successful playbook run should look similar to *Figure 12.12*:



```
jfreeman@mastery1: ~/examples/Chapter12/example08$ ansible-playbook -i mastery-hosts docker-interact.yaml

PLAY [build an image] ****
TASK [build that image] ****
changed: [localhost]

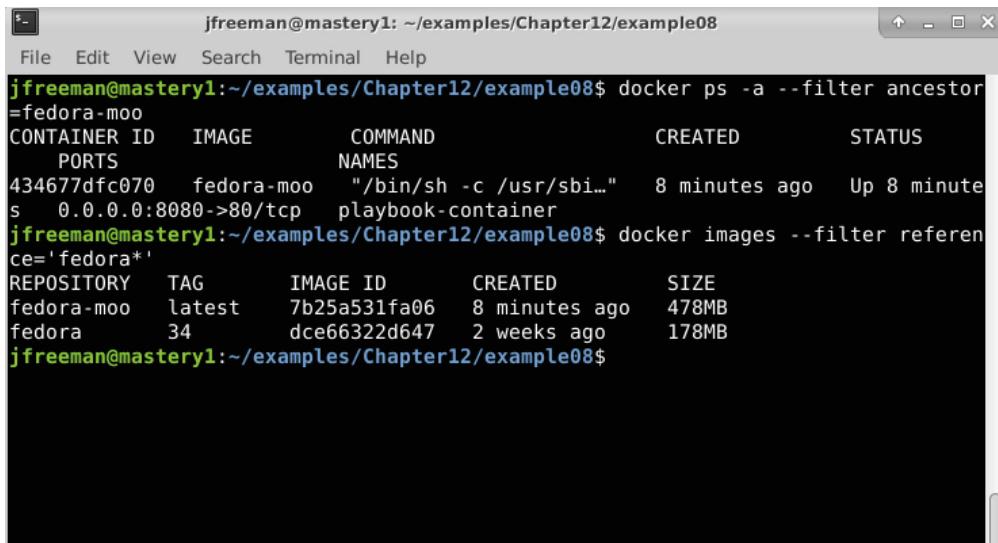
TASK [start the container] ****
changed: [localhost]

PLAY RECAP ****
localhost : ok=2    changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

jfreeman@mastery1:~/examples/Chapter12/example08$
```

Figure 12.12 – Building and running our first Docker container using Ansible

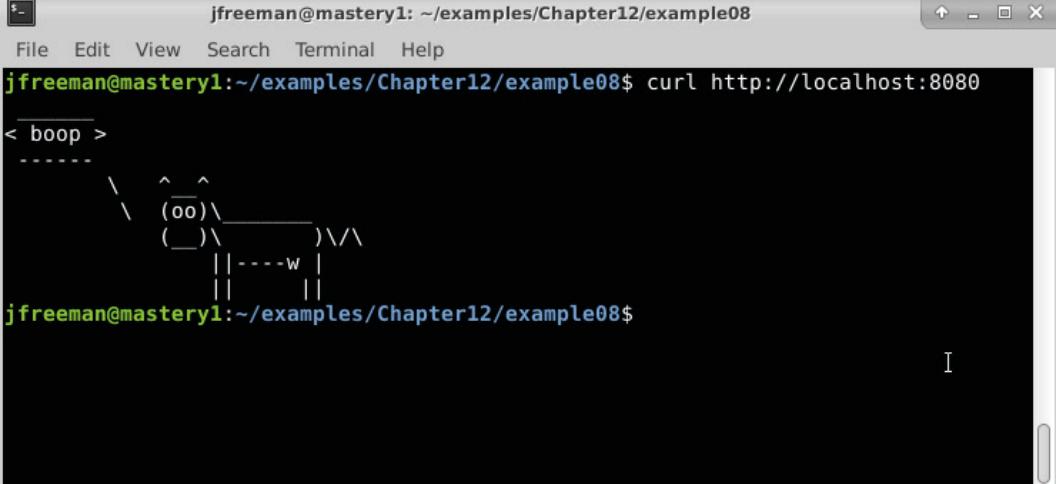
The verbosity of this playbook execution was reduced to save screen space. Our output simply shows that the task to build the image resulted in a change, as did the task to start the container. A quick check of running containers and available images should reflect our work – you can use the same `docker` commands as we used before the playbook run to validate this:



```
jfreeman@mastery1: ~/examples/Chapter12/example08$ docker ps -a --filter ancestor=fedora-moo
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
434677dfc070      fedora-moo        "/bin/sh -c /usr/sbi..."   8 minutes ago     Up 8 minutes
0.0.0.0:8080->80/tcp   playbook-container
jfreeman@mastery1:~/examples/Chapter12/example08$ docker images --filter reference='fedora*'
REPOSITORY          TAG           IMAGE ID            CREATED             SIZE
fedora-moo         latest        7b25a531fa06   8 minutes ago     478MB
fedora             34            dce66322d647   2 weeks ago       178MB
jfreeman@mastery1:~/examples/Chapter12/example08$
```

Figure 12.13 – Verifying the results of our Ansible playbook run in Docker

We can test the functionality of our container by using curl to access the web server, which should show us a cow saying boop, as demonstrated in *Figure 12.14*:



```
jfreeman@mastery1:~/examples/Chapter12/example08
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter12/example08$ curl http://localhost:8080
< boop >
-----
\ ^__^
  (oo)\_____
    (__)\       )\/\
        ||----w |
        ||     ||
jfreeman@mastery1:~/examples/Chapter12/example08$
```

Figure 12.14 – Retrieving the results of our container created and run with Ansible

In this manner, we have already shown how easy it is to interact with Docker using Ansible. However, this example is still based on using a native Dockerfile, and, as we progress through this chapter, we'll see some more advanced Ansible usage that removes the need for this.

## Building containers without a Dockerfile

Dockerfiles are useful, but many of the actions performed inside of Dockerfiles could be completed with Ansible instead. Ansible can be used to launch a container using a base image, then interact with that container using the docker connection method (as opposed to SSH) to complete the configuration. Let's demonstrate this by repeating the previous example, but without the need for a Dockerfile. Instead, all of the work will be handled by an entirely new playbook named `docker-all.yaml`. The first part of this playbook starts a container from a preexisting image of Fedora 34 from Docker Hub and adds the resulting container details to Ansible's in-memory inventory by using `ansible.builtin.add_host`:

```
---
- name: build an image
  hosts: localhost
  gather_facts: false
```

```
tasks:
  - name: start the container
    community.docker.docker_container:
      name: playbook-container
      image: docker.io/fedora:34
      ports: 8080:80
      state: started
      command: sleep 500
      container_default_behavior: no_defaults

  - name: make a host
    ansible.builtin.add_host:
      name: playbook-container
      ansible_connection: docker
      ansible_ssh_user: root
```

Then, using this newly added inventory host, we define a second play that runs Ansible tasks within the container that was just launched, configuring our `cowsay` service like before, but without the need for a Dockerfile:

```
- name: do things
  hosts: playbook-container
  gather_facts: false

  tasks:
    - name: install things
      ansible.builtin.raw: dnf install -y python-dnf

    - name: install things
      ansible.builtin.dnf:
        name: ['nginx', 'cowsay']

    - name: configure nginx
      ansible.builtin.lineinfile:
        line: "daemon off;"
        dest: /etc/nginx/nginx.conf
```

```
- name: boop
  ansible.builtin.shell: cowsay boop > /usr/share/nginx/
  html/index.html

- name: run nginx
  ansible.builtin.shell: nginx &
```

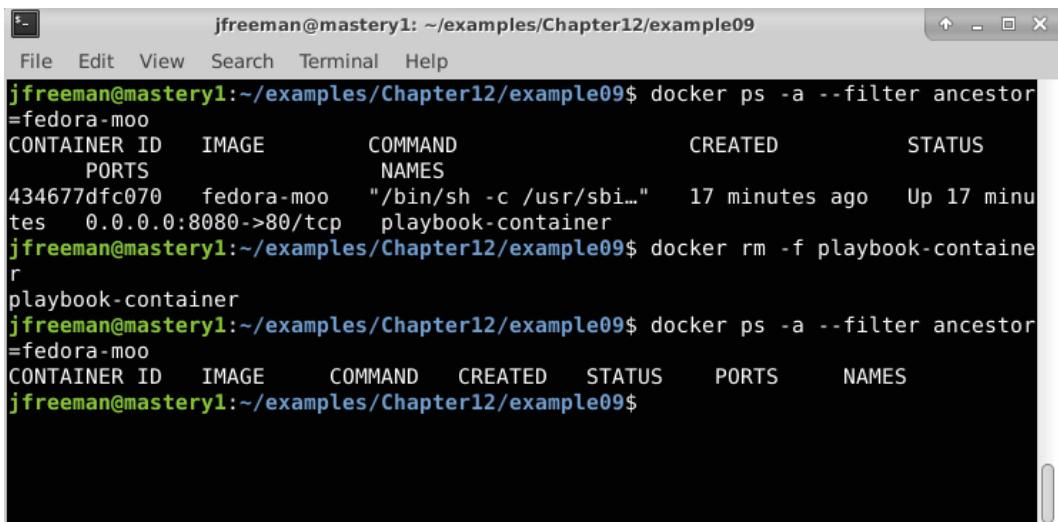
To recap, the playbook consists of two plays. The first play creates the container from the base `Fedora 34` image. The `community.docker.docker_container` task is given a `sleep` command to keep the container running for a period of time, as the `docker` connection plugin only works with active containers (unconfigured operating system images from Docker Hub generally exit immediately when they are run, as they have no default actions to perform). The second task of the first play creates a runtime inventory entry for the container. The inventory hostname must match the container name. The connection method is set to `docker` as well.

The second play targets the newly created host, and the first task uses the `ansible.builtin.raw` module to get the `python-dnf` package in place (which will bring the rest of Python in), so that we can use the `ansible.builtin.dnf` module in the next task. The `ansible.builtin.dnf` module is then used to install the desired packages, namely, `nginx` and `cowsay`. Then, the `ansible.builtin.lineinfile` module is used to add a new line to the `nginx` configuration. An `ansible.builtin.shell` task uses `cowsay` to create content for `nginx` to serve. Finally, `nginx` itself is started as a background process.

Before running the playbook, let's remove any running containers from the previous example by running the following:

```
docker ps -a --filter ancestor=fedora-moo
docker rm -f playbook-container
docker ps -a --filter ancestor=fedora-moo
```

You can verify this against the screenshot in *Figure 12.15*:

A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter12/example09". The window shows the command "docker ps -a --filter ancestor=fedora-moo" being run, listing a single container named "playbook-container" with ID "434677dfc070". It has a command of "/bin/sh -c /usr/sbi..." and was created 17 minutes ago. The status is "Up 17 minutes". The port mapping is 0.0.0.0:8080->80/tcp. The user then runs "docker rm -f playbook-container" to remove the container. Finally, "docker ps -a --filter ancestor=fedora-moo" is run again, showing no results, indicating the container has been removed.

```
jfreeman@mastery1:~/examples/Chapter12/example09$ docker ps -a --filter ancestor=fedora-moo
CONTAINER ID   IMAGE      COMMAND           CREATED          STATUS          PORTS
434677dfc070   fedora-moo   "/bin/sh -c /usr/sbi..."   17 minutes ago   Up 17 minutes   0.0.0.0:8080->80/tcp
playbook-container
jfreeman@mastery1:~/examples/Chapter12/example09$ docker rm -f playbook-container
playbook-container
jfreeman@mastery1:~/examples/Chapter12/example09$ docker ps -a --filter ancestor=fedora-moo
CONTAINER ID   IMAGE      COMMAND           CREATED          STATUS          PORTS
jfreeman@mastery1:~/examples/Chapter12/example09$
```

Figure 12.15 – Cleaning up running containers from our previous playbook run

With the running container removed, we can now run our new playbook to recreate the container, bypassing the image build step, using this command:

```
ansible-playbook -i mastery-hosts docker-all.yaml
```

The output of a successful run should look like that shown in *Figure 12.16*:

```
jfreeman@mastery1: ~/examples/Chapter12/example09$ ansible-playbook -i mastery-hosts docker-all.yaml

PLAY [build an image] ****
TASK [start the container] ****
changed: [localhost]

TASK [make a host] ****
changed: [localhost]

PLAY [do things] ****
TASK [install things] ****
changed: [playbook-container]

TASK [install things] ****
changed: [playbook-container]

TASK [configure nginx] ****
changed: [playbook-container]

TASK [boop] ****
changed: [playbook-container]

TASK [run nginx] ****
changed: [playbook-container]

PLAY RECAP ****
localhost                  : ok=2    changed=2    unreachable=0    failed=0    s
kipped=0      rescued=0    ignored=0
playbook-container          : ok=5    changed=5    unreachable=0    failed=0    s
kipped=0      rescued=0    ignored=0
```

Figure 12.16 – Building a container without a Dockerfile using Ansible

We see tasks from the first play execute on the `localhost`, and then the second play executes on the `playbook-container`. Once it's complete, we can test the web service and list the running containers to verify our work using these commands:

```
curl http://localhost:8080
docker ps -a --filter ancestor=fedora:34
```

Note the different filter this time; our container was built and run directly from the `fedora` image, without the intermediate step of creating the `fedora-moo` image – the output should look like that shown in *Figure 12.17*:

```
jfreeman@mastery1:~/examples/Chapter12/example09$ curl http://localhost:8080
< boop >
-----
 \ ^ ^
 (oo)\_____
 (_)\_____
 ||----w |
 ||      |

jfreeman@mastery1:~/examples/Chapter12/example09$ docker ps -a --filter ancestor=fedora:34
CONTAINER ID        IMAGE           COMMAND       CREATED          STATUS          PORTS
f38eb33c853a      fedora:34     "sleep 500"   4 minutes ago   Up 3 minutes   0.0.0.0:8080->80/tcp
playbook-container

jfreeman@mastery1:~/examples/Chapter12/example09$
```

Figure 12.17 – Verifying the results of our playbook run

This method of using Ansible to configure the running container has some advantages. First, you can reuse existing roles to set up an application, easily switching from cloud virtual machine targets to containers, and even to bare-metal resources, if desired. Secondly, you can easily review all configuration that goes into an application, simply by reviewing the playbook content.

Another use case for this method of interaction is to use Docker containers to simulate multiple hosts, in order to verify playbook execution across multiple hosts. A container can be started with an `init` system as the running process, allowing for additional services to be started as if they were on a full operating system. This use case is valuable within a continuous integration environment, to validate changes to playbook content quickly and efficiently.

## Docker inventory

Similar to the OpenStack and EC2 inventory plugins detailed earlier in this book, a Docker inventory plugin is also available. You can locate the Docker inventory script if you wish to examine it or use it in a similar manner to the way in which we have used other dynamic inventory plugins earlier in this chapter by creating a YAML inventory file to reference the plugin.

Let's start by locating the inventory script itself – on my demo system, it is located here:

```
/usr/local/lib/python3.8/dist-packages/ansible_collections/
community/general/scripts/inventory/docker.py
```

Once you get used to the installation base path of Ansible, you will note that it is quite easy to navigate the directory structure through the collections to find what you are looking for. Let's try running this script directly to see the options available to us when configuring it for playbook inventory purposes:

```
python3 /usr/local/lib/python3.8/dist-packages/ansible_
collections/community/general/scripts/inventory/docker.py
--help
```

The help output for the script shows many possible arguments; however, the ones that Ansible will use are `--list` and `--host` – your output will look similar to that shown in *Figure 12.18*:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter12/example10". The window contains the help output for the `docker.py` script. The output includes usage information, optional arguments, and detailed descriptions for each argument.

```
jfreeman@mastery1:~/examples/Chapter12/example10$ python3 /usr/local/lib/python3.8/dist-packages/ansible_collections/community/general/scripts/inventory/docker.py --help
usage: docker.py [-h] [--list] [--debug] [--host HOST] [--pretty]
                  [--config-file CONFIG_FILE] [--docker-host DOCKER_HOST]
                  [--tls-hostname TLS_HOSTNAME] [--api-version API_VERSION]
                  [--timeout TIMEOUT] [--cacert-path CACERT_PATH]
                  [--cert-path CERT_PATH] [--key-path KEY_PATH]
                  [--ssl-version SSL_VERSION] [--tls] [--tls-verify]
                  [--private-ssh-port PRIVATE_SSH_PORT]
                  [--default-ip-address DEFAULT_IP_ADDRESS]

Return Ansible inventory for one or more Docker hosts.

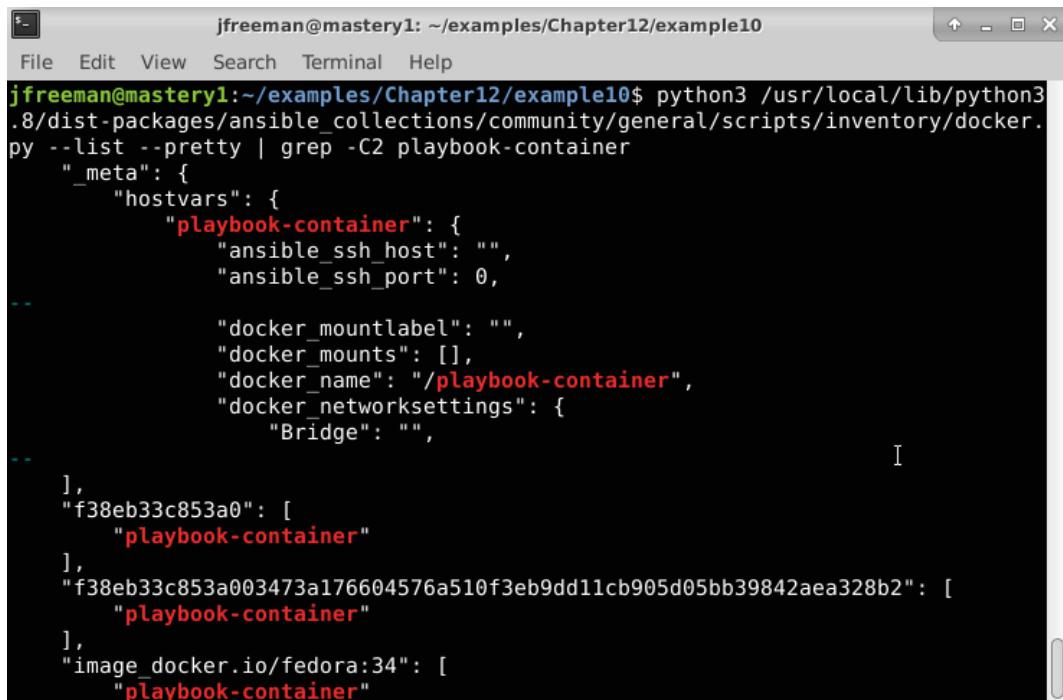
optional arguments:
  -h, --help            show this help message and exit
  --list               List all containers (default: True)
  --debug              Send debug messages to STDOUT
  --host HOST          Only get information for a specific container.
  --pretty             Pretty print JSON output(default: False)
  --config-file CONFIG_FILE
                      Name of the config file to use. Default is docker.yml
  --docker-host DOCKER_HOST
                      The base url or Unix sock path to connect to the
                      docker daemon. Defaults to unix://var/run/docker.sock
  --tls-hostname TLS_HOSTNAME
                      Host name to expect in TLS certs. Defaults to
                      localhost
  --api-version API_VERSION
                      Docker daemon API version. Defaults to 1.35
  --timeout TIMEOUT    Docker connection timeout in seconds. Defaults to 60
  --cacert-path CACERT_PATH
                      Path to the TLS certificate authority pem file.
  --cert-path CERT_PATH
                      Path to the TLS certificate pem file.
  --key-path KEY_PATH   Path to the TLS encryption key pem file.
  --ssl-version SSL_VERSION
                      TLS version number
  --tls                Use TLS. Defaults to False
  --tls-verify         Verify TLS certificates. Defaults to False
  --private-ssh-port PRIVATE_SSH_PORT
                      Default private container SSH Port. Defaults to 22
  --default-ip-address DEFAULT_IP_ADDRESS
                      Default container SSH IP address. Defaults to
                      127.0.0.1
```

Figure 12.18 – Examining the options available on the Docker dynamic inventory script

If the previously built container is still running when this script is executed, you can list hosts using the following command:

```
python3 /usr/local/lib/python3.8/dist-packages/ansible_
collections/community/general/scripts/inventory/docker.py
--list --pretty | grep -C2 playbook-container
```

It should appear in the output (grep has been used to make this more obvious in the screenshot):

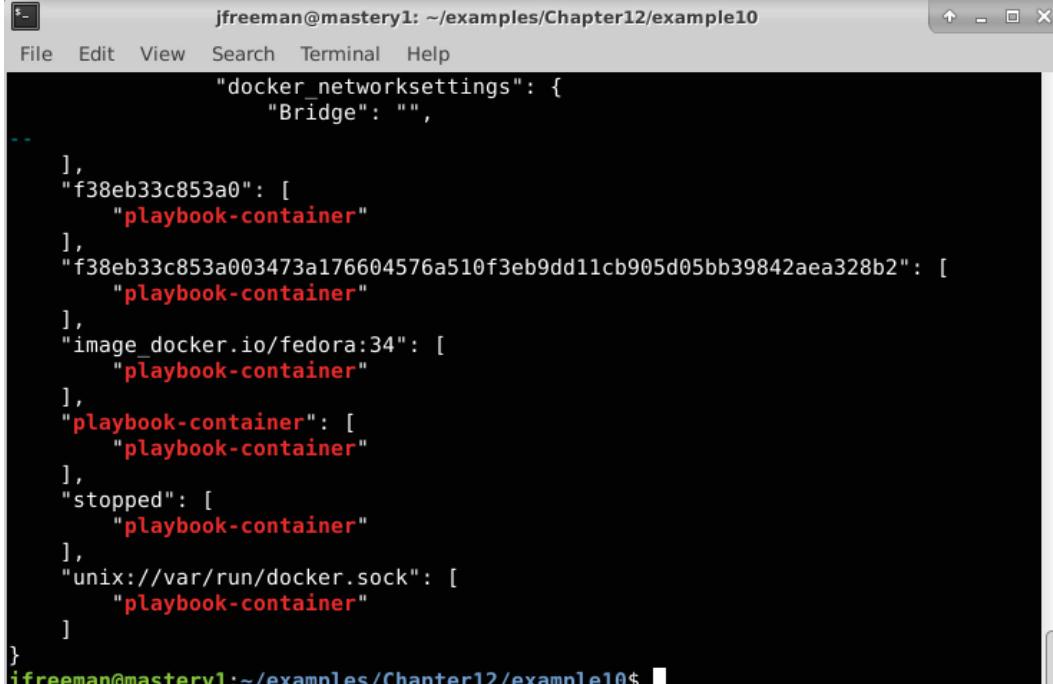


The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter12/example10". The window contains the command "python3 /usr/local/lib/python3.8/dist-packages/ansible\_collections/community/general/scripts/inventory/docker.py --list --pretty | grep -C2 playbook-container" followed by its output. The output is a JSON-like structure representing a list of hosts. Several host entries are highlighted in red, specifically "playbook-container". The structure includes fields like '\_meta', 'hostvars', 'ansible\_ssh\_host', 'ansible\_ssh\_port', 'docker\_mountlabel', 'docker\_mounts', 'docker\_name', 'docker\_networksettings', and 'Bridge'. There are also lists of host IDs and their corresponding names.

```
jfreeman@mastery1:~/examples/Chapter12/example10$ python3 /usr/local/lib/python3.8/dist-packages/ansible_collections/community/general/scripts/inventory/docker.py --list --pretty | grep -C2 playbook-container
{
    "_meta": {
        "hostvars": {
            "playbook-container": {
                "ansible_ssh_host": "",
                "ansible_ssh_port": 0,
                "docker_mountlabel": "",
                "docker_mounts": [],
                "docker_name": "/playbook-container",
                "docker_networksettings": {
                    "Bridge": ""
                }
            }
        }
    },
    "f38eb33c853a0": [
        "playbook-container"
    ],
    "f38eb33c853a003473a176604576a510f3eb9dd11cb905d05bb39842aea328b2": [
        "playbook-container"
    ],
    "image_docker.io/fedora:34": [
        "playbook-container"
    ]
}
```

Figure 12.19 – Running the Docker dynamic inventory plugin manually to explore its behavior

Like earlier, a number of groups are presented, which have the running container as a member. The two groups that were shown earlier are the short container ID and the long container ID. Many variables are also defined as a part of the output, which has been heavily truncated in the preceding screenshot. The tail end of the output reveals a few more groups:



```
jfreeman@mastery1: ~/examples/Chapter12/example10
File Edit View Search Terminal Help
    "docker_networksettings": {
        "Bridge": "",

    ],
    "f38eb33c853a0": [
        "playbook-container"
    ],
    "f38eb33c853a003473a176604576a510f3eb9dd11cb905d05bb39842aea328b2": [
        "playbook-container"
    ],
    "image_docker.io/fedora:34": [
        "playbook-container"
    ],
    "playbook-container": [
        "playbook-container"
    ],
    "stopped": [
        "playbook-container"
    ],
    "unix://var/run/docker.sock": [
        "playbook-container"
    ]
}
jfreeman@mastery1:~/examples/Chapter12/example10$
```

Figure 12.20 – Further exploring the output of the dynamic inventory script output

The additional groups are as follows:

- `docker_hosts`: All of the hosts running the Docker daemon that the dynamic inventory script has communicated with and queried for containers.
- `image_name`: A group for each image used by discovered containers.
- `container_name`: A group that matches the name of the container
- `running`: A group of all the running containers.
- `stopped`: A group of all the stopped containers – you can see in the preceding output that our container started previously has now stopped, as the 500-second sleep period has expired.

This inventory plugin, and the groups and data provided by it, can be used by playbooks to target various selections of containers available, in order to interact without the need for manual inventory management or the use of `add_host`. Using the plugin in a playbook is a simple matter of defining a YAML inventory file with the plugin name and connection details – to query the local Docker host, we could define our inventory as follows:

```
---  
plugin: community.docker.docker_containers  
docker_host: unix:/var/run/docker.sock
```

You can run ad hoc commands or playbooks with Ansible against this inventory definition in the normal manner and get details of all of the containers running on the local host. Connecting to remote hosts is not significantly more difficult, and the plugin documentation (available here: [https://docs.ansible.com/ansible/latest/collections/community/docker/docker\\_containers\\_inventory.html](https://docs.ansible.com/ansible/latest/collections/community/docker/docker_containers_inventory.html)) shows you the options available to you for this. We've now looked at several methods for building and interacting with Docker containers, but what if we wanted a more joined-up approach? We'll look at exactly this in the next section.

## Building containers with Ansible

As we mentioned at the beginning of the previous section, the world of containers has moved on greatly since the previous edition of this book was published. Although Docker is still a massively popular container technology, new and improved technologies have become favored, and indeed natively integrated into Linux operating systems. Canonical (the publisher of Ubuntu) is championing the **LXC** container environment, while Red Hat (the owner of Ansible) is championing **Buildah** and **Podman**.

If you read the third edition of this book, you will know that we covered a technology called **Ansible Container**, which was used to directly integrate Ansible with Docker and remove the need for *glue* steps such as adding hosts to the in-memory inventory, having two separate plays for instantiating the container, and building the container image contents. Ansible Container has now been deprecated, and all development work has ceased (according to their GitHub page – see <https://github.com/ansible/ansible-container> if you are interested).

Ansible Container has been succeeded by a new tool called **ansible-bender**, which features a pluggable architecture for different container build environments. At this early stage in its development, it only supports **Buildah**, but hopefully, further container technologies will be supported in the near future.

The Podman/Buildah toolsets are available on newer releases of Red Hat Enterprise Linux, CentOS, Fedora, and Ubuntu Server (but not 20.04, unless you go for a more bleeding edge version). As we have used Ubuntu Server for our demo machine throughout this book, we will stick to this operating system, but for this section of the chapter, we will switch to version 20.10 which, whilst not an LTS release, does have a native release of Buildah and Podman available.

To install Buildah and Podman on Ubuntu Server 20.10 (and newer), simply run the following:

```
sudo apt update  
sudo apt install podman runc
```

Once you have your container environment installed (don't forget to install Ansible if you haven't already – `ansible-bender` needs this to run!), you can install `ansible-bender` using the following command:

```
sudo pip3 install ansible-bender
```

That's it – now you're all ready to go! It is worth noting before we dive into our example code that `ansible-bender` is a lot simpler in its functionality than Ansible Container was. While Ansible Container could manage the entire lifecycle of a container, `ansible-bender` is only concerned with the build phase of containers – nonetheless, it provides a useful abstraction layer for easily building your container images using Ansible, and once it supports other containerization build platforms (such as LXC and/or Docker), it will become an incredibly valuable tool in your automation arsenal, as you will be able to build container images on a variety of platforms using almost identical playbook code.

Let's build our first playbook for `ansible-bender`. The header of the play will look, by now, familiar – with one important exception. Notice the `vars:` section in the play definition – this section contains important reserved variables for use by `ansible-bender` and defines items such as the source container image (we'll use `Fedora:34` once again), and the destination container image details, including the command to run when the container starts:

```
---  
- name: build an image with ansible-bender  
  hosts: localhost  
  gather_facts: false  
  vars:  
    ansible_bender:  
      base_image: fedora:34
```

```
target_image:  
  name: fedora-moo  
  cmd: nginx &
```

With this defined, we write our play tasks in exactly the same way as we did before. Notice that we don't need to worry about inventory definition (either through a dynamic inventory provider or via `ansible.builtin.add_host`) – `ansible-bender` runs all our tasks on the container image it instantiates using the details from the `ansible_bender` variable structure. Thus, our code should look like this – it is identical to the second play we used before, only we're not running the final `ansible.builtin.shell` task to start the `nginx` web server, as this is taken care of by details in the `ansible_bender` variable:

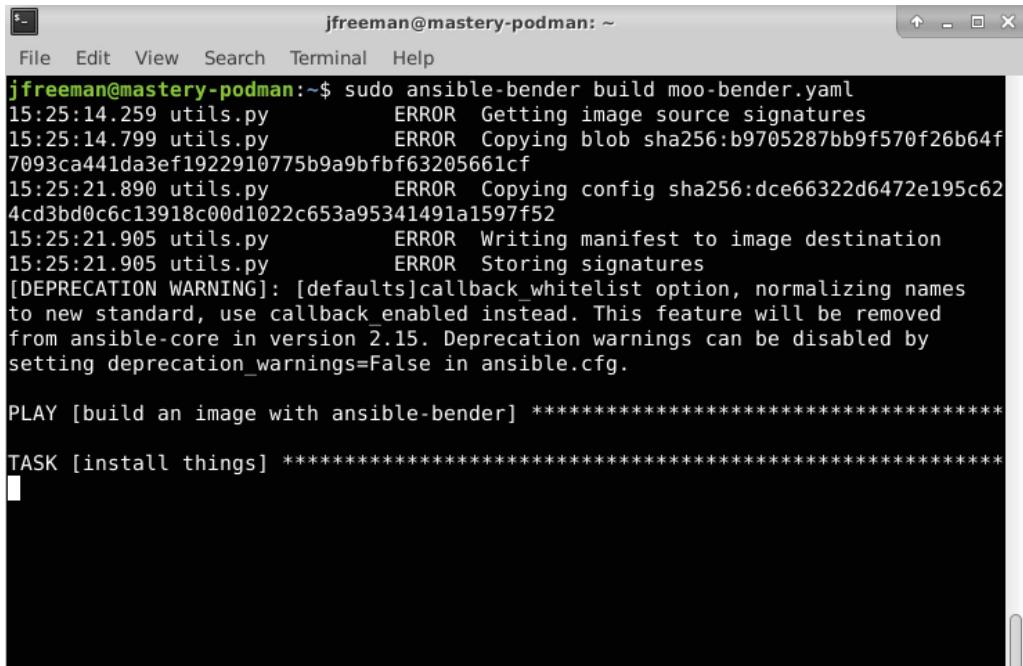
```
tasks:  
  - name: install things  
    ansible.builtin.raw: dnf install -y python-dnf  
  
  - name: install things  
    ansible.builtin.dnf:  
      name: ['nginx', 'cowsay']  
  
  - name: configure nginx  
    ansible.builtin.lineinfile:  
      line: "daemon off;"  
      dest: /etc/nginx/nginx.conf  
  
  - name: boop  
    ansible.builtin.shell: cowsay boop > /usr/share/nginx/  
html/index.html
```

That's it – the code is no more complex than that! Now, building your first container with `ansible-bender` is as simple as running the following command:

```
sudo ansible-bender build moo-bender.yaml
```

Note that the command must be run as root (that is, via `sudo`) – this is a specific related to Buildah and Podman and their behavior when run as an unprivileged user.

One oddity of `ansible-bender` is that when it starts to run, you will see some lines that state `ERROR` (see *Figure 12.21*). This is a bug in `ansible-bender` as these lines are not actually errors – they are simply information being returned from the Buildah tool:



```
jfreeman@mastery-podman:~$ sudo ansible-bender build moo-bender.yaml
15:25:14.259 utils.py      ERROR  Getting image source signatures
15:25:14.799 utils.py      ERROR  Copying blob sha256:b9705287bb9f570f26b64f
7093ca441da3ef1922910775b9a9bfb63205661cf
15:25:21.890 utils.py      ERROR  Copying config sha256:dce66322d6472e195c62
4cd3bd0c6c13918c00d1022c653a95341491a1597f52
15:25:21.905 utils.py      ERROR  Writing manifest to image destination
15:25:21.905 utils.py      ERROR  Storing signatures
[DEPRECATION WARNING]: [defaults]callback_whitelist option, normalizing names
to new standard, use callback_enabled instead. This feature will be removed
from ansible-core in version 2.15. Deprecation warnings can be disabled by
setting deprecation_warnings=False in ansible.cfg.

PLAY [build an image with ansible-bender] ****
TASK [install things] ****
```

Figure 12.21 – Starting the container build process with `ansible-bender`, and the false `ERROR` messages

As the build continues, you should see Ansible playbook messages return in the manner with which you are now familiar. At the end of the process, you should have a successful build indicated by output like that shown in *Figure 12.22*:

```
jfreeman@mastery-podman: ~
File Edit View Search Terminal Help
changed: [fedora-moo-20210807-152500567524-cont]

TASK [configure nginx] ****
changed: [fedora-moo-20210807-152500567524-cont]

TASK [boop] ****
changed: [fedora-moo-20210807-152500567524-cont]

PLAY RECAP ****
fedora-moo-20210807-152500567524-cont : ok=4      changed=4      unreachable=0      failed=0      skipped=0      rescued=0      ignored=0

Getting image source signatures
Copying blob sha256:4c85102d65a59c6d478bfe6bc0bf32e8c79d9772689f62451c7196380675
d4af
Copying blob sha256:0e0fb57c319f8dadfce98035971c0ec0dcff6432a8b05b72c2fdbbf86c0
bca9
Copying config sha256:94de2c723f8343fe0b3d246c907e59f40357e545fb648a6ebf2950cb08
2f9e91
Writing manifest to image destination
Storing signatures
94de2c723f8343fe0b3d246c907e59f40357e545fb648a6ebf2950cb082f9e91
Image 'fedora-moo' was built successfully \o/
jfreeman@mastery-podman:~$
```

Figure 12.22 – A successful container build with ansible-bender

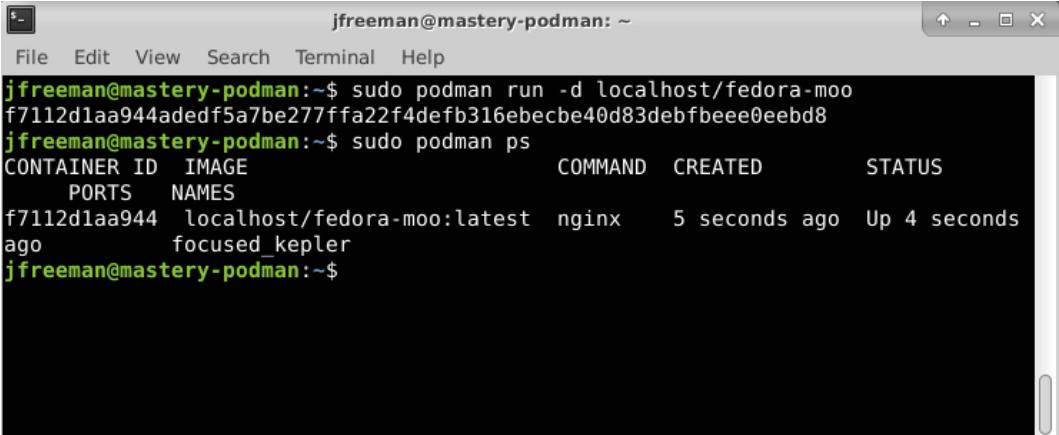
From here, you can run your newly built container with this command:

```
sudo podman run -d fedora-moo
```

The `fedora-moo` container name was set in the `ansible_bender` variable structure in the playbook file previously, whilst the `-d` flag is used to detach from the container and run it in the background. Similar to Docker, you can query the running containers on your system with this command:

```
sudo podman ps
```

The output from this process will look somewhat like that shown in *Figure 12.23*:



A screenshot of a terminal window titled "jfreeman@mastery-podman: ~". The window contains the following text:

```
jfreeman@mastery-podman:~$ sudo podman run -d localhost/fedora-moo
f7112d1aa944adedf5a7be277ffa22f4defb316ebecbe40d83debfbbeee0eebd8
jfreeman@mastery-podman:~$ sudo podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
f7112d1aa944 localhost/fedora-moo:latest nginx 5 seconds ago Up 4 seconds
ago focused_kepler
jfreeman@mastery-podman:~$
```

Figure 12.23 – Running and querying our newly built container in Podman

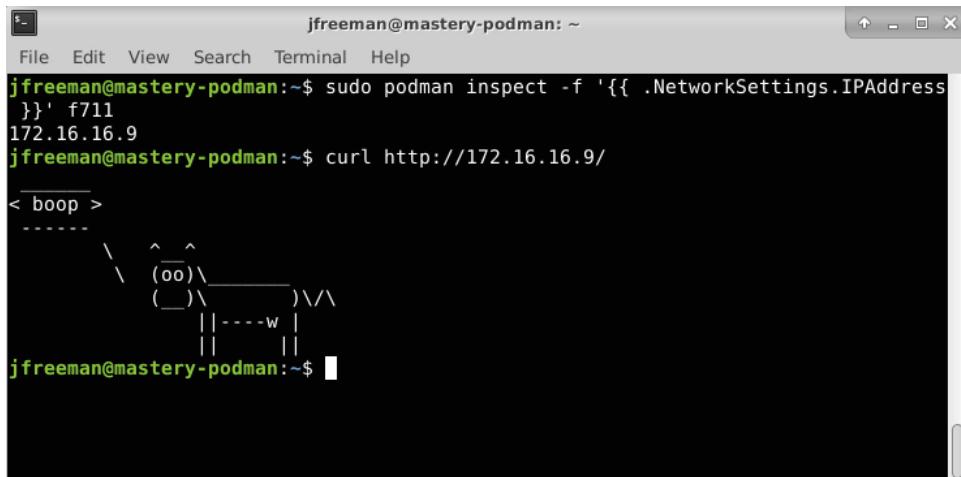
Finally, let's see if we can actually retrieve our `cowsay` web page from our container. Unlike our Docker example, we have not instructed Podman to redirect the web server port to a port on our build machine, so we will need to query the IP address of the container itself. Having obtained the `CONTAINER_ID` or `NAMES` from the output of `sudo podman ps`, we can query this with a command such as this (be sure to replace the container ID with the one from your system):

```
sudo podman inspect -f '{{ .NetworkSettings.IPAddress }}' f711
```

As with Docker, you can abbreviate your container ID provided the characters you enter are unique in the list of running containers. Once you have retrieved the IP address, you can use `curl` to download the web page, just as we did before – for example:

```
curl http://172.16.16.9
```

This whole process should look like that shown in *Figure 12.24*:



```
jfreeman@mastery-podman:~$ sudo podman inspect -f '{{ .NetworkSettings.IPAddress }}' f711
172.16.16.9
jfreeman@mastery-podman:~$ curl http://172.16.16.9/
< boop >
-----
\ ^__^
(oo)\_____
 (__)\       )\/\
    ||----w |
    ||     |
jfreeman@mastery-podman:~$
```

Figure 12.24 – Downloading our cowsay web page from our Podman container build with ansible-bender

That's all there is to it! The `ansible-bender` tool shows great promise in providing an automation framework for building container images with one common language – our own favorite, Ansible! As the tool develops, hopefully, some of the rough edges (such as the false `ERROR` statements) will be resolved, and the addition of support for more container platforms will truly make this a valuable automation tool for container images. That concludes our look at infrastructure provisioning with Ansible – hopefully, you have found it valuable.

## Summary

DevOps has pushed automation in many new directions, including the containerization of applications, and even the creation of infrastructure itself. Cloud computing services enable self-service management of fleets of servers for running services. Ansible can easily interact with these services to provide the automation and orchestration engine.

In this chapter, you learned how to manage on-premises cloud infrastructures, such as OpenStack, using Ansible. We then extended this with examples of public cloud infrastructure provision on both AWS and Microsoft Azure. Finally, you learned how to interact with Docker using Ansible, and how to neatly package Docker service definitions using Ansible Container.

Ansible can start just about any host, except for the one that it is running on, and with proper credentials, it can create the infrastructure that it wants to manage, either for one-off actions or to deploy a new version of an application into a production container management system. The end result is that once your hardware is in place and your service providers are configured, you can manage your entire infrastructure through Ansible, if you so desire!

In the final chapter of this book, we will look at a new and rapidly growing area of automation: network provisioning with Ansible.

## Questions

1. When creating or deleting VM instances on OpenStack, which inventory host should you reference in your play?
  - a) The OpenStack host
  - b) localhost
  - c) The VM Floating IP address
  - d) None of the above
2. How would you reference a newly created virtual machine in a second play without having to use a dynamic inventory script?
  - a) Use `ansible.builtin.raw` commands.
  - b) Use `ansible.builtin.shell` commands.
  - c) Use `ansible.builtin.add_host` to add the new VM to the in-memory inventory.
  - d) You need to use the dynamic inventory plugin.
3. You can still run dynamic inventory scripts directly in Ansible 4.x and newer, just as you could in Ansible 2.x releases.
  - a) True
  - b) False

4. To use a dynamic inventory script, and set its parameters, you would now (assuming the collection is already installed):
  - a) Define a YAML inventory file with the plugin name and parameters.
  - b) Reference the dynamic inventory script in the `-i` parameter of `ansible`/`ansible-playbook`.
  - c) Put the plugin name in your play definition itself.
5. When using a new module from a collection for the first time (for example, with a cloud provider), you should:
  - a) Always read the documentation to check for known issues.
  - b) Always read the documentation to see if you need to install additional Python modules.
  - c) Always read the documentation see how you should define your authentication parameters.
  - d) All of the above.
6. Ansible cannot function on a target host if there is no Python environment (this is sometimes the case on minimal cloud operating system images). If this is the case, you can still install Python from a playbook task with which module?
  - a) `ansible.builtin.python`
  - b) `ansible.builtin.raw`
  - c) `ansible.builtin.command`
  - d) `ansible.builtin.shell`
7. All cloud provider modules will wait for a VM instance to come up before the play is allowed to move on to the next task.
  - a) True
  - b) False
8. If you want to wait to ensure a host is accessible over SSH before you perform additional tasks, you can use which module?
  - a) `ansible.builtin.wait_for`
  - b) `ansible.builtin.ssh`
  - c) `ansible.builtin.test_connection`
  - d) `ansible.builtin.connect`

9. Ansible can build Docker containers both with and without a Dockerfile.
  - a) True
  - b) False
10. The ansible-bender tool currently supports which build environment?
  - a) Docker
  - b) LXC
  - c) Podman/Buildah
  - d) All of the above

# 13

# Network Automation

Historically, a network consisted of mostly hardware with just a modicum of software involvement. Changing the topology of it involved installing and configuring new switches or blades in a chassis or, at the very least, re-patching some cables. Now, the scenario has changed, and the complex infrastructures built to cater for multi-tenant environments such as cloud hosting, or microservice-based deployments, require a network that is more agile and flexible. This has led to the emergence of **Software-Defined Networking (SDN)**, an approach that centralizes the network configuration (where historically it was configured on a per-device basis) and results in a network topology being defined as a whole, rather than as a series of component parts. It is, if you like, an abstraction layer for the network itself and thus implies that just like infrastructure as a service, networks can now be defined in code.

Since the previous edition of this book was published, a great deal of work has gone into Ansible to enhance and standardize network automation within the project. In addition to this, the advent of collections has enabled the modules for many network devices to become decoupled from the `ansible-core` package, thus enabling network vendors to take greater ownership of their code and release them on an as-needed basis, rather than being driven by the cadence of the Ansible releases themselves. At the time of writing, only a handful of Ansible collections (and thus modules) remain under the remit of the Ansible Network team, with most now being maintained directly by the vendors themselves. This is a good thing for all concerned, and it ensures greater reliability and more rapid development of Ansible's network offering.

Ultimately, this means one thing – that you can now define your network infrastructure in an Ansible playbook, just as you can describe your compute infrastructure, as we described in the previous chapter.

In this chapter, we will explore this area of rapidly growing importance by covering the following topics:

- Ansible for network management
- Handling multiple device types
- Working with the `cli_command` module
- Configuring Arista EOS switches with Ansible
- Configuring Cumulus Networks switches with Ansible
- Best practices

## Technical requirements

To follow the examples presented in this chapter, you will need a Linux machine running **Ansible 4.3** or newer. Almost any flavor of Linux should do – for those interested in specifics, all the code presented in this chapter was tested on **Ubuntu Server 20.04 LTS**, unless stated otherwise, and on Ansible 4.3. The example code that accompanies this chapter can be downloaded from GitHub at this URL: <https://github.com/PacktPublishing/Mastering-Ansible-Fourth-Edition/tree/main/Chapter13>.

Check out the following video to see the Code in Action:

<https://bit.ly/3G5pNjJ>.

## Ansible for network management

Core network devices, such as switches, routers, and firewalls, have long had management interfaces, especially in enterprise environments. **Command-Line Interfaces (CLIs)** have always been popular on such devices as they support scripting, so, as you have already guessed, they lend themselves extremely well to Ansible automation.

Historically, teams have faced a myriad of challenges when managing these devices, including maintaining configuration, coping with the failure/loss of a device, and obtaining support in the event of an issue. Often, companies found themselves locked into a single network vendor (or at best, a small handful) to enable the use of proprietary tools to manage the network. As with any situation where you are locked into a technology, this carries both benefits and drawbacks. Add to this the complexity of software-defined networks that are rapidly changing and evolving, and the challenge becomes even greater. In this section, we will explore how Ansible addresses these challenges.

## Cross-platform support

As we have seen throughout this book, Ansible has been designed to make automation code portable and reusable in as many scenarios as possible. In *Chapter 12, Infrastructure Provisioning*, we used almost identical playbooks to configure infrastructure on four different providers, and to support this, the examples that were given were quite simplistic. Of course, we could have developed this further through the use of roles to remove the repetition of so much code if we had wished, but the simplicity was deliberate to demonstrate how similar the code was, regardless of the provider being used.

In short, Ansible made it possible to write playbooks that ran on multiple environments to achieve the same thing with minimal effort once we had defined the first one. The same is true of networks. The advent of collections means that there is no central network module index anymore, as the collections themselves define which platforms are supported. However, the *Ansible for Network Automation* page, available at <https://docs.ansible.com/ansible/latest/network/index.html>, is a great place to get started with all the basic concepts as it provides a list of many of the supported platforms. However, the list of platforms on this page is not complete – for example, later in this chapter, we will look at configuring a switch based on the Cumulus Linux platform, and support for this is not explicitly listed on the aforementioned page.

Part of the reason for this is that support for Cumulus Linux and a wide array of other network technologies are supported by the `Community.Network` collection. The list of supported platforms and modules can be found here: <https://docs.ansible.com/ansible/latest/collections/community/network/>.

As the Ansible documentation is automatically built, decentralizing modules into collections has been a little disruptive in areas such as networking, and will no doubt improve over time. In the meantime, with a little searching, you are sure to find support for your network platform as this support has only been expanding as Ansible has developed.

The result is that with such a wide (and growing) range of device support, it is easy for a network administrator to manage all of their devices from one central place, without the need for proprietary tools. However, the benefits are greater than just this.

## Configuration portability

As we have discussed already, Ansible code is highly portable. In the world of network automation, this is extremely valuable. To start with, it means that you could roll out a configuration change on a development network (or simulator) and test it, and then be able to roll out the same code against a different inventory (for example, a production one) once the configuration has been deemed to have been tested successfully.

The benefits don't stop there, however. Historically, in the event of issues with a software upgrade or configuration change, the network engineer's challenge was to engage the vendor for support and assistance successfully. This required sending sufficient detail to the vendor to enable them to at least understand the problem and most likely want to reproduce it (especially in the case of firmware issues). When the configuration for a network is defined in Ansible, the playbooks themselves can be sent to the vendor to enable them to quickly and accurately understand the network topology and diagnose the issue. I have come across cases where network vendors are now starting to insist on Ansible playbooks containing network configuration when a support ticket is raised. This is because it empowers them to resolve the issue faster than ever before.

Effective use of **Ansible Vault** ensures that sensitive data is kept out of the main playbooks, which means it can easily be removed before being sent to a third party (and even if it was sent accidentally, it wouldn't be readable as it is encrypted at rest).

## Backup, restore, and version control

Although most businesses have robust change control procedures, there is no guarantee that these are followed 100% of the time, and human beings have been known to tweak configurations without accurately recording the changes they've made. Moving the network configuration to Ansible removes this issue, as the configuration is a known state defined by the playbooks that can be compared easily to the running configuration using a check run.

Not only is this possible, but configurations can be backed up and restored with ease. Say, for example, a switch fails and has to be replaced. If the replacement is of the same type, it can be configured and brought into service rapidly by running the same Ansible playbooks that configured its predecessor, with the playbook run perhaps limited to just the replacement switch inventory host if appropriate – though Ansible's idempotent nature means that running it across the entire network should be benign.

This lends itself to version control too – network configuration playbooks can be pushed to a source control repository, enabling configuration versions to be tracked, and differences over time to easily be examined.

## Automated change requests

Often, minor changes to a network might be required to roll out a new project – perhaps a new VLAN or VXLAN, or some previously unused ports that have been brought into the service. The configuration parameters will be well-defined by a change request and/or network design, and it is probably not the best use of a highly qualified network engineer to be making simple configuration changes. Tasks such as these are typically routine, in that the configuration changes can be templated in an Ansible playbook, with variables passed to it that have been defined by the change request (for example, port numbers and VLAN membership details).

This then frees up the engineers' time for more important work, such as designing new architectures, and new product research and testing.

Coupled with the use of a package such as AWX or Ansible Tower (as we discussed earlier in this book), simple and well-tested changes could be completely automated or passed to a frontline team to be executed safely by simply passing in the required parameters. In this way, the risk of human error is significantly reduced, regardless of the skillset of the person performing the change.

With these benefits well established and understood, let's look at how we might start writing playbooks to handle a multi-device network.

## Handling multiple device types

In a world where we are not locked into a single vendor, it is important to know how we might handle the different network devices in an infrastructure. We established in the previous chapter that for different infrastructure providers, a similar process was established for each one in terms of getting Ansible to interact with it. This can be a little different with switches as not all command-line switch interfaces are created the same. Some, such as on a Cumulus Networks switch, can make use of straightforward SSH connectivity, meaning that everything we have learned about in this book so far on connecting to an SSH-capable device still applies.

However, other devices, such as F5 BIG-IP, do not use such an interface and therefore require the module to be run from the Ansible host. The configuration parameters must be passed to the module directly as opposed to using simple connection-related host variables such as `ansible_user`.

There is, of course, a gray area in the middle of this discussion. Some devices, such as an Arista EOS or Cisco IOS-based device will be SSH managed, so you could be mistaken for thinking you can connect to them using a straightforward SSH connection as if they were any other Linux host. However, this is not the case – if we reflect on *Chapter 1, The System Architecture and Design of Ansible*, we learned that for Ansible to automate commands over SSH, it sends over a tiny chunk of Python code to the remote host for execution (or PowerShell, in the case of Windows hosts). Most switches, while having an SSH-based user interface, cannot be expected to have a Python environment present on them, so this mode of operation is not possible (Cumulus Linux is the exception here since it features a usable Python environment). As a result, devices such as the Arista EOS and Cisco IOS ones historically used to use local execution, whereby the Ansible code is run on the control node itself, and then the automation requests are translated into the appropriate CLI (or API) calls and passed directly to the device. Thus, no remote Python environment is required.

You will find many historical examples that make use of this mode of operation, and they can easily be identified as they will have the following line somewhere in the play definition:

```
connection: local
```

Possibly, this might also be defined as an inventory variable:

```
ansible_connection=local
```

Regardless of how this happens, the local connection mode of operation has been deprecated, and while most legacy networking playbooks that utilize this connection mode will still run today, it is anticipated that support for this will be dropped next year.

Where possible, users are encouraged to use one of the following communication protocols instead:

- `ansible.netcommon.network_cli`: This protocol translates play tasks into CLI commands over SSH.
- `ansible.netcommon.netconf`: This protocol translates play tasks into XML data sent over SSH to the device for configuration by netconf.
- `ansible.netcommon.httpapi`: This protocol talks to network devices using an HTTP or HTTPS-based API.

The three preceding communication protocols are all persistent – that is to say, they don't need to set up and tear down network connections for each task – the local connection method does not support this, so it is significantly less efficient than these modes. Of the preceding list, `ansible.netcommon.network_cli` is the most common you will come across, and we will look at this in the next section.

It is not expected that many of you will have access to a wide variety of network hardware to use in the examples in this chapter. Later, we will look at two examples that are freely available to download (at the time of writing, subject to you sharing a little personal information) that you can try out if you wish. For now, though, we will go into more detail on the process to be employed when automating a new network device for the first time so that you know to apply this to a situation and preferred network vendor.

## Researching your modules

Your first task when working with any networking device is to understand what module you need to use with Ansible. This will involve two things:

- What device do you wish to automate the management of?
- What task(s) do you wish to perform on the device?

Armed with this information, you can search the Ansible documentation site and Ansible Galaxy to find out if your devices and desired tasks are supported. Let's say, for example, that you have an F5 BIG-IP device, and you want to save and load configuration on this device.

A quick scan of the available collections on Ansible Galaxy suggests we should look at the `f5networks.f5_modules` collection ([https://galaxy.ansible.com/f5networks/f5\\_modules](https://galaxy.ansible.com/f5networks/f5_modules)), and that from this, we should look into the `f5networks.f5_modules.bigip_config` module, which will do just what we need. Thus, we can proceed with the module configuration (see the next section) and then write the desired playbook around this module.

What happens if there is no module for your device, though? In this instance, you have two choices. Firstly, you could write a new module for Ansible to perform the tasks you require. This is something you could contribute back to the community, and *Chapter 10, Extending Ansible*, contains all the details you need to get started on this task.

Alternatively, if you want to get something up and running quickly, remember that Ansible can send raw commands in most of the transport methods it supports. For example, in the author's lab setup, they have a TP-Link managed switch. There are no native Ansible modules that support this particular switch – however, as well as a web-based GUI, this switch also has an SSH management interface. If I wanted to quickly get something up and running, I could use Ansible's `ansible.builtin.raw` module to send raw commands over SSH to the switch. Naturally, this solution lacks elegance and makes it difficult to write playbooks that are idempotent, but it does enable me to get up and running quickly with Ansible and this device.

This captures the beauty of Ansible – the ease with which new devices can be managed, and how, with just a little ingenuity, it can be extended for the benefit of the community.

## Configuring your modules

As we have already demonstrated the use of the `ansible.builtin.raw` module, as well as extending Ansible, earlier in this book, we will proceed with the case where we have found a module we want to work with. As you may have noticed in some of the earlier chapters in this book, although Ansible includes many modules out of the box, not all of them work straight away.

Ansible is written in Python and, in most cases, where there are dependencies, there will be Python modules. The important thing is to review the documentation. For example, take the `f5networks.f5_modules.bigip_config` module we selected in the previous section. A quick review of the *Requirements* section of the documentation shows that this requires (at the time of writing) the `ipaddress` Python module if you are running a Python version older than 3.5.

If you are not running Python 3.5 or later, you will need to install this for the collection's modules to function correctly. There are multiple ways to install this – some operating systems may have a native package built, and if this is available, then provided it meets the version requirements, it is perfectly fine to use this. This may be advantageous in terms of vendor support. However, if such a package is not available, Python modules can easily be installed using the pip (or pip3) tool. Assuming this is already on your system, the installation is as simple as using the following code:

```
sudo pip install ipaddress
```

Also, be sure to review the *Notes* section of the documentation (for the module we are currently discussing, go to [https://clouddocs.f5.com/products/orchestration/ansible-devel/modules/bigip\\_config\\_module.html#notes](https://clouddocs.f5.com/products/orchestration/ansible-devel/modules/bigip_config_module.html#notes)). Continuing with this example, we can see that it only supports BIG-IP software version 12 and newer, so if you are on an earlier version, you will have to find another route to automate your device (or upgrade the software if this is an acceptable path).

## Writing your playbooks

Once your modules have been configured and all the requirements (be they Python module dependencies or device software ones) have been met, it's time to start writing your playbook. This should be a simple task of following the documentation for the module. Let's suppose we want to reset the configuration on an **F5 BIG-IP** device. From the documentation, we can see that authentication parameters are passed to the module itself. Also, the example code shows the use of the `delegate_to` task keyword; both of these clues tell us that the module is not using the native SSH transport of Ansible, but rather one that is defined in the module itself. Thus, a playbook to reset the configuration of a single device might look something like this:

```
---
- name: reset an F5
  hosts: localhost
  gather_facts: false

  tasks:
    - name: reset my F5
      f5networks.f5_modules.bigip_config:
        reset: yes
        save: yes
```

```
provider:  
  server: lb.mastery.example.com  
  user: admin  
  password: mastery  
  validate_certs: no
```

In this case, we are using a textbook example from the documentation to reset our configuration. Note that as our hosts parameter only defines localhost, we do not need the delegate\_to keyword, since the f5networks.f5\_modules.bigip\_config module will only be run from localhost in this playbook.

In this way, we have automated a simple, but otherwise manual and repetitive, task that might need to be performed. Running the playbook would be as simple as executing the following command:

```
ansible-playbook -i mastery-hosts reset-f5.yaml
```

Naturally, to test this playbook, you would have to have an F5 BIG-IP device to test against. Not everyone will have this available, so, later in this chapter, we will move on to demonstrate real-world examples that everyone reading this book can work with. However, this part of this chapter is intended to give you a solid overview of integrating your network devices, whatever they may be, with Ansible. Thus, it is hoped that even if you have a device that we haven't mentioned here, you understand the fundamentals of how to get it working.

## Working with the cli\_command module

Before we get to the practical hands-on examples, we must look at a module that has become central to network device configuration since the previous edition of this book was published.

As we discussed in the preceding section, most network devices cannot be expected to have a working Python environment on them, and as such, Ansible will use local execution – that is to say, all tasks related to network devices are executed on the Ansible control node itself, translated into the correct format for the device to receive (be that a CLI, an HTTP-based API, or otherwise), and then sent over the network to the device. Ansible 2.7 relied mostly on a communication protocol known as `local` for network device automation. This worked well but suffered from several drawbacks, including the following:

- The `local` protocol does not support persistent network connections – a new connection needs to be set up and then torn down for each task that's executed. This is hugely inefficient and slow, and not in line with the original vision for Ansible at all.
- Each module was responsible for its own communication protocol, so the library requirements for each module were often different, and code was not being shared.
- There was little commonality in the manner in which credentials could be provided for network device communication, and credentials had to be provided in each task, again resulting in inefficient code.

As a result of these issues, the `local` protocol is expected to be dropped from Ansible within the next year, and you are encouraged to start using one of the three new protocols listed in the *Handling multiple device types* section, earlier in this chapter. Of these, the most common is the `ansible.netcommon.network_cli` protocol, which can be used to connect to a great deal of the network devices you may wish to automate with Ansible – you can see how common the use of this module is for network device configuration by looking at the table provided at [https://docs.ansible.com/ansible/latest/network/user\\_guide/platform\\_index.html#settings-by-platform](https://docs.ansible.com/ansible/latest/network/user_guide/platform_index.html#settings-by-platform).

The beauty of this protocol is that authentication parameters can now be set in the inventory, just like they can for any other operating system, simplifying playbooks and removing the need for repetitive credentials being set. Persistent connections are also supported, meaning a much faster automation run. So, how does it work?

Well, let's suppose we have a Cisco IOS-based network device to configure. We could define a simple inventory file that looks like this:

```
[ios_devices]
ios-switch1.example.org

[ios_devices:vars]
ansible_connection: ansible.netcommon.network_cli
ansible_network_os: cisco.ios.ios
ansible_user: admin
ansible_password: password123
ansible_become: yes
ansible_become_method: enable
ansible_become_password: password123
```

Notice how easy that is? We set the same `ansible_user`, `ansible_password`, and `ansible_become` inventory variables that we have already seen throughout the examples in this book. However, we have added the `ansible_connection` variable here, which tells Ansible to make use of the `ansible.netcommon.network_cli` protocol. Of course, this is only half of the story – this protocol tells Ansible to send CLI commands over SSH but doesn't tell Ansible what device type is on the other end of the connection. As all CLIs are different in some way, this matters, so we use `ansible_network_os` to tell Ansible what type of device it's talking to so that it can speak the right CLI language to the device.

Finally, we need to change the `ansible_become` method – on Linux, this would almost certainly be `sudo`, but on an IOS switch, it is `enable`. We also need to provide the password for the elevation of rights, just as you would if you had `sudo` configured to require a password.

That's as complex as it gets – a simple playbook that makes use of this inventory and its assigned variables might look like this:

```
---
- name: Simple IOS example playbook
  hosts: all
  gather_facts: no

  tasks:
    - name: Save the running config
```

```
cisco.ios.ios_config:  
    save_when: always
```

Notice how easy that is – we can now write playbooks in the same manner as when we work with Linux or Windows hosts. Of course, each networking platform has subtle differences, and the platform-specific options for a myriad of supported devices can be found here: [https://docs.ansible.com/ansible/latest/network/user\\_guide/platform\\_index.html](https://docs.ansible.com/ansible/latest/network/user_guide/platform_index.html).

The other great thing about using the `ansible.netcommon.network_cli` protocol is that it supports the `ansible_ssh_common_args` inventory variable, just as any other SSH managed host (Linux or Windows with OpenSSH) would. This is important because many network devices are managed over a secure, isolated network – and given the damage that could be caused if access to this fell into the wrong hands, rightly so. This means that these hosts are often accessed using a bastion host (also known as a jump host). To run your automation playbooks via this bastion, you can add the following to your inventory variables:

```
ansible_ssh_common_args: '-o ProxyCommand="ssh -W %h:%p -q  
jumphost.example.org"'
```

The preceding example assumes your bastion or jump host has a hostname of `jumphost.example.org` and that you have already set up passwordless key-based SSH access to it from your Ansible control node. There are other ways to authenticate with this bastion host, of course, and this is left as an exercise for you to explore.

Of course, this is just one example, and Cisco IOS-based devices are not something that everyone reading this book will have access to. However, at the time of writing, you can easily and freely download the Arista vEOS images by signing up for a free account on their website and heading over to <https://www.arista.com/en/support/software-download>. From here, you can load these images into a network simulation tool such as GNS3 and experiment with Ansible's network automation for yourself, without needing access to any expensive hardware. We'll look at this in the next section.

## Configuring Arista EOS switches with Ansible

Getting up and running with an Arista switch (or virtual switch) is left as an exercise for you, but if you are interested in doing this in GNS3, a popular and freely available open source tool for learning about networks, there is some excellent guidance here: <https://gns3.com/marketplace/appliances/arista-veos>.

You might be lucky enough to have an Arista EOS-based device at your fingertips, and that's fine too – the automation code in this section will work equally well in either case.

The following examples were created against an Arista vEOS device in GNS3, created using the instructions found in the aforementioned link. Upon booting the device for the first time, you will need to cancel ZeroTouch provisioning. To do this, log in with the admin username (the password is blank by default) and enter the following command:

```
zerotouch cancel
```

The virtual device will reboot, and when it comes up again, log in using the same credentials. Enter the following command to enter privileged user mode:

```
enable
```

Note that this is the `ansible_become` method that we used in the Cisco IOS example earlier, and we'll be using the same again here shortly. Now, enter configuration mode with the following command:

```
configure terminal
```

You cannot administer a vEOS device over SSH if it has a blank password by default, so we'll set a simple password for our virtual device with the following command:

```
username admin secret admin
```

This sets the password for the admin user to admin. Next (assuming you have wired up the management interface of the vEOS device to your virtual network), you will need to enable this interface and give it a valid IP address. The exact IP address will depend on your test network, but the commands to achieve this would look like this:

```
interface management 1
no shutdown
ip address 10.0.50.99/8
```

Finally, exit configuration mode and write the configuration to the switch so that it comes up again on the next reboot:

```
end
write
```

That's it – your vEOS device is now ready for management with Ansible!

With this configuration in place, you can now define an inventory for your test switch. I created mine as follows (based on the preceding configuration):

```
[eos]
mastery-eos ansible_host=10.0.50.99

[eos:vars]
ansible_connection=ansible.netcommon.network_cli
ansible_network_os=arista.eos.eos
ansible_user=admin
ansible_password=admin
ansible_become=yes
ansible_become_method=enable
```

Notice how similar this is to the Cisco IOS-based example inventory we created in the previous section? This is one of the great things about the `ansible.netcommon.network_cli` protocol – all your code is much easier to write when using this protocol. Of course, as with most of the examples in this book, you wouldn't leave your administrative password out in the clear, it but serves to keep the examples simple and concise, and you are encouraged to explore the use of Ansible Vault for storing them safely.

From here, we can develop a simple playbook to demonstrate command automation against our virtual switch. Let's pick a simple task – we'll ensure that the `Ethernet1` interface on the switch is enabled, give it a meaningful name, and then write the config to the switch so that it persists across reboots. A playbook that achieves this might look something like this:

```
---
- name: A simple play to enable Ethernet1 on our virtual switch
  and write the config
  hosts: all
  gather_facts: no

  tasks:
    - name: Enable Ethernet1 on the switch
      arista.eos.eos_interfaces:
        config:
          - name: Ethernet1
```

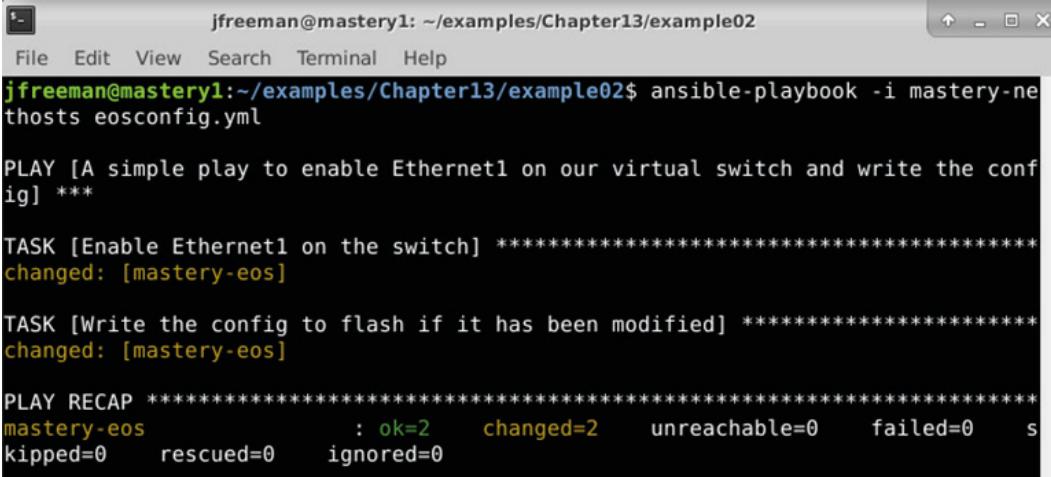
```
enabled: yes
description: Managed by Ansible
state: replaced

- name: Write the config to flash if it has been modified
  arista.eos.eos_config:
    save_when: modified
```

You can run this playbook in the same way you are used to. If you're running the example code that accompanies this book, the command for this would be as follows:

```
ansible-playbook -i mastery-nethosts eosconfig.yml
```

When you run this against your switch, you should see an Ansible output similar to the following:



The screenshot shows a terminal window titled 'jfreeman@mastery1: ~/examples/Chapter13/example02'. The user runs the command 'ansible-playbook -i mastery-nethosts eosconfig.yml'. The output shows the execution of a play to enable Ethernet1 and write the configuration, resulting in two tasks being changed on the 'mastery-eos' host. The final PLAY RECAP summary indicates 2 ok, 2 changed, 0 unreachable, 0 failed, 0 skipped, 0 rescued, and 0 ignored.

```
jfreeman@mastery1:~/examples/Chapter13/example02$ ansible-playbook -i mastery-nethosts eosconfig.yml

PLAY [A simple play to enable Ethernet1 on our virtual switch and write the config] ***

TASK [Enable Ethernet1 on the switch] *****
changed: [mastery-eos]

TASK [Write the config to flash if it has been modified] *****
changed: [mastery-eos]

PLAY RECAP *****
mastery-eos : ok=2     changed=2     unreachable=0     failed=0     skipped=0    rescued=0    ignored=0
```

Figure 13.1 – Configuring an Arista vEOS device with Ansible

Now, of course, since we are performing this configuration change with Ansible, we expect the change to be idempotent – we should be able to run the same playbook again and nothing disruptive will happen. If you run your playbook a second time, the output should look like this:

```
jfreeman@mastery1: ~/examples/Chapter13/example02
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter13/example02$ ansible-playbook -i mastery-ne thosts eosconfig.yml

PLAY [A simple play to enable Ethernet1 on our virtual switch and write the config] ***

TASK [Enable Ethernet1 on the switch] *****
ok: [mastery-eos]

TASK [Write the config to flash if it has been modified] *****
ok: [mastery-eos]

PLAY RECAP *****
mastery-eos : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Figure 13.2 – Running the same playbook again to demonstrate idempotency

As shown by the green `ok` task statuses, this playbook has run successfully a second time and that this time around, no changes were made to the switch configuration.

If you choose to, you can validate the results of our playbook run by SSHing directly into the switch and executing the following commands:

```
enable
show running-config
```

From here, you should see something like the following:

```
mastery-eos#
File Edit View Search Terminal Help
!
! Command: show running-config
! device: mastery-eos (vEOS-lab, EOS-4.26.2F)
!
! boot system flash:/vEOS-lab.swi
!
no aaa root
!
username admin role network-admin secret sha512 $6$BdLZSEfc/ay.qzJP$gt9XCUI16fnE
QZOPzp2oFDJvrN0MbPalj0dzd97ToovgBNq.NS0S6oT1qVC.r.c0EFviy852rXHTcTftkL0.1
!
transceiver qsfp default-mode 4x10G
!
service routing protocols model ribd
!
hostname mastery-eos
!
spanning-tree mode mstp
!
interface Ethernet1
  description Managed by Ansible
!
interface Ethernet2
!
--More--
```

Figure 13.3 – Querying the configuration of our vEOS device manually

Here, we can see that the `Ethernet1` interface has the description we set in our playbook and has no directive to disable it, thus ensuring it is enabled.

That's it – by working through this example, you've just performed your first real-life network device automation in Ansible! Hopefully, this shows you that, particularly now, given the advent of the `ansible.netcommon.network_cli` protocol, it is very easy and quick to achieve the configuration you desire. Most devices that support this protocol will work similarly, and you are encouraged to explore this further if this is of interest to you. However, what if we want to work with another device? Well, Cumulus Linux (now owned by NVIDIA) is an open source operating system for network devices that can run on white box hardware – that is to say, it is not proprietary to any specific hardware. Fortunately, you can freely download a copy of Cumulus VX, a virtual version of their switch operating system to experiment with. We'll look at how Ansible can automate this network platform in the next section.

## Configuring Cumulus Networks switches with Ansible

Cumulus Linux (created by Cumulus Networks, which was acquired by NVIDIA) is an open source network operating system that can run on a variety of bare metal switches, offering an open source approach to data center networking. This is a great leap forward for network design and a significant shift away from the proprietary models of the past. They offer a free version of their software that will run on the hypervisor of your choice for test and evaluation purposes called Cumulus VX. The examples in this section are based on Cumulus VX version 4.4.0.

### Defining our inventory

A quick bit of research shows us that Cumulus VX will use the standard SSH transport method of Ansible. Since it is a Linux distribution designed specifically to run on switch hardware, it is capable of running in remote execution mode, so it does not require the `ansible.netcommon.network_cli` protocol. Furthermore, just one module has been defined for working with this system, `network.cumulus.nclu`, which is part of the `community.network` collection (<https://galaxy.ansible.com/community/network>). No prerequisite modules are required to use this module, so we can proceed straight to defining our inventory.

By default, Cumulus VX boots up with the management interface that's been configured to get an IP address with DHCP. Depending on how you run it, you may find that it also has three other virtual switch ports for us to test and play with its configuration, though if you integrate it into a tool such as GNS3, you will find that you can easily reconfigure the number of virtual switch ports available to you.

If you run a version of Cumulus Linux older than 3.7, you will find that the image has default login credentials already set. So, provided you establish the IP address of your virtual switch, you can create a simple inventory such as the following one, which uses the default username and password:

```
[cumulus]
mastery-switch1 ansible_host=10.0.50.110

[cumulus:vars]
ansible_user=cumulus
ansible_ssh_pass=CumulusLinux!
```

Newer releases of Cumulus Linux, such as version 4.4.0 – the latest available at the time of writing and used for the examples in this section – require you to set the password for the switch upon first boot. If you are working with this version, you will need to boot the virtual switch for the first time, and then log in with the default username of `cumulus` and the default password of `cumulus`. You will then be prompted to change your password.

With this done, you are ready to automate your switch configuration. You can use the inventory we defined previously and simply replace the `ansible_ssh_pass` value with the password that you set.

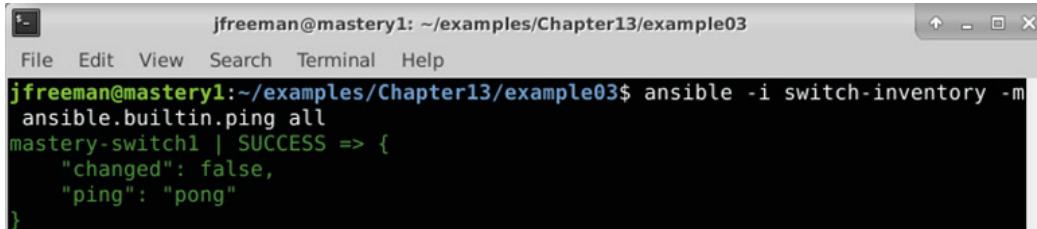
Note the following:

- The IP address specified in `ansible_host` will almost certainly differ from mine – make sure you change this to the correct value for your Cumulus VX virtual machine. You might have to log into the VM console to get the IP address with a command such as `ip addr show`.
- Once again, you would never put the password in clear text in the inventory file – however, for simplicity and to save time, we will specify the default password here. In a real-world use case, always use a vault, or set up key-based SSH authentication.

With the inventory defined, let's test connectivity with the `ping` module using an ad hoc command, like so:

```
ansible -i switch-inventory -m ansible.builtin.ping all
```

If all is set up correctly, you should receive the following output:

A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter13/example03". The window shows a command being run: "ansible -i switch-inventory -m ansible.builtin.ping all". The output of the command is displayed below the command line, showing a single host named "mastery-switch1" with a status of "SUCCESS". The output is formatted as a JSON object with fields "changed": false and "ping": "pong".

```
jfreeman@mastery1:~/examples/Chapter13/example03$ ansible -i switch-inventory -m ansible.builtin.ping all
mastery-switch1 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Figure 13.4 – Checking Ansible connectivity to our virtual Cumulus Linux switch

As we discussed in *Chapter 1, The System Architecture and Design of Ansible*, the `ansible.builtin.ping` module performs a complete end-to-end connectivity test, including authentication at the transport layer. As a result, if you received a successful test result like the one shown previously, we can proceed with confidence and write our first playbooks.

## Practical examples

The Cumulus VX image comes completely unconfigured (save for the DHCP client configuration on the `eth0` management port). Depending on the version you download, it may have three switch ports labeled `swp1`, `swp2`, and `swp3`. Let's query one of those interfaces to see whether there is any existing configuration. We can use a simple playbook called `switch-query.yaml` to query `swp1`:

```
---
- name: query switch
  hosts: mastery-switch1

  tasks:
  - name: query swp1 interface
    community.network.nclu:
      commands:
        - show interface swp1
    register: interface
```

```
- name: print interface status
  ansible.builtin.debug:
    var: interface
```

Now, let's say we run this with the following command:

```
ansible-playbook -i switch-inventory switch-query.yaml
```

We should see something like the following:

```
jfreeman@mastery1: ~/examples/Chapter13/example03
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter13/example03$ ansible-playbook -i switch-inventory switch-query.yaml

PLAY [query switch] ****
TASK [Gathering Facts] ****
ok: [mastery-switch1]

TASK [query swp1 interface] ****
ok: [mastery-switch1]

TASK [print interface status] ****
ok: [mastery-switch1] => {
  "interface": {
    "changed": false,
    "failed": false,
    "msg": "          Name  MAC                Speed  MTU  Mode\n-----\n-----\nADMDN  swp1  0c:27:59:86:00:01  N
9216  NotConfigured\nRouting\n-----\n  Interface swp1 is down\n  Link
ups:      0   last: (never)\n  Link downs:   13   last: 2021/08/31 13:55:36.
24\n  PTM status: disabled\n  vrf: default\n  index 3 metric 0 mtu 9216 speed 10
00\n  flags: <BROADCAST,MULTICAST>\n  Type: Ethernet\n  HWaddr: 0c:27:59:86:00:0
1\n  Interface Type Other\n  protodown: off\n\n"
  }
}

PLAY RECAP ****
mastery-switch1 : ok=3    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
```

Figure 13.5 – Querying the defaults of a switch port on Cumulus Linux with Ansible

This confirms our initial statement about the VM image – we can see that the switch port is not configured. It is very easy to turn this VM into a simple layer 2 switch with Ansible and the `community.network.nclu` module. The following playbook, called `switch-12-configure.yaml`, does exactly this:

```
---
- name: configure switch
```

```
hosts: mastery-switch1

tasks:
  - name: bring up ports swp[1-3]
    community.network.nclu:
      template: |
        {% for interface in range(1,3) %}
        add interface swp{{interface}}
        add bridge ports swp{{interface}}
        {% endfor %}
      commit: true

  - name: query swp1 interface
    community.network.nclu:
      commands:
        - show interface swp1
    register: interface

  - name: print interface status
    ansible.builtin.debug:
      var: interface
```

Notice how we are using some clever inline Jinja2 templating to run a `for` loop across the three interfaces, saving the need to create repetitive and cumbersome code. These commands bring up the three switch interfaces and add them to the default layer 2 bridge.

This also demonstrates the differences between the various networking modules available in Ansible. In the previous section, where we configured our EOS-based switch, we had numerous different modules to work with, each serving a different purpose – for example configuring interfaces, configuring routing, and configuring VLANs. In contrast, Cumulus Linux-based switches only have one module: `community.network.nclu`. This is not an issue, but as we are sending all our configuration commands through one module, making use of Jinja2 templates (which can support constructs such as `for` loops) serves us well.

Finally, the `commit: true` line applies these configurations immediately to the switch. Now, let's say we run this with the following command:

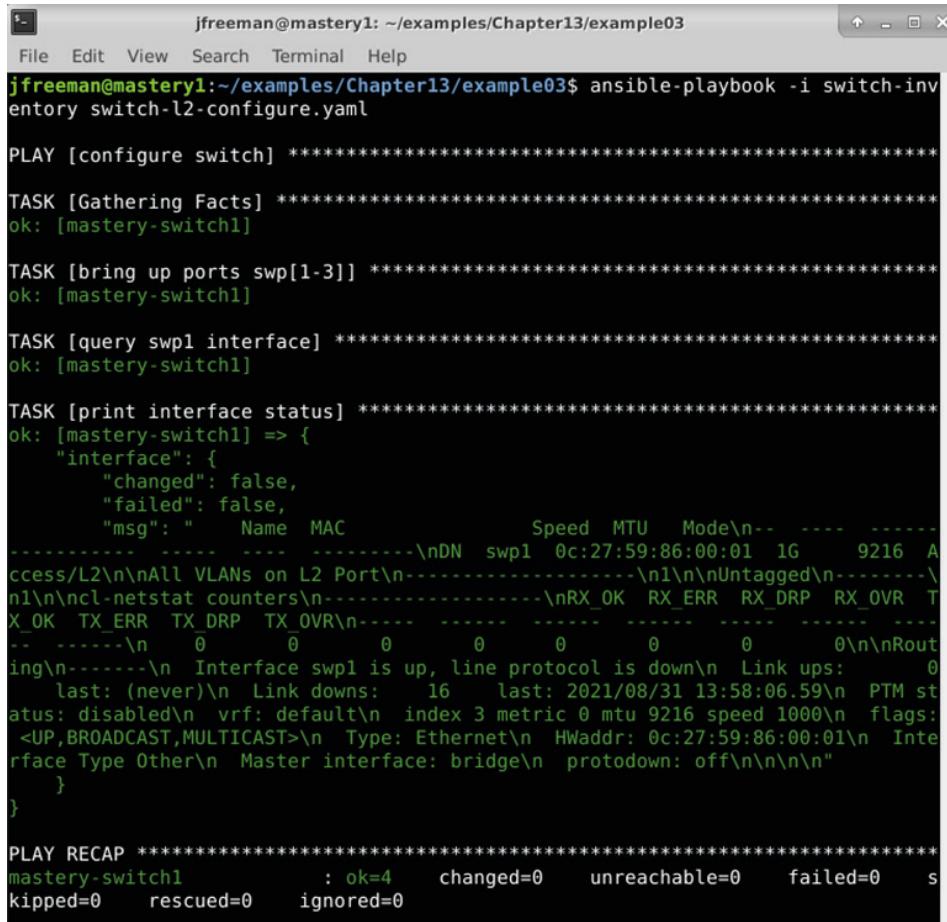
```
ansible-playbook -i switch-inventory switch-12-configure.yaml
```

At this point, we will see a different status for `swp1`, as shown here:

Figure 13.6 – Successfully configuring our Cumulus Linux virtual switch with Ansible

As we can see, the `swp1` interface is now up and part of the bridge, ready to switch traffic. However, if you look closely at the preceding screenshot, you'll see that the `bring_up_ports swp[1-3]` task has a status of `ok`, rather than `changed`. Yet we can see from the `switch` query results that the configuration was changed. This appears to be a bug in version `3.0.0` of the `community.network` collection and has been raised by the author. Hopefully, newer releases of this collection will correctly show the `changed` status when the configuration of Cumulus switches is changed.

Since we're querying the port's status, though, we can still run the playbook a second time to test idempotency. Let's see what happens if we run the playbook again without performing any other steps on the switch:



The screenshot shows a terminal window titled "jfreeman@mastery1: ~/examples/Chapter13/example03". The command run is "ansible-playbook -i switch-inventory switch-l2-configure.yaml". The output shows the playbook running through several tasks: "Gathering Facts", "bring up ports swp[1-3]", "query swp1 interface", and "print interface status". The "print interface status" task outputs detailed information about the swp1 interface, including MAC address, speed, MTU, and various counters. The "PLAY RECAP" section at the end shows the task results for the "mastery-switch1" host.

```
jfreeman@mastery1: ~/examples/Chapter13/example03
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter13/example03$ ansible-playbook -i switch-inventory switch-l2-configure.yaml

PLAY [configure switch] ****
TASK [Gathering Facts] ****
ok: [mastery-switch1]

TASK [bring up ports swp[1-3]] ****
ok: [mastery-switch1]

TASK [query swp1 interface] ****
ok: [mastery-switch1]

TASK [print interface status] ****
ok: [mastery-switch1] => {
    "interface": {
        "changed": false,
        "failed": false,
        "msg": "      Name MAC                  Speed MTU Mode\n--  --\n-----\nDN  swp1  0c:27:59:86:00:01  1G   9216 A
ccess/L2\nAll VLANs on L2 Port\n-----\n1\nUntagged\n-----\n1\nncl-netstat counters\n-----\nRX_OK RX_ERR RX_DRP RX_OVR T
X_OK TX_ERR TX_DRP TX_OVR\n-----\n-----\n-----\n0     0     0     0     0     0     0     0\nRout
ing\n-----\n Interface swp1 is up, line protocol is down\n Link ups: 0
    last: (never)\n Link downs: 16   last: 2021/08/31 13:58:06.59\n PTM st
atus: disabled\n vrf: default\n index 3 metric 0 mtu 9216 speed 1000\n flags:
<UP,BROADCAST,MULTICAST>\n Type: Ethernet\n HWaddr: 0c:27:59:86:00:01\n Inte
rface Type Other\n Master interface: bridge\n protodown: off\n\n"
    }
}

PLAY RECAP ****
mastery-switch1 : ok=4    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
```

Figure 13.7 – Testing the idempotency of our playbook against our Cumulus Linux virtual switch

This time, the state of this task is still `ok`, but we can see that the interface status query results are the same, showing that our configuration has persisted and not been modified in any adverse way (which could happen if idempotency was not supported). In this way, playbooks that automate the configuration of our Cumulus Linux switch are idempotent and result in a consistent state, even when they're run multiple times. This also means that if the configuration of the switch drifts (for example, due to user intervention), it is very easy to see that something has changed. Unfortunately, the `community.network.nclu` module doesn't currently support the `check` mode of `ansible-playbook`, but it still provides a powerful way to configure and manage your switches.

Automating network hardware, such as that running Arista EOS and Cumulus Linux, with Ansible is as simple as that – just think what you could do to automate your network configuration using a combination such as this! I encourage you to explore these free tools to learn more about network automation; I think you'll quickly see the value in it. Hopefully, even with these simple examples, you can see that automating network infrastructure with Ansible is now no more difficult than automating anything else in your infrastructure.

## Best practices

All the usual best practices of using Ansible apply when automating network devices with it. For example, never store passwords in the clear, and make use of `ansible-vault` where appropriate. Despite this, network devices are their own special class of devices when it comes to Ansible, and support for them started to flourish from the 2.5 release of Ansible onward. As such, there are a few special best practices that deserve to be mentioned when it comes to network automation with Ansible.

### Inventory

Make good use of the inventory structure supported by Ansible when it comes to organizing your network infrastructure and pay particular attention to grouping. Doing so will make your playbook development much easier. For example, suppose you have two switches on your network – one is a Cumulus Linux Switch, as we examined previously, and the other is an Arista EOS-based device. Your inventory may look like this:

```
[switches:children]
eos
cumulus

[eos]
mastery-eos ansible_host=10.0.50.99

[cumulus]
mastery-switch1 ansible_host=10.0.50.110
```

We know that we cannot run the `community.network.nclu` module on anything other than a Cumulus switch, so, with some careful use of the `when` statement, we can build tasks in playbooks to ensure that we run the correct command on the correct device. Here is a task that will only run on devices in the `cumulus` group we defined in the preceding inventory:

```
- name: query swp1 interface
  community.network.nclu:
    commands:
      - show interface swp1
  register: interface
  when: inventory_hostname in groups['cumulus']
```

Similarly, good use of grouping enables us to set variables on a device basis. Although you would not put passwords in the clear into your inventory, it might be that your switches of a given type all use the same username (for example, `cumulus` in the case of Cumulus Linux devices). Alternatively, perhaps your EOS devices need specific Ansible host variables set for connectivity to work and to achieve the privilege escalation that's required to perform configuration. Thus, we can extend our preceding inventory example by adding the following code:

```
[cumulus:vars]
ansible_user=cumulus
ansible_password=password

[eos:vars]
ansible_connection=ansible.netcommon.network_cli
ansible_network_os=arista.eos.eos
ansible_user=admin
ansible_password=admin
ansible_become=yes
ansible_become_method=enable
```

Good inventory structure and variable definition will make developing your playbooks a great deal easier, and the resulting code will be more manageable and easier to work with.

## Gathering facts

Ansible includes several specific fact-gathering modules for network devices, and these may be useful for running conditional tasks or simply reporting back data about your devices. If you are using the older `local` connection-based protocol, these device-specific fact modules cannot be run at the start of the playbook run since, at this stage, Ansible does not know what sort of device it is communicating with. Thus, we must tell it to gather the facts for each device as appropriate. Using the `ansible.netcommon.network_cli` protocol described earlier in this chapter resolves this issue, as Ansible knows from the inventory what sort of device it is talking to.

Even so, there are times when it is useful to manually gather facts on your devices – perhaps to validate some configuration work performed as part of a larger playbook. Whether you are doing this, or, for legacy reasons, still reliant on the `local` connection-based protocol, you need to be aware that you will use different fact-gathering modules for different connection types. Let's expand our Arista EOS and Cumulus Linux example to look at this.

There is no specific facts module for Cumulus Linux switches (although, since they are based on Linux, the standard host facts can still be gathered). Using the example of our Arista EOS device, we would run the `arista.eos.eos_facts` module in our playbook based on a unique key in our inventory. In the example inventory that we defined in the previous section, our Arista EOS switches are in the `eos` group, and also have `ansible_network_os` set to `arista.eos.eos`. We can use either of these as a condition in a `when` statement to run the `arista.eos.eos_facts` module on our switches. As such, the beginning of our playbook might look like this:

```
---
- name: "gather all device facts"
  hosts: all
  gather_facts: false

  tasks:
    - name: gather eos facts
      arista.eos.eos_facts:
        when: ansible_network_os is defined and ansible_network_os
        == 'arista.eos.eos'
    - name: gather cumulus facts
      ansible.builtin.setup:
        when: inventory_hostname in groups['cumulus']
```

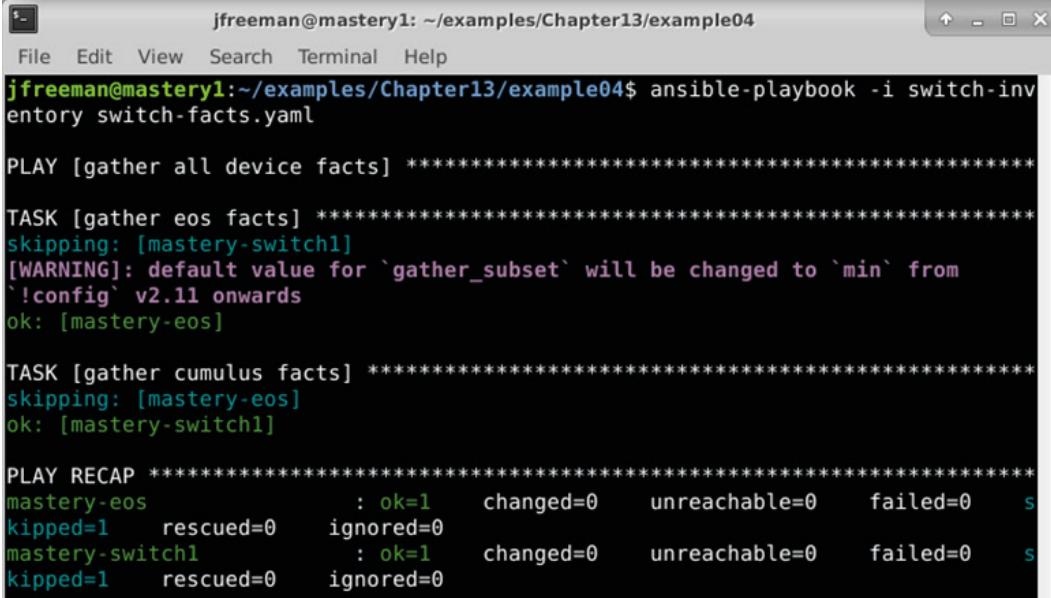
Notice that we set `gather_facts` to `false` at the beginning of this playbook. If you are using the `local` connection-based protocol, you would need to do this as we discussed previously – otherwise, using `ansible.netcommon.network_cli`, you can gather facts at the start of the play if you wish (obviously, though, this is redundant in this example!).

You will also note the more complex conditional we have used on our Arista EOS devices. As Cumulus Linux-based switches use the same SSH-based transport as Linux hosts, they do not need (or indeed have) `ansible_network_os` set, and if we attempt to test this variable in a conditional, we will produce a *variable undefined* error, and thus no subsequent tasks for hosts without the variable defined (our Cumulus Linux switches, in this case) will be attempted. This is hardly the outcome we are looking for! As a result, when combining these hosts with other network devices in the same play, we must always check that the `ansible_network_os` variable is defined before we attempt to perform any queries on it, just as we did in the preceding example.

If you have followed the examples in this chapter and set up your virtual Arista EOS and Cumulus Linux-based devices, you could run this playbook with the following command:

```
ansible-playbook -i switch-inventory switch-facts.yaml
```

The output of a successful playbook run should look like this:



A screenshot of a terminal window titled "jfreeman@mastery1: ~/examples/Chapter13/example04". The window shows the command "ansible-playbook -i switch-inventory switch-facts.yaml" being run. The output displays the execution of the playbook, showing tasks for gathering facts from different device types. It includes messages about skipping tasks for specific hosts and warning about changes in the 'gather\_subset' default value. The final "PLAY RECAP" section summarizes the results for each host, showing counts for ok, changed, unreachable, failed, skipped, rescued, and ignored states.

```
jfreeman@mastery1:~/examples/Chapter13/example04$ ansible-playbook -i switch-inventory switch-facts.yaml

PLAY [gather all device facts] ****
TASK [gather eos facts] ****
skipping: [mastery-switch1]
[WARNING]: default value for `gather_subset` will be changed to `min` from
`!config` v2.11 onwards
ok: [mastery-eos]

TASK [gather cumulus facts] ****
skipping: [mastery-eos]
ok: [mastery-switch1]

PLAY RECAP ****
mastery-eos : ok=1    changed=0    unreachable=0    failed=0    s
skipped=1   rescued=0   ignored=0
mastery-switch1 : ok=1    changed=0    unreachable=0    failed=0    s
skipped=1   rescued=0   ignored=0
```

Figure 13.8 – Gathering facts for multiple device types with a single playbook

Here, you can see how our different facts modules are run on the appropriate device type, thanks to the conditionals we used in our playbook. You can use the techniques outlined in this part of this chapter to build far more complex, multi-device setups by extrapolating the work we have done together. However, no chapter on network automation would be complete without a little more detail on jump hosts. We will look at these next.

## Jump hosts

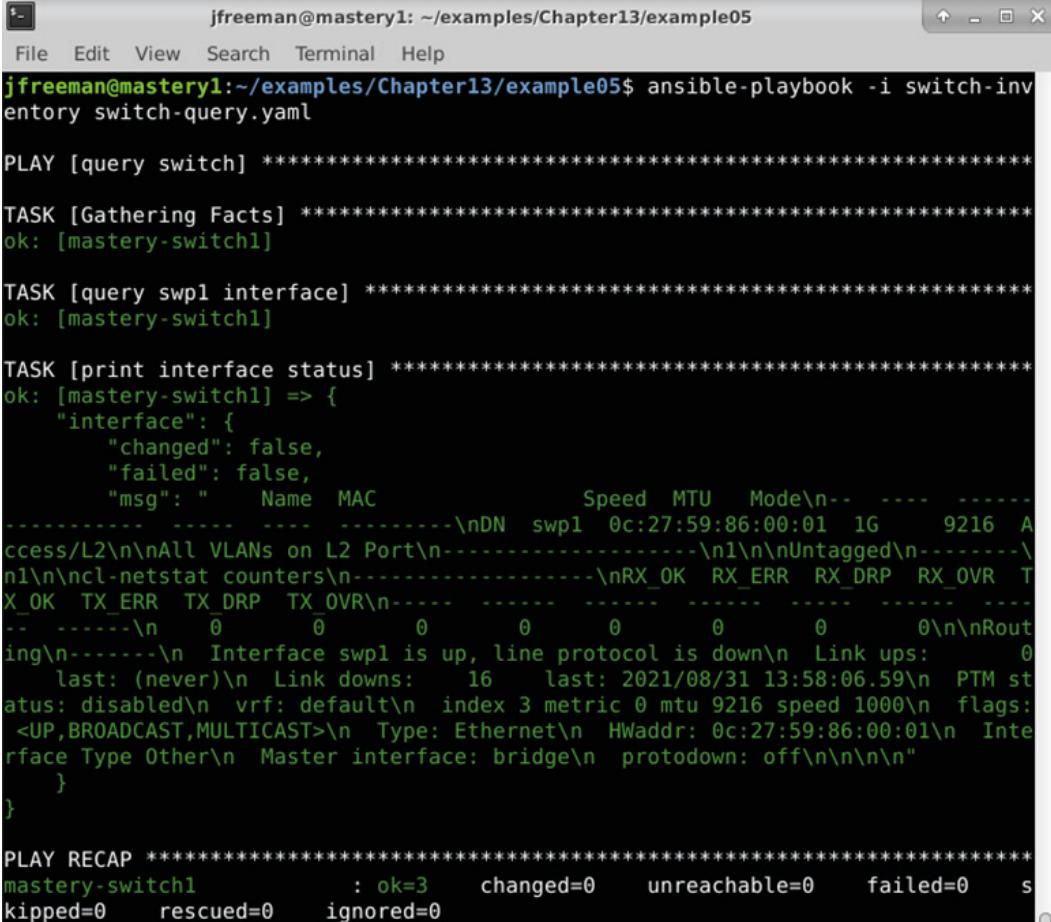
Finally, a word on jump hosts. It is common for network devices to be behind a bastion or jump host of some kind for important security reasons. Ansible provides several mechanisms for doing this, depending on the underlying network transport. For example, SSH connectivity (such as with Cumulus Linux switches) can make use of SSH's ability to proxy commands. There are several ways to achieve these, but the simplest is to add another group variable to the inventory. For example, if we can only access our Cumulus Linux switch via a host called `bastion01.example.com` and are authenticating using an account called `jfreeman`, our inventory variables section would look like this:

```
[cumulus:vars]
ansible_user=cumulus
ansible_ssh_pass=CumulusLinux!
ansible_ssh_common_args=' -o ProxyCommand="ssh -W %h:%p -q
jfreeman@bastion01.example.com"'
```

The preceding proxy command assumes that passwordless authentication has already been configured and is working for the `jfreeman` account on `bastion01.example.com`, and that SSH host keys have already been accepted. Failure to complete these tasks will result in an error.

SSH proxy commands like this would work for other `ansible_connection` modes that are used in network device management too, including `ansible.netcommon.netconf` and `ansible.netcommon.network_cli`, offering support for jump hosts to handle a wide range of network devices. As ever, the best method to be sure about the way to handle a specific type of connectivity is to check the documentation for your specific network device and follow the guidance therein.

If we repeat our earlier example for querying the `swp1` interface of our Cumulus Linux switch, we will see that (with the bastion host correctly set up) the playbook works exactly as it did earlier in this chapter and that no further steps or changes to the code are required:



```
jfreeman@mastery1: ~/examples/Chapter13/example05
File Edit View Search Terminal Help
jfreeman@mastery1:~/examples/Chapter13/example05$ ansible-playbook -i switch-inventory switch-query.yaml

PLAY [query switch] ****
TASK [Gathering Facts] ****
ok: [mastery-switch1]

TASK [query swp1 interface] ****
ok: [mastery-switch1]

TASK [print interface status] ****
ok: [mastery-switch1] => {
  "interface": {
    "changed": false,
    "failed": false,
    "msg": "      Name MAC                  Speed MTU Mode\n--  --
-----\nDN  swp1  0c:27:59:86:00:01  1G   9216 A
ccess/L2\nAll VLANs on L2 Port\n-----\nUn
tagged\n-----\ncl-netstat counters\n-----\nRX_OK RX_ERR RX_DRP RX_OVR TX_O
K TX_ERR TX_DRP TX_OVR\n-----\n 0 0 0 0 0 0 0 0 0\nRout
ing\n-----\n  Interface swp1 is up, line protocol is down\n  Link ups: 0
  last: (never)\n  Link downs: 16  last: 2021/08/31 13:58:06.59\n  PTM st
atus: disabled\n  vrf: default\n  index 3 metric 0 mtu 9216 speed 1000\n  flags:
<UP,BROADCAST,MULTICAST>\n  Type: Ethernet\n  HWaddr: 0c:27:59:86:00:01\n  Inte
rface Type Other\n  Master interface: bridge\n  protodown: off\n\n"
  }
}

PLAY RECAP ****
mastery-switch1 : ok=3    changed=0    unreachable=0    failed=0    s
kipped=0    rescued=0    ignored=0
```

Figure 13.9 – Running an earlier example playbook, but this time through a preconfigured bastion host

This is yet another reason why Ansible has become so popular – there is no need to set up a special proxy application or server for it to access an isolated network. Using the standard SSH protocol, it can connect via any secure host on the network that has SSH configured.

This concludes our exploration of network automation with Ansible. The number and range of devices you can automate the configuration of is limited only by your imagination, and I hope that this chapter has helped you gain a solid foundation in this important field and given you the confidence to explore further.

## Summary

As more and more of our infrastructure gets defined and managed by code, it becomes ever more important that the network layer can be automated effectively by Ansible. A great deal of work has gone into Ansible since the previous release of this book in precisely this area, especially since the release of Ansible 2.5. With these advancements, it is now easy to build playbooks to automate network tasks, from simple device changes to rolling out entire network architectures through Ansible. All of the benefits of Ansible relating to code reuse, portability, and so on are available to those who manage network devices.

In this chapter, you learned about how Ansible enables network management. You learned about effective strategies for handling different device types within your infrastructure and how to write playbooks for them, and then you expanded on this with some specific examples on Arista EOS and Cumulus Linux. Finally, you learned about some of the best practices that must be applied when using Ansible to manage network infrastructure.

This brings us to the conclusion of this book. I would like to thank you for joining me on this journey into the very heart of Ansible, and I hope you have found it beneficial. I believe you should now know about the strategies and tools for managing everything, from small configuration changes to entire infrastructure deployments with Ansible, and wish you luck in this important and ever-evolving area of technology.

## Questions

Answer the following questions to test your knowledge of this chapter:

1. Ansible brings all the benefits of automation from infrastructure management to the world of network device management.
  - a) True
  - b) False
2. When working with a new network device type for the first time, you should always do what?
  - a) Perform a factory reset of the device.
  - b) Consult the Ansible documentation to learn about which collections and modules support it, and what the requirements for those might be.
  - c) Use the `ansible.netcommon.network_cli` connection protocol.
  - d) Use the local connection protocol.

3. Which execution type is described by Ansible as running its automation code on the remote host directly?
  - a) Remote execution
  - b) Local execution
4. Which execution type is described by Ansible as running its automation code on the control node, and then sending the required data over a pre-selected channel (for example, SSH or an HTTP-based API)?
  - a) Remote execution
  - b) Local execution
5. Which connection protocol has (for the most part) superseded the older local connection-based protocol for network devices?
  - a) ansible.netcommon.netconf
  - b) ansible.netcommon.httpapi
  - c) ansible.netcommon.network\_cli
  - d) local
6. Can you gather facts for an Arista-based EOS device at the beginning of a play?
  - a) Yes.
  - b) No.
  - c) Yes, but only when using the ansible.netcommon.network\_cli protocol.
7. All network config on Arista EOS is performed using a single module.
  - a) True
  - b) False
8. Cumulus Linux does not require the ansible.netcommon.network\_cli protocol because of which reason?
  - a) It is not a network operating system.
  - b) It contains a full Linux implementation, including Python.
  - c) It uses the SSH protocol for management.
  - d) It does not have a CLI.

9. Good inventory management is especially important when working in multi-device-type networks.
  - a) True
  - b) False
10. Ansible can support the use of bastion or jump hosts without the need for special configuration or software installation.
  - a) True
  - b) False





Packt.com

Subscribe to our online digital library for full access to over 7,000 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?

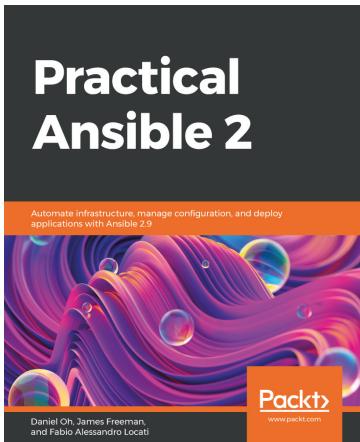
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print,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

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 [packt.com](http://packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub.com](mailto:customercare@packtpub.com) for more details.

At [www.packt.com](http://www.packt.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.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

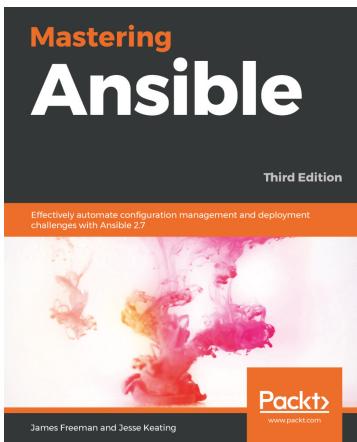


## Practical Ansible 2

Daniel Oh, James Freeman, Fabio Alessandro Locati

ISBN: 9781789807462

- Become familiar with the fundamentals of the Ansible framework
- Set up role-based variables and dependencies
- Avoid common mistakes and pitfalls when writing automation code in Ansible
- Extend Ansible by developing your own modules and plugins
- Contribute to the Ansible project by submitting your own code
- Follow best practices for working with cloud environment inventories
- Troubleshoot issues triggered during Ansible playbook runs



## Mastering Ansible - Third Edition

James Freeman, Jesse Keating

ISBN: 9781789951547

- Gain an in-depth understanding of how Ansible works under the hood
- Fully automate Ansible playbook executions with encrypted data
- Access and manipulate variable data within playbooks
- Use blocks to perform failure recovery or cleanup
- Explore the Playbook debugger and the Ansible Console
- Troubleshoot unexpected behavior effectively
- Work with cloud infrastructure providers and container systems
- Develop custom modules, plugins, and dynamic inventory sources

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](http://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.

## Share Your Thoughts

Now you've finished *Mastering Ansible Fourth Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click [here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

# Index

## A

action plugins  
about 368  
debugging 338

Active Directory Certificate Services (ADCS) 126

Amazon Web Services (AWS) 442

Ansible  
Arista EOS switches, configuring with 487-492  
build, checking 111  
connecting, to Windows with WinRM 119-121  
Cumulus Networks switches, configuring with 492  
Docker containers, building with 465-471  
for network management 477  
installing, from scratch 58-60  
Linux, installing in WSL 112-114  
modules, extending 135  
modules, selecting 132, 133  
running, from Windows 110, 111  
software, installing 134  
used, for automating Windows tasks 131  
WSL, enabling 112

Ansible 2.9  
virtual environment, creating for 59

Ansible 2.9.1 55

Ansible 3.0  
uninstalling 57, 58

Ansible 4.0  
about 4  
Ansible Content Collections 55, 56  
modification 54, 55  
legacy playbooks, porting to 73-77

Ansible 4.0.0 56

Ansible 4.0.1 56

Ansible 4.1.0 56

Ansible 5.0.0 56

ansible-bender 465

Ansible, best practices  
about 499  
facts, gathering 501, 502  
inventory 499, 500  
jump hosts 503, 504

Ansible Collections  
about 60, 61  
anatomy 62-68  
reference link 63

- Ansible, collections list  
  reference link 69
- Ansible configurations 4, 5
- Ansible Container  
  about 465  
  reference link 465
- Ansible Content Collections 55, 56
- Ansible control  
  Windows hosts, setting up with  
    OpenSSH 128-131  
  Windows hosts, setting up  
    with WinRM 115
- ansible-core 56
- Ansible, for network management  
  automated change requests 479  
  backup 478, 479  
  configuration portability 478  
  cross-platform support 477  
  restore 478, 479  
  version control 478, 479
- Ansible Galaxy  
  about 61, 300-305  
  additional modules, installing  
    with 69-73  
  URL 61, 300
- Ansible Galaxy namespaces  
  reference link 62
- Ansible installations  
  upgrading from 57
- Ansible installations, upgrading  
  Ansible 3.0, uninstalling 57, 58
- ansible-playbook  
  executing, with encrypted files 96-98
- Ansible, porting guides  
  reference link 77
- Ansible project  
  contributing 379  
  pull request, making 387
- submissions, contributing 379, 380  
tests, executing 380
- Ansible project, tests  
  code-style tests 385-387  
  integration tests 382-385  
  unit tests 381, 382
- Ansible repository 380
- Ansible system, fatal-error scenario  
  about 401  
  any\_errors\_fatal option 401-404  
  handlers, forcing 407-409  
  max\_fail\_percentage option 404-406
- Ansible version 2.4 300
- Ansible versions 4, 5  
any\_errors\_fatal option 401-404
- application programming interface (API)  
  about 323  
  using 171, 172
- arguments variable 194, 195
- Arista EOS switches  
  configuring, with Ansible 487-492
- auditing 166, 167
- AWX  
  about 140  
  benefits 140  
  installation process 141  
  login screen, accessing 148  
  notifications 170  
  surveys 167  
  workflow templates 169  
  workflow visualizer 169
- AWX deployment  
  about 144  
  self-signed certificate 143
- AWX, integrating with playbook  
  about 149  
  credentials, defining 156-158  
  inventory, defining 152-156

project, defining 151, 152  
stages 150  
template, defining 158-162  
AWX Operator  
  deployment 142  
  installing 142

## B

b64encode filter  
  encoding 211, 212  
basename filter 208  
behavioral inventory variable 10  
blocks feature 238  
Buildah 465

## C

callbacks plugins 363-367  
caller variable 198-200  
catch\_kwargs variable 196, 197  
catch\_varargs variable 197, 198  
certificate authority (CA) 117  
check mode 358  
cli\_command module  
  working with 484-487  
code execution  
  debugging 319  
code-style tests 385-387  
conditional  
  about 180-183  
  inline expressions 183-185  
configuration management  
  database (CMDB) 12  
connection-type plugins 360  
control machine 42

control structure, Jinja2  
  about 180  
  conditional 180-183  
  loop 185-187  
  macro 191  
create, read, update, and  
  delete (CRUD) 425  
Cumulus Networks switches  
  configuring, with Ansible 492  
Cumulus Networks switches, with Ansible  
  inventory, defining 492-494  
  practical examples 494-499  
custom modules 345

## D

data  
  encrypting 82  
data manipulation  
  about 201  
  Ansible, provides custom filters 205  
  built-in filters 203  
  Python object methods 215  
  syntax 201-203  
  undefined arguments, omitting 214  
data manipulation, Ansible  
  additional filters  
  base64 encoding 211, 212  
  content, searching 212, 213  
  dealing, with pathnames 208  
  related, to task status 205, 206  
  shuffle filter 207, 208  
data manipulation, built-in filters  
  default filter 203  
  length filter 204  
  random filter 204  
  round filter 204

data manipulation, Python object methods  
  float methods 217  
  int methods 217  
  list methods 216, 217  
  string methods 215, 216  
default filter 203  
defaults variable 195, 196  
dependency 289  
DevStack  
  reference link 426  
dirname filter 209  
disruptions, with Ansible  
  delaying 410-415  
  minimizing 410  
  tasks, running only once 415-417  
Docker containers  
  building, with Ansible 465-471  
  building, without Dockerfile 456-461  
Docker inventory 461-465  
  images, building 451-456  
  interacting with 451  
Dockerfile  
  avoiding, while building Docker  
  containers 456-461  
Docker inventory 461-465  
Domain Name System (DNS) 334  
Don't Repeat Yourself (DRY) 258  
dot notation 317  
dynamic inventory plugins  
  about 371-376  
  developing 369, 370  
  hosts, listing 370  
  host variables, listing 370  
  script performance, optimizing 377-379  
dynamic vars\_files inclusion 278, 279

## E

encrypted data  
  mixing, with plain YAML 98-102  
encrypted files  
  ansible-playbook, executing with 96-98  
  creating 86  
  decrypting 95, 96  
  editing 86, 92, 93  
  password file 88  
  password prompt 87  
  password rotation on 93-95  
  password script 89  
error condition  
  defining 226-232  
error recovery  
  about 238  
  always section, using 241-244  
  rescue section, using 239-241  
  unreliable environment,  
    handling 245-248  
errors  
  ignoring 224-226  
established file  
  encrypting 90, 91  
executor code  
  debugging 329-333  
expand and contract strategy 398-401  
expanduser filter 210  
external data  
  accessing 42  
extra variables 40  
extra-vars 283, 284

## F

fact-caching plugin 361  
fact modules 40

failure  
 defining 224  
 filter plugins 362, 363  
 filters, dealing with pathnames  
 basename 208  
 dirname 209  
 expanduser 210  
 float methods 217  
 force\_handlers 407-409  
 fully qualified class name (FQCN) 334  
 Fully Qualified Collection Names  
 (FQCNs) 27, 75, 259, 348

**H**

handler 258  
 handlers  
 about 297  
 including 274-277  
 HAProxy behavior 10

**I**

import\_role plugin 300  
 included tasks  
 complex data, passing to 262-266  
 conditionals 266-268  
 tagging 268-271  
 include\_role plugin 300  
 include\_vars 280-282  
 ingress configuration, AWX  
 deployment 147  
 in-place upgrades 394-397  
 integration tests 382-385  
 Internet Protocol (IP) 334, 399  
 int methods 217  
 inventories 152, 153

inventory code  
 debugging 324-328  
 inventory group 154  
 inventory parsing and data sources  
 about 6  
 dynamic inventories 12-17  
 inventory limiting 18-21  
 inventory ordering 8  
 inventory variable data 9-12  
 runtime inventory additions 17  
 static inventories 6-8  
 inventory variables 39  
 iterative tasks  
 with loops 248-253

**J**

Jinja2 35, 73

**K**

Kubernetes pod status  
 after AWX deployment 145, 146

**L**

LDAP 12  
 legacy playbooks  
 porting, to Ansible 4.0 73-76  
 length filter 204  
 list index method 317  
 list methods 216, 217  
 local Ansible code  
 debugging 322, 323  
 local files  
 secrets, logging to 103, 104  
 lookup plugins 42, 361

loops  
about 185-187  
indexing 188-191  
items, filtering 187  
tasks, including with 271-274  
using, in iterative tasks 248-253

LXC container 465

## M

macros  
about 191, 192  
variables 192, 193

macros, variables  
arguments variable 194, 195  
caller variable 198-200  
catch\_kwarg variable 196, 197  
catch\_vararg variable 197, 198  
defaults variable 195, 196  
name variable 193, 194

magic variables 40, 41

max\_fail\_percentage option 404-406

modules  
about 288  
check mode 358  
configuring 482, 483  
construct 345  
developing 344  
documenting 349-356  
fact data, providing 356-358  
installing, with Ansible Galaxy 69-73  
researching 481, 482

modules, check mode  
handling 359, 360  
supporting 358

module transport and execution  
about 33, 37

module arguments 34-36

module blacklisting 36  
module reference 33, 34  
task performance 37

multi-device network  
playbooks, writing to handle 480, 481

## N

name variable 193, 194  
network management  
Ansible 477

NGINX configuration file 10

notifications 170

## O

omit variable 214

on-premise cloud infrastructure  
managing 424, 425

OpenStack inventory sources,  
using 435-441

servers, creating 425, 426

on-premise cloud infrastructure, servers  
runtime inventory, adding 431-434  
virtual servers, booting 426-431

OpenSSH  
Windows hosts, setting up for  
Ansible control 128-131

OpenStack inventory sources  
using 435-441

OpenStack Nova 13

organizations 164

## P

pip  
about 57, 58  
installation link 58

- plain YAML
    - encrypted data, mixing with 98-102
  - playbook
    - about 258
    - debugging 320-322
    - including 284-286
    - logging 310, 311
    - writing 483, 484
    - writing, to handle multi-device network 480, 481
  - playbook code
    - debugging 328, 329
  - playbook parsing
    - about 21
    - execution strategies 28, 29
    - host selection, for plays and tasks 29
    - order of operations 22-24
    - play and task names 30-32
    - play behavior directives 27, 28
    - relative path assumptions 24-26
  - playbooks, writing to handle
    - multi-device network
    - modules, configuring 482, 483
    - modules, researching 481, 482
  - play variables 39
  - plugins
    - about 288
    - action plugins 368
    - callback plugins 363-367
    - connection-type plugins 360
    - developing 360
    - distributing 368
    - fact-caching plugin 361
    - filter plugins 362, 363
    - lookup plugins 361
    - shell plugins 361
    - vars plugins 361
  - Podman 465
  - project 151
  - public cloud infrastructure
    - managing 442-450
  - Python-based modules
    - examples 346-349
  - Python object method
    - about 215
    - versus variable subelements 317-319
- ## R
- random filter 204
  - regular expression (regex) 29, 212
  - remote code
    - debugging 333-338
  - remote files
    - secrets, logging to 103, 104
  - remote hosts
    - secrets, transmitting to 103
  - restart groot 418
  - role
    - application 294-296
    - mixing, with tasks 297-299
  - role-based access control (RBAC) 163
  - role default 39
  - role defaults 288
  - role dependencies
    - about 291
    - conditionals 293
    - tags 292, 293
    - variables 291, 292
  - roles
    - about 287
    - sharing 300
    - structured layout 287

roles, structured layout  
  dependencies 289  
  files and templates 289  
  handlers 288  
  modules 288, 289  
  plugins 288, 289  
  tasks 287  
    variables 288  
role variables 39, 288  
round filter 204  
runtime inventory  
  adding 431-434

## S

scheduling options 164, 165  
secrets  
  logging, to local files 103, 104  
  logging, to remote files 103, 104  
  protecting, while operating 102  
  transmitting, to remote hosts 103  
Secure Shell (SSH) 103, 330  
Secure Sockets Layer (SSL) 103  
self-signed certificate  
  for AWX deployment 143  
service principal 446  
shell plugins 361  
shuffle filter 207, 208  
single tasks  
  serializing 417-419  
source control management (SCM) 151  
standard output (stdout) 310  
standard subscript syntax 317  
string methods 215, 216  
survey 167

## T

task result  
  command execution 237, 238  
  command handling 234-237  
  defining 232-234  
tasks  
  about 258  
  including 259-262  
  including, with loops 271-274  
  mixing, with roles 297-299  
task variables 40

## U

Ubuntu Server 20.04 LTS 4  
Uniform Resource Locator (URL) 304  
unit tests 381, 382

## V

values  
  comparing 217  
  comparison expressions 217, 218  
  logic 218  
  tests 218, 219  
variable 258  
variable introspection  
  performing 311-315  
variable location 38  
variable precedence  
  about 42  
  hashes, merging 48, 49  
  precedence order 43, 44  
  variable group priority ordering 44-47  
variables  
  dynamic vars\_files inclusion 278, 279  
  extra-vars 283, 284

include\_vars 280-282  
including 277  
vars\_files 277, 278  
variable subelements  
  about 315-317  
  versus Python object method 317-319  
variable types 38-40  
vars\_files 277, 278  
vars plugins 361  
Vault  
  about 82  
  IDs 83, 84  
  passwords 83, 84  
Vault feature  
  used, for encrypting any structured  
    data used by Ansible 85  
verbosity  
  about 310, 311  
  levels 310  
virtual environment  
  creating, for Ansible 2.9 59  
virtual servers  
  booting 426-431

## W

Windows  
  Ansible, connecting with  
    WinRM 119-121  
  Ansible, running 110, 111  
Windows authentication  
  administrator account 125  
  certificate validation, over  
    WinRM 126, 127  
  encrypting, with WinRM 121  
  handling, with WinRM 121  
  mechanisms 122-125

Windows hosts  
  setting up, for Ansible control  
    with OpenSSH 128-131  
  setting up, for Ansible control  
    with WinRM 115  
Windows Remote Management (WinRM)  
  about 103  
  listener, enabling 116-119  
  reference link 123  
  system requirement, for automation  
    with Ansible 115, 116  
  used, for connecting Ansible  
    to Windows 119-121  
  used, for encrypting Windows  
    authentication 121  
  used, for handling Windows  
    authentication 121  
  Windows hosts, setting up for  
    Ansible control 115  
Windows Subsystem for Linux (WSL) 111  
Windows tasks  
  automating, with Ansible 131  
workflow templates 169  
workflow visualizer 169

## Y

YAML Ain't Markup Language  
(YAML) 259, 323