

Aditya Patawari, Vikas Aggarwal

Ansible 2 Cloud Automation Cookbook

Write Ansible playbooks for AWS, Google Cloud,
Microsoft Azure, and OpenStack



Packt>

Ansible 2 Cloud Automation Cookbook

Write Ansible playbooks for AWS, Google Cloud, Microsoft Azure, and OpenStack

Aditya Patawari
Vikas Aggarwal



BIRMINGHAM - MUMBAI

Ansible 2 Cloud Automation Cookbook

Copyright © 2018 Packt Publishing

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

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the 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.

Commissioning Editor: Gebin George

Acquisition Editor: Rahul Nair

Content Development Editor: Abhishek Jadhav

Technical Editor: Prachi Sawant

Copy Editor: Safis Editing

Project Coordinator: Judie Jose

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Graphics: Tom Scaria

Production Coordinator: Aparna Bhagat

First published: February 2018

Production reference: 1270218

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78829-582-6

www.packtpub.com



`mapt.io`

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

Why subscribe?

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

PacktPub.com

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

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

Contributors

About the authors

Aditya Patawari is a Systems Engineer and DevOps practitioner. He runs a consulting company where he helps organizations with several systems engineering and DevOps tools like Ansible, Docker, Kubernetes. He is an expert in managing cloud-based infrastructure and helps companies evaluate their infrastructure against various parameters. He has contributed to various open source projects, including Fedora Project and Kubernetes. He is an international speaker on various technical topics including Ansible, Docker, Kubernetes, Infrastructure Management, and Gap Analysis.

Writing this book would have been very difficult without the support of my family, my parents, and my brother. My friends played an important part by dragging me out of my study. I am thankful to Vikas, my coauthor, and the team at Packt. Finally, I would like to thank Puja, my wife. Without her cooperation, I could not have written this book.

Vikas Aggarwal is an Infrastructure Engineer and a SRE, employed with HelpShift Technologies where he is a part of the Operations team, codifying automation for managing numerous deployed application clusters across different cloud services. He also helped his previous firm Browserstack in introducing a lot of automation to deploy and manage a hybrid cloud of thousands of servers. He has matured his skills in cloud and automation in course of his career by investing big-time deploying cloud automation with various tools such as Ansible, Terraform, and more.

First and foremost, I would like to thank my coauthor Aditya Patawari for such a great mentorship, guidance, and experience. He has been a great leader throughout my experience working with him. Special thanks to Packt Publications for giving this opportunity.

About the reviewer

Travis Truman has 20+ years of experience in the technology industry. His previous roles include software engineering, software product architecture, SaaS platform architecture, and VP of Engineering in several Philadelphia-area startups. He is a regular contributor to Open Source software, and has contributed code to OpenStack, Ansible, GopherCloud, Terraform, Packer, Consul, and many other projects powering modern cloud computing. He currently works as a Cloud Architect focused on OpenStack, AWS, and Azure for a Fortune 50 media and technology company based in Philadelphia.

Packt is searching for authors like you

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

Table of Contents

Preface	1
Chapter 1: Getting Started with Ansible and Cloud Management	8
Introduction	8
Infrastructure as Code	10
Introduction of Ansible entities	11
Installing Ansible	12
How to do it...	12
Executing the Ansible command line to check connectivity	12
How to do it...	13
Working with cloud providers	13
Executing playbooks locally	14
How to do it...	14
Managing secrets with Ansible Vault	15
How to do it...	15
Understanding sample application	18
How to do it...	19
Using dynamic inventory	22
How to do it...	23
Chapter 2: Using Ansible to Manage AWS EC2	25
Introduction	26
Preparing Ansible to work with AWS	26
How to do it...	26
Creating and managing a VPC	28
How to do it...	29
Creating and managing security groups	32
How to do it...	32
How it works...	33
Creating EC2 instances	34
Getting ready	34
How to do it...	34

Creating and assigning Elastic IPs	37
How to do it...	37
How it works...	37
Attaching volumes to instances	38
Getting ready	38
How to do it...	38
Creating an Amazon Machine Image	39
How to do it...	39
Creating an Elastic Load Balancer and attaching to EC2 instances	40
How to do it...	40
Creating auto scaling groups	42
How to do it...	42
Deploying the phonebook application	45
How to do it...	46
Chapter 3: Managing Amazon Web Services with Ansible	47
Introduction	47
Creating an RDS instance	48
Getting ready	48
How to do it...	49
How it works...	49
Creating and deleting records in Route53	50
Getting ready	50
How to do it...	50
How it works...	51
Managing S3 objects	52
How to do it...	53
How it works...	54
Managing Lambda	54
Getting ready	54
How to do it...	55
How it works...	56
Managing IAM users	57
How to do it...	57
How it works...	58
Using dynamic inventory	58

How to do it...	59
Deploying the sample application	60
How to do it...	60
How it works...	60
Chapter 4: Exploring Google Cloud Platform with Ansible	61
Introduction	62
Preparing to work with Google Cloud Platform	62
How to do it...	63
Creating GCE instances	65
How to do it...	65
How it works...	67
Attaching persistent disks	67
How to do it...	68
How it works...	68
Creating snapshots for backup	68
How to do it...	68
How it works...	69
Tagging an instance	69
How to do it...	69
Managing network and firewall rules	70
How to do it...	70
How it works...	71
Managing load balancer	71
How to do it...	71
Managing GCE images	72
How to do it...	72
How it works...	73
Creating instance templates	73
How to do it...	74
Creating managed instance groups	74
How to do it...	74
How it works...	75
Managing objects in Google Cloud Storage	76
How to do it...	76
Creating a Cloud SQL instance (without Ansible module)	77

How to do it...	78
Using dynamic inventory	80
How to do it...	80
Deploying the phonebook application	83
How to do it...	83
How it works...	83
Chapter 5: Building Infrastructure with Microsoft Azure and Ansible	84
Introduction	84
Preparing Ansible to work with Azure	85
How to do it...	85
Creating an Azure virtual machine	93
How to do it...	95
How it works...	96
Managing network interfaces	96
How to do it...	97
How it works...	98
Working with public IP addresses	99
How to do it...	99
How it works...	100
Using public IP addresses with network interfaces and virtual machines	100
How do it...	100
How it works...	101
Managing an Azure network security group	102
How to do it...	102
How it works...	103
Working with Azure Blob storage	104
How to do it...	105
How it works...	105
Using a dynamic inventory	106
How to do it...	107
Deploying a sample application	108
How to do it...	108
Chapter 6: Working with DigitalOcean and Ansible	109
Introduction	109

Preparing to work with DigitalOcean	110
How to do it...	110
Adding SSH keys to a DigitalOcean account	111
How to do it...	112
Creating Droplets	112
How to do it...	112
Managing Block Storage	114
How to do it...	114
Attaching a Floating IP	115
How to do it...	115
Using a Load Balancer	116
How to do it...	116
Adding an A DNS record	117
How to do it...	117
Using dynamic inventory	117
How to do it...	118
Deploying a sample application	119
How to do it...	120
Chapter 7: Running Containers with Docker and Ansible	121
Introduction	121
Preparing Ansible to work with Docker	122
How to do it...	123
Running a container	123
How to do it...	124
How it works...	124
Downloading Docker images	125
How to do it...	125
How it works...	126
Mounting volumes in containers	126
How to do it...	127
How it works...	127
Setting up Docker Registry	128
How to do it...	128
How it works...	129

Logging into Docker Registry	129
How to do it...	129
How it works...	130
Using Docker Compose to manage services	130
How to do it...	131
How it works...	132
Scaling up Compose-based service	133
How to do it...	133
How it works...	134
Deploying a sample application	135
How to do it...	135
How it works...	135
Chapter 8: Diving into OpenStack with Ansible	136
Introduction	137
Preparing Ansible to work with OpenStack	138
How to do it...	139
Adding a keypair	140
How to do it...	140
Managing security groups	141
How to do it...	141
How it works...	142
Managing network resources	142
How to do it...	142
How it works...	143
Managing a Nova compute instance	143
How to do it...	143
How it works...	144
Creating a Cinder volume and attaching it to a Nova compute instance	144
How to do it...	145
Managing objects in Swift	145
How to do it...	145
How it works...	146
User management	146
How to do it...	147

How it works...	148
Creating a flavor	149
How to do it...	150
Adding an image	150
How to do it...	150
How it works...	151
Dynamic inventory	151
How to do it...	152
Deploying the phonebook application	153
How to do it...	154
Chapter 9: Ansible Tower	155
Introduction	155
Installing Ansible Tower	156
How to do it...	156
Getting started with Tower	157
How to do it...	158
Adding a machine credential	159
How to do it...	159
Building a simple inventory	160
How to do it...	161
Executing ad-hoc commands	162
How to do it...	162
Using Ansible Tower with a cloud provider	163
How to do it...	163
Integrating Ansible roles with tower	165
How to do it...	165
Scheduling jobs	167
How to do it...	168
Ansible Tower API	169
How to do it...	169
Autoscaling using Callback	170
How to do it...	170
Appendix A: Other Books You May Enjoy	171
Leave a review - let other readers know what you think	173

Preface

Ansible is one of the top configuration management and orchestration tools out there. Released over half a decade ago, Ansible has one of the lowest learning curves among all the configuration management tools out there. It is simple to get started with Ansible since there are no agents required on the remote end, and the transport is SSH, which is preinstalled on almost all Linux servers. One of the major reasons why Ansible is so popular is because it comes with over a thousand modules for various tasks and operations. These modules provide us with the capability to manage infrastructure, servers, deployments, and common software.

One aspect of these modules is cloud management. As cloud is becoming ubiquitous, the need to create infrastructure as code is becoming prominent. We want automated, unattended, and repeatable ways to create an infrastructure with a cloud provider of our choice. Good for us that Ansible has over 300 modules specifically written to interact with the APIs of over 20 cloud providers. These modules let us create, destroy, and manage resources making things automated and repeatable.

In this book, we are going to look at some of the popular cloud providers and use Ansible to create various components of an infrastructure. Each chapter covers a cloud provider and the deployment of an application backed by the database. We will learn, step by step, how to build an infrastructure from scratch and deploy an application. The sample application that we have written has two versions; one uses SQLite, and the other uses MySQL. We will use these variants to demonstrate cloud providers' capabilities and deployment methodology using Ansible. We suggest our readers to go through the [Chapter 1, *Getting Started with Ansible and Cloud Management*](#) to learn more about these applications. For readers who are getting started with cloud automation, we recommend reading the chapters from the beginning. A seasoned user of Ansible and the cloud providers will be able to look at any recipe from the chapter and be able to make out what was discussed in previous recipes. Finally, we suggest that our readers to try executing the code for a deeper understanding of how Ansible works with cloud providers.

Who this book is for

This book is great for readers who have some basic knowledge of how to use Ansible and want to automate their cloud infrastructure setup. This book would also be helpful for readers who are looking to migrate from one cloud provider to another cloud provider. This book has been written to help system engineers, operations engineers, cloud developers, and architects manage their infrastructure as code.

What this book covers

Chapter 1, *Getting Started with Ansible and Cloud Management*, introduces us to some basic Ansible concepts such as secret management using Ansible Vault and the phonebook application, working with cloud providers and how to execute playbooks locally.

Chapter 2, *Using Ansible to Manage AWS EC2*, starts with an introduction to AWS EC2. Here, we build basic infrastructure such as network and security groups, which are eventually used to start a compute instance.

Chapter 3, *Managing Amazon Web Services with Ansible*, takes us deeper into AWS, beyond EC2. Here, we would build and manage AWS components such as RDS. We will also deploy the phonebook application, which will demonstrate the usage of various components of AWS and Ansible modules.

Chapter 4, *Exploring Google Cloud Platform with Ansible*, walks us through the Google Cloud Platform, where we will build network, firewalls, and compute instances. We will look at object storage, instance templates, and any dynamic inventory. Finally, we will deploy our phonebook application to see it in action.

Chapter 5, *Building Infrastructure with Microsoft Azure and Ansible*, helps us learn more about various components of Microsoft Azure. We will start with network components and work our way to create virtual machines. We will use dynamic inventory and deploy our phonebook application.

Chapter 6, *Working with DigitalOcean and Ansible*, introduces us to DigitalOcean. We will create droplets, block storage, and manage load balancers. We will try out dynamic inventory and deploy the phonebook application on a droplet.

Chapter 7, *Running Containers with Docker and Ansible*, helps us learn about Docker containers. We will build images and run containers. We will also look at running our own registry to store images. Finally, we will check out Docker Compose and deploy our phonebook application.

Chapter 8, *Diving into OpenStack with Ansible*, shows how to use Ansible to manage various resources in OpenStack, such as compute instances, blob storage, and network management. We will use dynamic inventory and deploy the phonebook application on a Nova Compute Instance.

Chapter 9, *Ansible Tower*, in this chapter, Tower provides a web-based user interface that executes Ansible's codebase. On top of that, it comes with many features including access control, security, better logging, and workflows.

To get the most out of this book

This book assumes that readers are already familiar with the basics of Ansible and the cloud provider they are going to work on. The book helps the readers to write infrastructure as code and automation. Readers will need a way to authenticate and authorize themselves to the desired cloud providers. Usually, that requires creating an account with said cloud provider. Although care has been taken to use trial and free-tier cloud providers wherever possible, certain recipes might cost users a small amount of money. Please be aware of the financial implications of that.

From a hardware point of view, any modern computer running 64-bit Linux flavor will be able to run the recipes. We have run these recipes from a single core 1 GB RAM compute instance.

Download the example code files

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

You can download the code files by following these steps:

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

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

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

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

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

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here:

https://www.packtpub.com/sites/default/files/downloads/Ansible2CloudAutomationCookbook_ColorImages.pdf.

Conventions used

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

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Once we have the credentials, we should put them in `vars/secrets.yml`."

A block of code is set as follows:

```
- name: Create Custom Network
  gce_net:
    name: my-network
    mode: custom
    subnet_name: "public-subnet"
    subnet_region: us-west1
    ipv4_range: '10.0.0.0/24'
    state: "present"
    service_account_email: "{{ service_account_email }}"
    project_id: "{{ project_id }}"
    credentials_file: "{{ credentials_file }}"
  tags:
    - recipe1
```

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

```
$ pip install boto
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "After that, from the left sidebar, select **IAM & Admin** and then go to the **Service Accounts** section."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Reviews

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

For more information about Packt, please visit packtpub.com.

1

Getting Started with Ansible and Cloud Management

In this chapter, we will cover the following recipes:

- Installing Ansible
- Executing the Ansible command line to check connectivity
- Working with cloud providers
- Executing playbooks locally
- Managing secrets with Ansible Vault
- Understanding sample application
- Using dynamic inventory

Introduction

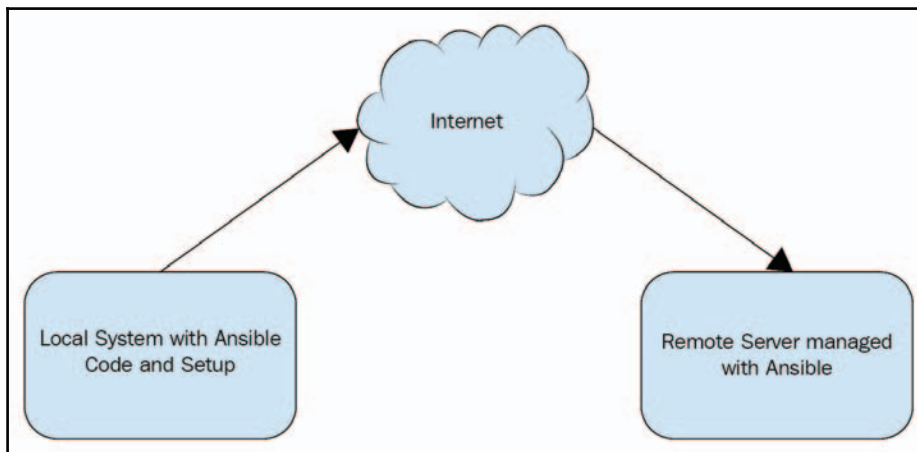
Ansible is a modern automation tool that makes our lives easier by helping us manage our servers, deployments, and infrastructure. We declare what we want and let Ansible do the hard work. Some of the things that Ansible can do are as follows:

- Install and configure software
- Manage users and databases
- Deploy applications
- Remote execution
- Manage Infrastructure as Code

We will focus on the Infrastructure as Code part of Ansible for a significant part of this book.

Ansible has certain distinct advantages over other similar tools.

- Ansible is agentless. So we do not need to install any software on the servers that are to be managed. It does require Python runtime on the servers and a SSH server on remote hosts.
- Ansible supports both push and pull modes. So we can execute Ansible code from a central control machine to make changes on remote machines or the remote machines can pull configuration from a well defined source periodically.
- Code for Ansible is written in YAML (<http://yaml.org/>), which stands for YAML Ain't Markup Language. Ansible did not create and manage a language (or DSL) from scratch. YAML is easy to read, write, and understand. This makes up most of Ansible code's self documentation and reduces the learning curve of Ansible significantly.
- Ansible does not try to re-invent the wheel. Hence it uses SSH as a transport and YAML as a **Domain Specific Language (DSL)**. In typical cases, there are two entities involved, a system (A) where the playbook execution is initiated, and another system (B), usually remote, which is configured using Ansible:



In a nutshell, Ansible helps to manage various components of servers, deployments and infrastructure in a repeatable manner. Its self-documenting nature helps with understanding and auditing true nature of infrastructure.

Infrastructure as Code

Traditionally, infrastructure has been managed manually. At best, there would be a user interface which could assist in creating and configuring compute instances. For most of the users who begin their journey with a cloud, a web-based dashboard is the first and most convenient way to interact. However, such a manual method is error prone. Some of the most commonly faced problems are:

- Requirement for more personnel to manage infrastructure round the clock
- Probability of errors and inconsistencies due to human involvement
- Lack of repeatability and auditability

Creating Infrastructure as Code addresses these concerns and helps in more than one way. A well maintained code base will allow us to refer to the infrastructure state, not only at the present but also at various points in the past.

Ansible helps us code various aspects of infrastructure including provisioning, configuring, and eventually, retiring. Ansible supports coding over 20 cloud providers and self-managed infrastructure setups. Due to its open nature, existing providers can be enhanced and customized and new providers can be added easily.

Once we start managing Infrastructure as Code, we open ourselves to the possibility of a lot of automation. While this book focuses on creating and managing the infrastructure, the possibilities are limitless. We can:

- Raise an alarm if a critical machine becomes unreachable.
- Personnel who do not have access to infrastructure can still help by coding the infrastructure. A code review exercise could help to enforce best practices.
- We can scale infrastructure dynamically based on our requirements.
- In case of a disaster, we can create replacements quickly.
- Passing knowledge of best practices within and outside the organization becomes easier.

Throughout this book, we will create our infrastructure from Ansible code and demonstrate its usability and repeatability.

Introduction of Ansible entities

Before we start diving into the Ansible world, we need to know some basics:

- **Inventory:** We need to have a list of hosts that we want to manage. Inventory is that list. In its simplest form, this can be a text file created manually which just lists the IP addresses of the servers. This is usually enough for small infrastructure or if the infrastructure is static in nature. It follows `ini` syntax and a typical inventory would look like this:

```
[webservers]
server1
[application]
server1
server2
```

If our infrastructure is dynamic, where we add and remove servers frequently, we can use dynamic inventory. This would allow us to generate the inventory in real time. Ansible provides dynamic inventory scripts for many cloud providers and allows us to create dynamic inventory scripts as per our need for non-standard setups. We will use dynamic inventory in this book since it is better suited to cloud based environments.

- **Modules:** Ansible modules are executable plugins that get the real job done. Ansible comes with thousands of modules which can do tasks from installing packages to creating a server. Most of the modules accept parameters based upon which they take suitable actions to move the server towards a desired state. While this book uses modules primarily in the YAML code, it is possible to use modules in command line as an argument to Ansible ad hoc command.
- **Tasks:** A task is a call to an Ansible module along with all the requirements like parameters, variables etc. For example, a task may call upon the template module to read a Jinja template and set of variables, and generate the desired state of a file for the remote server.
- **Roles:** Ansible roles describe the particular role that a server is going to play. It consists of YAML code which defines tasks. It also consists of the dependencies for the execution of tasks like required files, templates and variables.
- **Playbooks:** A playbook is a YAML file where we associate roles and hosts. It is possible to write the tasks in the playbook itself and not use roles but we would strongly discourage this practice. For the sake of readability and better code management, we suggest that playbooks should be kept small with just the name of the hosts or groups to target as defined in inventory and calls to the roles.

- **Variables:** Just like any other programming language, Ansible also has good old-fashioned variables. They hold values which can be used in tasks and templates to perform certain actions. Variables can be created before execution in a file or generated and modified during runtime. A very common use case is to invoke certain tasks if a variable holds a certain value or to store the output of a task in a variable and use it in one of the subsequent tasks.

Installing Ansible

There are many ways to install Ansible. Most of the Linux distributions has Ansible packages in their repositories. Compiling from source is also an option. However, for the sake of uniformity, we are going to use Python pip to install the same version of Ansible for all the readers.

How to do it...

The following command will fetch the Ansible source and install Ansible 2.4.0.0 on our working machine. We have used this version of Ansible throughout the book and we urge our readers to install the same:

```
$ sudo pip install ansible==2.4.0.0
```

Executing the Ansible command line to check connectivity

The simplest form in which we can use Ansible with Ansible ad hoc command line tool. We can execute the tasks using modules without actually writing code in the file. This is great for quick testing or for one off tasks but we should not turn this into a habit since this type of usage is not easily documented and auditable.

How to do it...

We just have to use the Ansible command and pass the ping module as an argument to parameter `-m`. A successful execution will return the string `pong`. It signifies that Ansible can reach the server and execute tasks, subject to the authorization level of the user, of course:

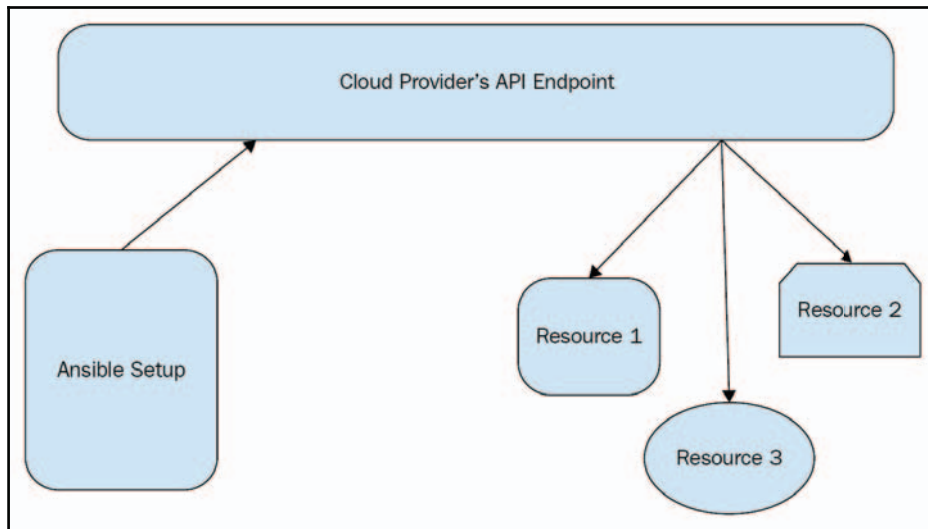
```
$ ansible localhost -m ping
localhost | SUCCESS => {
  "changed": false,
  "failed": false,
  "ping": "pong"
}
```

Working with cloud providers

Under normal circumstances, users execute the `ansible-playbook` command from a system, say A. This system has inventory, playbooks, roles, variable definitions and other information required to configure a remote system, say B, to a desired state.

When we talk about building infrastructure using Ansible, things change a bit. Now, we are not configuring a remote system. We are actually interacting with a cloud provider to create or allocate certain resources to us. We may, at a later point in time, choose to configure these resources using Ansible as well. Interacting with a cloud provider is slightly different from executing a regular playbook. There are two important points that we need to keep in mind:

- A lot of the tasks will execute on the local machine and will interact with API provided by a cloud provider. In principle, we won't need SSH setup because, in typical cases, requests will go from our local machine to the cloud provider using HTTPS.
- The cloud provider will need to authenticate and authorize our requests. Usually this is done by providing a set of secrets, or keys, or tokens. Since these tokens are sensitive, we should learn a little bit about Ansible Vault.



Executing playbooks locally

Most of the time, during the course of this book, our playbooks will be running locally and interacting with a cloud provider. The cloud provider, usually, exposes an API over HTTPS. Generally, we need an inventory file, which has a record of all the hosts, for Ansible to run the playbooks. Let us try to work around it.

The easiest way of running a playbook locally is by using the keyword `localhost` in the playbook as a value for the `hosts` key. This will save us from creating and managing the inventory file altogether.

How to do it...

Consider the following playbook which we can execute without an inventory:

```
---
- hosts: localhost
  tasks:
    - name: get value
      debug:
        msg: "The value is: secret-value"
```

To execute this, we can just run the `ansible-playbook` command with this playbook:

```
$ ansible-playbook playbook.yml
[WARNING]: Host file not found: /etc/ansible/hosts
[WARNING]: provided hosts list is empty, only localhost is available
PLAY [localhost]
*****
TASK [setup]
*****
ok: [localhost]
TASK [include secret]
*****
ok: [localhost]
TASK [get value]
*****
ok: [localhost] => {
  "msg": "The value is: secret-value"
}
PLAY RECAP
*****
localhost                : ok=3    changed=0    unreachable=0    failed=0
```

Managing secrets with Ansible Vault

Secret management is an important aspect of any configuration management tool. Ansible comes with a tool called Ansible Vault which encrypts secrets (technically, it can encrypt any arbitrary file but we will focus on secrets) at rest with 256 bit AES encryption. These secrets can be used in tasks in various ways.

To understand this better, let us create a sample secret and use it in a task.

How to do it...

We will begin with a standard variable file, let us call it `secret.yml`, in Ansible:

```
---
mysecret: secret-value
```

To use this in a playbook, we can include the file as a variable and call it in a task:

```
---
- hosts: localhost
  tasks:
```

```

- name: include secret
  include_vars: secret.yml

- name: get value
  debug:
    msg: "The value is: {{ mysecret }}"

```

Let us run our playbook to verify that everything is good:

```

$ ansible-playbook playbook.yml
PLAY [localhost]
*****
TASK [setup]
*****
ok: [localhost]
TASK [include secret]
*****
ok: [localhost]
TASK [get value]
*****
ok: [localhost] => {
  "msg": "The value is: secret-value"
}
PLAY RECAP
*****
localhost                : ok=3    changed=0    unreachable=0    failed=0

```

Our goal is to protect the content of `secret.yml`. So let us use `ansible-vault` and encrypt it:

```

$ ansible-vault encrypt secret.yml
Vault password:
Encryption successful

```

Now the content of `secret.yml` should look something like this:

```

$ANSIBLE_VAULT;1.1;AES256
646561383562633364326536633239663739613636373830353936313839636433633431623
93764
6634663662333863373937373139326230326366643862390a6434356632373338323663363
23861
316665653339373433333731333538383961663562333164356433633561613665363562303
96534
3038316565336630630a3939386137646165303365656538663461306664663461306335633
46564
33313230336265383532313033653237643662616437636263633039373065346537

```

Executing the playbook like before will fail because our variable file is encrypted. Ansible provides a way to read encrypted files on the fly without decrypting it on the disk. The flag, `--ask-vault-pass`, will request the password from and execute the playbook normally when provided with the correct password:

```
$ ansible-playbook --ask-vault-pass playbook.yml
Vault password:
PLAY [localhost]
*****
TASK [setup]
*****
ok: [localhost]
TASK [include secret]
*****
ok: [localhost]
TASK [get value]
*****
ok: [localhost] => {
  "msg": "The value is: secret-value"
}
PLAY RECAP
*****
localhost                : ok=3    changed=0    unreachable=0    failed=0
```

We will be using Ansible Vault throughout this book to store secrets.

Code Layout

We follow standard code layout to make it easy for everyone to understand the roles. Each chapter has a two playbooks and two roles. One playbook and role has code specific to managing our cloud resources. The other roles has code for deploying the phonebook application. Since there are secrets with each chapter, our final layout would look more or less like this:



```
└── chapter1
    └── roles
        ├── <cloud provider>
        │   ├── files
        │   ├── tasks
        │   └── main.yml
```



```

|   |—— templates
|   |—— vars
|   |—— main.yml
|   |—— secrets.yml
|—— phonebook
    |—— files
    |   |—— phone-book.service
    |—— tasks
    |   |—— main.yml
    |—— templates
    |   |—— config.py
    |—— vars
        |—— secrets.yml

```

Understanding sample application

Throughout the book, we will give examples of how to deploy an application on the infrastructure created by the Ansible playbooks. We have written a simple phone book application using Python's flask framework (<http://flask.pocoo.org>). The phone book application listens on port 8080 and we can use any browser to use the phone book. The app has two variations, one uses SQLite as a database whereas the other one uses MySQL. The application code remains the same, we have just used different databases to demonstrate the application running in a single compute instance and it running across multiple instances or even different components of a cloud provider.

The application code can be obtained from:

- Phone book SQLite: <https://github.com/ansible-cookbook/phonebook-sqlite>
- Phone book MySQL: <https://github.com/ansible-cookbook/phonebook-mysql>

Phone Book		
Contacts (Add contact)		
Name	Email	Phone
Alice	alice@example.com	+1-654-2345
Bob	bob@example.com	+1-876-4532

How to do it...

The application deployment can be done using Ansible. If we are going to deploy the application using SQLite then the following tasks for the phonebook role are good enough:

```
---
- name: install epel repository
  package:
    name: epel-release
    state: present

- name: install dependencies
  package:
    name: "{{ item }}"
    state: present
  with_items:
    - git
    - python-pip
    - gcc
    - python-devel

- name: install python libraries
  pip:
    name: "{{ item }}"
    state: present
  with_items:
    - flask
    - flask-sqlalchemy
    - flask-migrate
```

```
- uwsgi

- name: get the application code
  git:
    repo: git@github.com:ansible-cookbook/phonebook-sqlite.git
    dest: /opt/phone-book

- name: upload systemd unit file
  copy:
    src: phone-book.service
    dest: /etc/systemd/system/phone-book.service

- name: start phonebook
  systemd:
    state: started
    daemon_reload: yes
    name: phone-book
    enabled: yes
```

In the case of MySQL, we need to add some more tasks and information to work with Ansible:

```
---

- name: include secrets
  include_vars: secrets.yml

- name: install epel repository
  package:
    name: epel-release
    state: present

- name: install dependencies
  package:
    name: "{{ item }}"
    state: present
  with_items:
    - git
    - python-pip
    - gcc
    - python-devel
    - mysql-devel

- name: install python libraries
  pip:
    name: "{{ item }}"
    state: present
  with_items:
    - flask
```

```
- flask-sqlalchemy
- flask-migrate
- uwsgi
- MySQL-python

- name: get the application code
  git:
    repo: git@github.com:ansible-cookbook/phonebook-mysql.git
    dest: /opt/phone-book
    force: yes

- name: upload systemd unit file
  copy:
    src: phone-book.service
    dest: /etc/systemd/system/phone-book.service

- name: upload app config file
  template:
    src: config.py
    dest: /opt/phone-book/config.py

- name: create phonebook database
  mysql_db:
    name: phonebook
    state: present
    login_host: "{{ mysql_host }}"
    login_user: root
    login_password: "{{ mysql_root_password }}"

- name: create app user for phonebook database
  mysql_user:
    name: app
    password: "{{ mysql_app_password }}"
    priv: 'phonebook.*:ALL'
    host: "%"
    state: present
    login_host: "{{ mysql_host }}"
    login_user: root
    login_password: "{{ mysql_root_password }}"

- name: start phonebook
  systemd:
    state: started
    daemon_reload: yes
    name: phone-book
    enabled: yes
```

Accordingly, we will create a `secrets.yml` in `vars` directory and encrypt it using `ansible-vault`. The unencrypted data will look like this:

```
---
mysql_app_password: appSecretPassword
mysql_root_password: secretPassword
mysql_host: 35.199.168.191
```

The `phone-book.service` will take care of initializing the database and running the `uwsgi` server for serving the application for both SQLite and MySQL based setups:

```
[Unit]
Description=Simple Phone Book

[Service]
WorkingDirectory=/opt/phone-book
ExecStartPre=/bin/bash /opt/phone-book/init.sh
ExecStart=/usr/bin/uwsgi --http-socket 0.0.0.0:8080 --manage-script-name --
mount /phonebook=app:app
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

Throughout the coming chapters, we will use this role to deploy our phone book application.

Using dynamic inventory

We have talked about dynamic inventory a little bit in this chapter. Throughout this book, in every chapter, we are going to talk about and use dynamic inventory. So let us explore the concept in a bit more depth.

Reiterating what we wrote earlier, dynamic inventory is useful for infrastructures that are dynamic in nature or for cases where we do not want to or cannot maintain a static inventory. Dynamic inventory queries a datasource and builds the inventory in real time. For the sake of this book, we will query cloud providers to get data and build the inventory. Ansible provides dynamic inventory scripts for most of the popular cloud providers.

However, it is simple to create a dynamic inventory script by ourselves. Any executable script that can return a JSON with a list of inventory host groups and hosts in a predetermined format, when passed with a parameter `--list` can be used as an inventory script. A very simple inventory would output something like this:

```
{
    "application": ["10.0.0.11", "10.0.0.12"],
    "database": ["10.0.1.11"]
}
```

More elaborate inventory scripts would output much more information like instance tags, names, operating systems, geographical locations, and, also known as host facts.

How to do it...

To present a realistic example, we have created a simple inventory script for Amazon Web Service in Python. The code is available on GitHub (https://github.com/ansible-cookbook/ec2_tags_inventory):

```
#!/usr/bin/env python
import boto3
import json
import ConfigParser
import os

def get_address(instance):
    if "PublicIpAddress" in instance:
        address = instance["PublicIpAddress"]
    else:
        address = instance["PrivateIpAddress"]
    return address

if os.path.isfile('ec2.ini'):
    config_path = 'ec2.ini'
elif os.path.isfile(os.path.expanduser('~/.ec2.ini')):
    config_path = os.path.expanduser('~/.ec2.ini')

config = ConfigParser.ConfigParser()
config.read(config_path)
id = config.get("credentials", "aws_access_key_id", raw=True)
key = config.get("credentials", "aws_secret_access_key", raw=True)

client = boto3.client('ec2', aws_access_key_id = id, aws_secret_access_key
= key, region_name="us-east-1")
```

```
inventory = {}

reservations = client.describe_instances()['Reservations']
for reservation in reservations:
    for instance in reservation['Instances']:
        address = get_address(instance)
        for tag in instance['Tags']:
            if tag['Key'] == "ansible_role":
                roles = tag['Value'].split(",")
                for role in roles:
                    if role in inventory:
                        inventory[role].append(address)
                    else:
                        inventory[role] = [address]

print json.dumps(inventory)
```

This script reads a file called `ec2.ini` for AWS access and secret key. For the sake of simplicity, we have hardcoded the region to `us-east-1` but this can be changed easily. The script goes through AWS EC2 in the `us-east-1` region and looks for any instance that has a tag with the name `ansible_role` and any valid value like `webserver` or `database`. It will add the IP addresses of those instances to the Python dictionary variable called `inventory`. In the end, this variable is dumped as JSON as output.

We can test this by executing:

```
$ python ec2_tags_inventory.py --list
{"application": ["10.0.0.11", "10.0.0.12"], "database": ["10.0.1.11"]}
```

Note that output may vary depending on the instances that are tagged in EC2. To use this in an Ansible command, we need to make it executable and just pass the script instead of inventory file to `-i` flag like this:

```
$ chmod +x ec2_tags_inventory.py
$ ansible -i ec2_tags_inventory.py database -m ping
database | SUCCESS => {
  "changed": false,
  "failed": false,
  "ping": "pong"
}
```

Needless to say, this is a very simple example and the actual dynamic inventory script provided by Ansible is much more comprehensive and it looks beyond EC2 to other services, such as RDS.

2

Using Ansible to Manage AWS EC2

In this chapter, we will cover the following recipes:

- Preparing Ansible to work with AWS
- Creating and managing a VPC
- Creating and managing security groups
- Creating EC2 instances
- Creating and assigning Elastic IPs
- Attaching volumes to instances
- Creating an Amazon Machine Image
- Creating an Elastic Load Balancer and attaching to EC2 instances
- Creating auto scaling groups
- Deploying the phonebook application

Introduction

Amazon Web Services (AWS) is one of the most popular cloud providers out there. It consists of over 15 regions geographically located across 4 continents, and provides over 70 different services. It serves customers in over 190 countries, including several governments. Elastic Compute Cloud, better known as EC2, launched in 2006, and is one of the most popular services of AWS. Some of the most popular components of EC2 are:

- Instances (virtual machines)
- Volumes (disks)
- Amazon Machine Image or AMI (disk snapshots)
- Security groups (firewalls)
- Elastic Load Balancers (application and network load balancers)
- Auto scaling groups (to scale instances)

Most of the basic building blocks for running a simple application are provided by AWS EC2 itself.

Preparing Ansible to work with AWS

Ansible interacts with AWS APIs to manage various infrastructure components. Using APIs requires credentials and an IAM user with the right permissions would help us do this. We also need to set up some libraries that would be required to execute certain Ansible modules. So let us get started.

How to do it...

Ansible ships with scores of AWS modules. These Ansible modules use AWS Python SDK, called Boto, as dependency and interact with AWS.

1. Let us install Boto using Python pip to get started:

```
$ pip install boto
```


2. Along with Boto, we also need to have a user who has enough privileges to create and delete AWS resources. AWS has a predefined policy called `AmazonEC2FullAccess` which can be attached to a user. However, we prefer using a more permissive policy since we would be working on other AWS components in the next chapter.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "NotAction": [
        "iam:*",
        "organizations:*"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "organizations:DescribeOrganization",
      "Resource": "*"
    }
  ]
}
```

3. Once we have the policy defined, we need the user's access key ID and secret access key. These can be generated using AWS IAM dashboard. We will protect these keys using Ansible vault. Let us create a secret file with the keys:

```
---
access_key: AKIAIFA7A4UKUHQ3LLL
secret_key: plmkoi+jhy654gbjuyd345789o/-098u
```

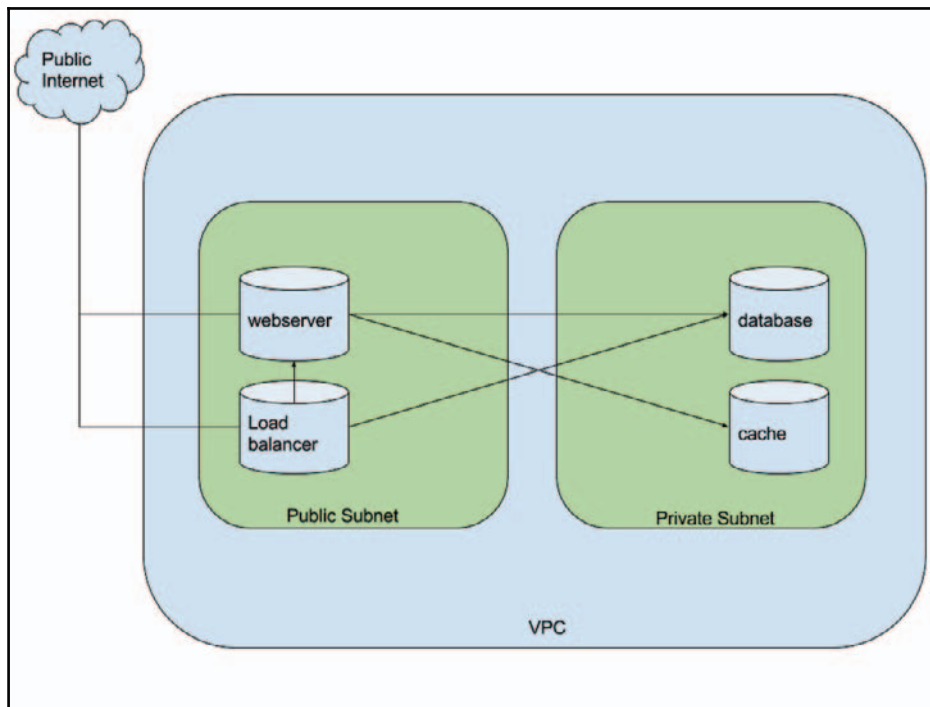
4. Now, we encrypt them:

```
$ ansible-vault encrypt chapter2/roles/ec2/vars/secret.yml
```

Once we have the `Boto` library and credentials for a privileged user, we are good to try out some recipes from this chapter.

Creating and managing a VPC

Virtual Private Cloud, or VPC, is technically not a part of EC2. However, this is usually the first step when getting started with EC2. VPC creates a virtual network which logically isolates our resources. This improves security and management since, logically, subnet and gateway are dedicated for our resources only. A common usage of VPC is to isolate public-facing services (like load balancers or instances running public services) and servers storing data (like databases) which do not require direct access from the wider internet.



Technically, a VPC has several moving parts, as depicted in the preceding image. Even a simple architecture would consist of the following components:

- The VPC itself, where we will allocate a high-level **Classless InterDomain Routing (CIDR)** block and choose a region.
- A public subnet, which will use a chunk of CIDR from the larger CIDR that we defined above.
- A private subnet, which will use a chunk of CIDR from the larger CIDR that we defined above.

- An Internet Gateway, which will be attached to the public subnet. This gateway will route the traffic to the public internet.
- A NAT Gateway, which will be attached to the private subnet. This gateway will provide **Network Address Translation (NAT)** services to outbound traffic for the private subnet.
- A route table attaching the Internet Gateway with the public subnet.
- A route table attaching the NAT Gateway with the private subnet.

Now that we have broken down the list of components that we need to build a VPC, let us start writing Ansible code for the same.

How to do it...

We can create a VPC by using an `ec2_vpc_net` module. This module will take a name, the regions, and a CIDR block as the argument along with our credentials.

1. Let us define the task:

```
- name: Create AWS VPC
  ec2_vpc_net:
    name: "{{ vpc_name }}"
    cidr_block: "{{ vpc_cidr_block }}"
    region: "{{ aws_region }}"
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    state: present
  register: my_first_vpc
```

Note that we have registered the output of the task in a variable called `my_first_vpc`. We will use values from this variable in the subsequent tasks. We have used quite a few variables as well. Using variables appropriately makes it easier to reuse the roles and playbooks at a later point. Other than `access_key` and `secret_key`, the rest of the variables are defined in `chapter2/roles/ec2/vars/main.yml`:

```
# VPC Information
vpc_name: "My VPC"
vpc_cidr_block: "10.0.0.0/16"
aws_region: "us-east-1"
```

2. Now let us create a public and a private subnet using an `ec2_vpc_subnet` module. We will supply a smaller block of CIDR out of the CIDR block that we used while creating the VPC. We also need to provide information about the region and the availability zone within the region. We will get the VPC ID from the variable that we registered in the previous task:

```
- name: Create Public Subnet in VPC
  ec2_vpc_subnet:
    vpc_id: "{{ my_first_vpc.vpc.id }}"
    cidr: "{{ vpc_public_subnet_cidr }}"
    region: "{{ aws_region }}"
    az: "{{ aws_zone }}"
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    state: present
    tags:
      Name: Public Subnet
  register: my_public_subnet

- name: Create Private Subnet in VPC
  ec2_vpc_subnet:
    vpc_id: "{{ my_first_vpc.vpc.id }}"
    cidr: "{{ vpc_private_subnet_cidr }}"
    region: "{{ aws_region }}"
    az: "{{ aws_zone }}"
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    state: present
    tags:
      Name: Private Subnet
  register: my_private_subnet
```

We have created two subnets using these two tasks. The tasks are identical, except for the CIDR block allocated to them. At this point, there is not much of a difference between the public and private subnet in terms of functionality. The functional difference will arise when we attach route tables later. We will register the output of these tasks in a variable for further use. For these tasks, we need to add the following variables to our `roles/ec2/vars/main.yml`:

```
aws_zone: "us-east-1a"
# Subnets
vpc_public_subnet_cidr: "10.0.0.0/24"

# Subnets
vpc_private_subnet_cidr: "10.0.1.0/24"
```

3. Let us create the Internet Gateway now. This is quite simple. All we need to do is provide the VPC ID and region along with the credentials. We will register the output of this task in a variable:

```
- name: Create Internet Gateway
  ec2_vpc_igw:
    vpc_id: "{{ my_first_vpc.vpc.id }}"
    region: "{{ aws_region }}"
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    state: present
  register: my_first_igw
```

4. After this, we will create the NAT Gateway. One thing to note here is that the NAT Gateway is attached to the private subnet but it is created in the public subnet. This is because inbound traffic needs to reach this instance, which will then be translated and forward onto instances in the private subnet. We will get the public subnet ID from the variable that we registered:

```
- name: Create NAT Gateway
  ec2_vpc_nat_gateway:
    if_exist_do_not_create: yes
    subnet_id: "{{ my_public_subnet.subnet.id }}"
    region: "{{ aws_region }}"
    state: present
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    wait: yes
  register: my_first_nat_gateway
```

5. With both the Internet Gateway and NAT Gateway created, we will create and attach the routing table using an `ec2_vpc_route_table` module. We will get the VPC ID, subnet ID, and gateway ID from the variables that we have registered before:

```
- name: Create Route Table for Public Subnet
  ec2_vpc_route_table:
    vpc_id: "{{ my_first_vpc.vpc.id }}"
    region: "{{ aws_region }}"
    routes:
      - dest: 0.0.0.0/0
        gateway_id: "{{ my_first_igw.gateway_id }}"
    subnets:
      - "{{ my_public_subnet.subnet.id }}"
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
```

```

tags:
  Name: Public Subnet Route Table

- name: Create Route Table for Private Subnet
  ec2_vpc_route_table:
    vpc_id: "{{ my_first_vpc.vpc.id }}"
    region: "{{ aws_region }}"
    routes:
      - dest: 0.0.0.0/0
        gateway_id: "{{ my_first_nat_gateway.nat_gateway_id }}"
    subnets:
      - "{{ my_private_subnet.subnet.id }}"
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
  tags:
    Name: Private Subnet Route Table

```

With these tasks, our VPC is configured and ready to use. We can create resources in this VPC and use them to deploy our applications.

Creating and managing security groups

EC2 security groups are virtual firewalls, which control inbound and outbound traffic to and from our EC2 Instance. We will create security groups before an EC2 Instance because this resource is required for creating an EC2 instance. Security groups and EC2 instances have many-to-many relationships. We can have a single instance with multiple security groups and a single security group can be applied to multiple instances, even multiple AWS instances present in the same subnet can have different security groups.

How to do it...

We can create a security group, using an `ec2_group` module, this will take the VPC ID, the region, and rules as input.

Let's create a task for a security group:

```

---
- name: Create EC2 Security Group
  ec2_group:
    name: my_first_sg
    description: A sample security group webservers
    vpc_id: "{{ my_first_vpc.vpc.id }}"
    region: "{{ aws_region }}"

```

```
aws_secret_key: "{{ secret_key }}"
aws_access_key: "{{ access_key }}"
rules:
  - proto: tcp
    from_port: 80
    to_port: 80
    cidr_ip: 0.0.0.0/0
  - proto: tcp
    from_port: 22
    to_port: 22
    cidr_ip: "{{ vpc_cidr_block }}"
  - proto: tcp
    from_port: 443
    to_port: 443
    cidr_ip: 0.0.0.0/0
  - proto: icmp
    from_port: 8
    to_port: -1
    cidr_ip: "{{ vpc_cidr_block }}"
rules_egress:
  - proto: all
    cidr_ip: 0.0.0.0/0
register: my_first_sg
```

How it works...

Here we have used variables registered from previous tasks to pass the VPC ID as an input (`vpc_id`). This task creates a security group with the name `my_first_sg`. The parameter `rules` is used for defining all ingress policies and similarly `rules_egress` for outbound policies. Every block inside ingress rules requires four key bits of information; that is, the protocol (TCP, UDP, or ICMP), the start of the port range (`from_port`), the end of the port range (`to_port`), and the CIDR to the whitelist. In all other protocols (except ICMP) we define the port range. But if we choose protocol as ICMP we have to define ICMP code. For example, 8 is used for ICMP echo requests and -1 is a wildcard (that is, any ICMP type number). We have allowed port 80 and port 443 from anywhere; that is 0.0.0.0/0. But port 22 (which is a default port for SSH connections) and all ICMP requests are accessible within the CIDR address space of our VPC.



We have registered the security group as `my_first_sg`. We will be using this as a variable in upcoming tasks.

Creating EC2 instances

Elastic Cloud Compute or EC2, forms a central part of Amazon Web Services. It is one of the most popular services by AWS, which provides rented virtual computers (called instances) under various capacities in terms of CPU, memory, disk, network, and so on where users can run and host their applications.

Getting ready

Before going ahead and launching an EC2 Instance, we will require resources we created in preceding tasks; that is, security groups and VPC (subnets).

Creating an EC2 instance requires the following basic parameters:

- **Instance type**, which determines hardware of the host computer used for our instance. An instance type offers a wide range of compute, memory, network, and storage capacity.
- **Amazon Machine Image (AMI)**, which contains information to launch the instance. It consists of a template for root volume; that is, an operating system and applications. We will learn more about AMI in upcoming tasks.
- **EC2 keypair** which is a set of RSA public-private keys. It is used to log in as the default user of the instance.

How to do it...

We can create an EC2 instance using an Ansible module, which will take an instance type, AMI, subnet ID of VPC, tags, the exact count of instances, and region as input. Let's define tasks:

1. To log into the created instance, we need to create an EC2 keypair. There are two ways to create a keypair. One is that we create a keypair on the AWS console and download a private key; the other is to import an existing keypair into AWS. We will import a keypair into EC2:

```
- name: Create EC2 Key Pair
  ec2_key:
    name: my_first_key
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    key_material: "{{ key }}"
```



```
region: "{{ aws_region }}"
state: present
```

Every AWS EC2 region maintains its own set of keypairs for instances to be launched in respective regions. We have passed `aws_region`, where we will be booting our instance and another variable that is `key`, which contains the content of our public key. This variable will be defined under `ec2/vars/main.yml`. After this task, our EC2 keypair will be generated and we can use the name defined for this keypair as the input in our next task to create an EC2 instance.

2. We need to add the following things to our variables:

```
#Key Pair Content
key: 'ssh-rsa AAAAB3NzaC1yc2EAAAAD.....'
```

3. After we have our keypair in place, we are ready to create our EC2 instances in the subnets we created in VPC tasks. We have passed the following inputs to define our instance:

- Instance configuration using the variable `instance_type`
- Private subnet of the VPC we created using the registered variable; that is `my_private.subnet.id` in VPC task
- Instance tags to refer our instance as key, value pair that is `Name: Private Instance`
- Count of instances we want to keep in AWS EC2 region with specific tags using variable `private_instance_count`

The argument `count_tag` with `exact_count` determine how many nodes of an instance with the specific tag should be running. If the count of instances exceeds the parameters `exact_count` then extra instances would be terminated. In this example, we are keeping exact count for the instance tagged with `Name: Private Instance`.

Private Instance.

```
- name: Create EC2 Instance in private subnet
  ec2:
    key_name: my_first_key
    instance_type: "{{ instance_type }}"
    image: "{{ ami_id }}"
    wait: yes
    group: my_first_sg
    vpc_subnet_id: "{{ my_private_subnet.subnet.id }}"
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
```

```

    region: "{{ aws_region }}"
    count_tag:
      Name: Private Instance
    exact_count: "{{ private_instance_count }}"
    instance_tags:
      Name: Private Instance

```

4. We need to define the following variables for the preceding tasks:

```

#EC2 Instance Info

instance_type: 't2.micro'
ami_id: 'ami-46c1b650'
private_instance_count: 1
public_instance_count: 1

```

5. Similarly, we can create an instance in the public subnet as well:

```

- name: Create EC2 Instance in public subnet
  ec2:
    key_name: my_first_key
    instance_type: "{{ instance_type }}"
    image: "{{ ami_id }}"
    wait: yes
    group: my_first_sg
    vpc_subnet_id: "{{ my_public_subnet.subnet.id }}"
    assign_public_ip: yes
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    region: "{{ aws_region }}"
    count_tag:
      Name: Public Instance
    exact_count: "{{ public_instance_count }}"
    instance_tags:
      Name: Public Instance
  register: ec2_public_instance

```

This task is identical to the previous task. The only difference is the different tags and subnet ID. We have created two instances in the preceding tasks, in each of the subnets we created in VPC tasks.

Creating and assigning Elastic IPs

An Elastic IP address, also known as EIP is a static IPv4 address, which can be assigned to an EC2 instance. The Elastic IP address is generally used for covering the failure of an EC2 Instance by quickly remapping it to another EC2 instance. An Elastic IP address is allocated to an AWS account and can only be used in a specific region. That is, we cannot associate an Elastic IP address allocated in one region to an EC2 instance in a different region.

How to do it...

We can allocate and associate an Elastic IP address with an EC2 instance using an `ec2_ip` module. This will require the instance ID of the EC2 instance we want to associate this Elastic IP address with and the region of that instance as input parameters.

```
- name: Allocate Elastic IP and associate it with an instance
  ec2_eip:
    device_id: "{{ item }}"
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    region: "{{ aws_region }}"
  with_items:
    - "{{ ec2_public_instance.instance_ids }}"
  register: elastic_ip
```

How it works...

In this task, we are allocating an Elastic IP address and associating it with the EC2 instance we have created in the previous task.



We should note that we have used the `ec2_public_instance` variable that we registered in the previous task. If we have created more than one EC2 instance then this task will allocate the same count of Elastic IP addresses and will associate them as one-to-one mapping with those EC2 instances.

Attaching volumes to instances

Amazon Elastic Block Storage, also known as EBS, is block-level storage that can be attached to an EC2 Instance. Once we attach an EBS volume, we can create a file system on top of these volumes, mount it, and use it like an attached disk, technically as block storage. Amazon EBS volumes are placed in a specific availability zone and they can only be attached to an EC2 instance present in the same availability zone. EBS volumes are automatically replicated to protect our data from the failure of a single component. In this section, we will be writing a task to create an EBS volume and attaching it to an existing EC2 instance (which was created in previous tasks).

Getting ready

Before we write code for EBS volumes, we need to know about device name:

Device name: When we attach an EBS volume to our instance, we assign a device name (such as `/dev/xvdf` or `/dev/xvdfj`, and so on) for our instance which is used by Amazon EC2. While EC2 requires this parameter, the block driver of the instance assigns the actual name for mounting that volume. The name assigned by the block driver can be different than the one that we specify while attaching the volume.

How to do it...

To create an EBS volume, we can use an `ec2_vol` module. This will take the device name, the instance ID, the volume size, and the tag name as input parameters:

```
- name: Create EBS volume and attach to Instance
  ec2_vol:
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    region: "{{ aws_region }}"
    instance: "{{ item }}"
    volume_size: 10
    name: Public Instance 1
    device_name: /dev/xvdf
    with_items: "{{ ec2_public_instance.instance_ids }}"
    register: ec2_vol
```

This task will create an EBS volume with the tag name as Public Instance 1 and size 10 GB. Once the EBS volume is ready, it will be attached to the EC2 instance we provided with the device name `/dev/xvdf`.

We should note here that in this task, `name` and `device_name` are two different parameters with their own significance. A name is used as a tag; whereas `device_name` is unique for a one-to-one mapping between an EC2 instance and EBS volume used by AWS EC2.

Also, we have used the same registered variable for the EC2 instance ID that we used in Elastic IPs. If the variable had details of more than one instance, then this task would have created the same number of volumes and would have mapped it one-to-one.

Creating an Amazon Machine Image

We have already discussed **Amazon Machine Image (AMI)** while creating EC2 instances. AMI is the most common way to create backup images of an EC2 Instance. For the sake of consistency, while creating an AMI, AWS reboots the instance by default. However, this can be overridden and AMIs can be created without rebooting the EC2 Instance. AMIs store references to device mapping with corresponding volume snapshots. AMIs can be used to define a standard template which can be used while launching EC2 instances. For example, we may want to use a standard flavor of Linux with predefined system configurations across all EC2 instances. We can create an AMI for this and use it while creating EC2 instances.

How to do it...

1. We can create an AMI of an existing EC2 Instance using an `ec2_ami` module. In this task we will be creating an AMI from the existing instance which, we created in one of the previous tasks. This task will require the instance id, the option to reboot the instance while creating the image, the name, and the tags for the image to be created as input parameters:

```
- name: Create AMI of Public Instance Created
  ec2_ami:
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    region: "{{ aws_region }}"
    instance_id: "{{ item }}"
    wait: yes
    name: first_ami
```

```
no_reboot: yes
tags:
  Name: First AMI
with_items:
  - "{{ ec2_public_instance.instance_ids }}"
register: image
```

2. While creating an AMI for running an EC2 instance, we can select not to reboot that instance. If we are sure that the state of that instance will be consistent while creating AMI, we can use `no_reboot` as yes; otherwise the image created for that instance will not be in the same state we wanted it to be.



We have used here the same EC2 registered variable, used in various previous tasks to pass instance ID's.

Creating an Elastic Load Balancer and attaching to EC2 instances

Often, for the purpose of load balancing, as well as to achieve high availability, we may choose to serve the traffic using a load balancer. AWS provides a virtual load balancer, called **Elastic Load Balancer (ELB)**, which can be used to receive traffic from clients and route the traffic to a set of instances which are attached to the ELB. Apart from stability, the ELB has quite a few features which can help in improving the overall uptime. One of the most important of these features is health check. This lets ELB determine that an attached instance has gone bad and it should stop routing traffic to it. ELB can also insert cookies which can be used for making routing decisions and it can be used to offload SSL from an application.

How to do it...

Let's start by creating an ELB. We will need to add the ELB to a security group. We also need to provide the region and subnet in which we want to create the ELB. We will make this ELB serve HTTP requests on port 80 and route the requests to port 80 of the instance:

```
- name: Create ELB in public subnet
ec2_elb_lb:
  state: present
```

```

name: "{{ elb_name }}"
security_group_ids: "{{ my_first_sg.group_id }}"
region: "{{ aws_region }}"
subnets: "{{ my_public_subnet.subnet.id }}"
aws_access_key: "{{ access_key }}"
aws_secret_key: "{{ secret_key }}"
purge_subnets: yes
listeners:
- protocol: http
  load_balancer_port: 80
  instance_port: 80
register: my_first_elb
tags:
- elb
- recipe8

```

Once our ELB is set up, we need to identify the instances that we are going to attach to the ELB. One way to do so is by filtering the instance with a tag:

```

- name: Get all ec2 instances with given tags
  ec2_instance_facts:
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    aws_region: "{{ aws_region }}"
    filters:
      "tag:Name": Public Instance
  register: ec2_instances_public
  tags:
    - elb
    - recipe8

```

Now that we have the instance(s) that we want to attach with the ELB, let's go ahead and attach them for real. The variable `ec2_instances_public` holds the details of the instances which we got by using the filter:

```

- name: Register all public instances with elb created
  ec2_elb:
    instance_id: "{{ item.id }}"
    ec2_elbs: "{{ my_first_elb.elb.name }}"
    state: present
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    aws_region: "{{ aws_region }}"
  with_items:
    - "{{ ec2_instances_public.instances }}"
  ignore_errors: yes
  tags:

```

- elb
- recipe8

Creating auto scaling groups

So far, we have seen various services provided by AWS EC2. We have also seen the dynamic nature of the cloud that lets us spin any number of instances, volumes, load balancers, and so on.

When we deploy an application in production, we are likely to see non-uniform traffic patterns.

We might see a pattern where peak time starts at mid-afternoon and ends at midnight. For such cases, we might need to add more resources at certain times to keep our application latency uniform. Using auto scaling groups, we can achieve this goal. AWS EC2 provides three major components for auto scaling EC2 instances:

- **Launch Configurations** act a template for auto scaling groups to launch EC2 instances. It contains information like the AMI ID, the security group, and so on required to launch an EC2 instance.
- **Auto Scaling Groups** is a collection of EC2 instances which share similar characteristics. This is defined in the launch configuration. Auto scaling groups are configured with parameters such as a maximum number of instances, a minimum number of instances, scaling policies, and load balancers to manage EC2 instances.
- **Scaling Group Policies** are policies attached to an auto scaling group, which define actions like scale up or scale down. These policies are hooked with CloudWatch metrics alarm, which notifies the policy to increase or decrease EC2 instances in auto scaling groups.

How to do it...

We will be using the `ec2_lc` module for creating launch configurations, the `ec2_asg` module for creating auto scaling groups, the `ec2_scaling_policy` module to define scaling policies and the `ec2_metric_alarm` module for creating a metric alarm to take actions like scaling up EC2 instances.

1. In this task we will be creating an auto scaling group with a scale-up policy piped with a CloudWatch alarm configured with CPU utilization metrics; and if it goes above a defined threshold, it will scale up EC2 instances in auto scaling groups:

```
- name: Create Launch Configuration
  ec2_lc:
    region: "{{ aws_region }}"
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    name: my_first_lc
    image_id: "{{ ami_id }}"
    key_name: my_first_key
    instance_type: "{{ instance_type }}"
    security_groups: "{{ my_first_sg.group_id }}"
    instance_monitoring: yes
```

In the preceding task, we created a launch configuration which would be used to create new instances in case of a scale-up event. We used the same parameters that we used to create an EC2 instance; that is an AMI ID, a key pair name, a security group and instance ID, and a region. We used one additional parameter (`instance_monitoring`) which will enable CloudWatch metric monitoring such as CPU utilization metric.

2. Let us create the auto scaling group next:

```
- name: Create Auto Scaling group
  ec2_asg:
    name: my_first_asg
    region: "{{ aws_region }}"
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    load_balancers:
      - first-elb
    launch_config_name: my_first_lc
    min_size: 1
    max_size: 5
    desired_capacity: 3
    vpc_zone_identifier:
      - "{{ my_private_subnet.subnet.id }}"
    tags:
      - environment: test
```

In this task, we are creating an auto scaling group for our EC2 instances. This task will need the launch configuration name, the min/max count of instances we want to keep in our auto scaling group, the desired capacity we want to keep in our auto scaling group, the VPC subnet where we want to add more EC2 instances, and the tags as input parameters. We should note here that we are using the subnet ID registered while creating the VPC subnet.

3. Next, we will create a scaling policy:

```
- name: Configure Scaling Policies (scale-up)
  ec2_scaling_policy:
    region: "{{ aws_region }}"
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    name: scale-up policy
    asg_name: my_first_asg
    state: present
    adjustment_type: ChangeInCapacity
    min_adjustment_step: 1
    scaling_adjustment: +1
    register: scale_up_policy
```

Here we have defined an auto scaling policy for scaling up EC2 instances. This task requires the name of the auto scaling group, the amount by which the auto scaling group is adjusted (`scaling_adjustment`), the minimum amount type by which the auto scaling group is adjusted by the policy (`min_adjustment`), and the type of change in capacity (`adjustment_type`), which can be set as `ChangeInCapacity`, `ExactCapacity`, or `PercentageChangeInCapacity`.

4. Now, we will configure CloudWatch alarm:

```
- name: Configure CloudWatch Metric for Scaling Up Policy
  ec2_metric_alarm:
    state: present
    region: "{{ aws_region }}"
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    name: "scale-up-metric"
    description: "This alarm notify auto scaling policy to step up
instance"
    metric: "CPUUtilization"
    namespace: "AWS/EC2"
    statistic: "Average"
    comparison: ">="
    threshold: 60.0
```

```
unit: "Percent"
period: 300
evaluation_periods: 3
dimensions:
  AutoScalingGroupName: "my_first_asg"
alarm_actions:
  - "{{ scale_up_policy.arn }}"
```

In the preceding task, we are creating a `CloudWatch` metric alert which will trigger the scaling policy we defined in the preceding task. This task will need the following parameters:

- Metric to monitor, which we have selected as CPU utilization.
- Namespace of that metric, which, in our case, is `AWS/EC2` as we are working with our EC2 instances.
- Aggregation (statistics) we want to perform on the metric. We will be doing the average over data points.
- Unit of metric. We are using percentage here.
- Will look at data points for a period 300 seconds.
- Threshold to trigger the alert, which is 60.0 % of CPU utilization.
- Evaluation period of 3, which means, over a period of 300 seconds, if the average CPU utilization is greater than 60% for 3 consecutive runs, it will trigger an alarm.
- Action required if the alarm is triggered. We are notifying our auto scaling policy defined in the previous task as an `alarm_actions`.

Deploying the phonebook application

Our phonebook application can be deployed to the instances that we have created. When deploying an application to an instance, we either need to know the IP address of the instance and prepare the inventory, or we can figure out the IP address at runtime. Preparing the inventory is often simple, however, it requires manual intervention. We have to run tasks to boot an EC2 instance with the required parameters and copy the IP address of the instance to the inventory file. After this, we can run the playbook for deploying the application.

Manually adding IPs to the inventory is not possible for unattended setups. In certain cases, the infrastructure is dynamic to the extent that managing IPs might not even be possible. For such cases, there are two possibilities: we can use Ansible's `add_host` module to deploy an application when we boot up a new instance without adding anything manually anywhere. Or we can use dynamic inventory, where we query the API of the cloud provider and build an in-memory inventory every time. For this chapter, we are going to look at using the `add_host` module and discuss the dynamic inventory in the next chapter.

How to do it...

1. In a previous recipe, we created an Elastic IP and registered the value of the output. We are going to use that and add it to a group called `phonebook-infra`:

```
- name: Adding Elastic IP to the phonebook-infra
  add_host:
    hostname: "{{ eip.public_ip }}"
    groups: phonebook-infra
```

2. Once we are done with the `add_host` task in the EC2 role, our playbook can deploy the application by calling both the EC2 role and `phonebook` role like this:

```
---
- hosts: localhost
  roles:
    - ec2
- hosts: phonebook-infra
  roles:
    - phonebook
```

To understand the `phonebook` role better, we can refer to [Chapter 1, *Getting Started with Ansible and Cloud Management*](#) of this book. This playbook will create an EC2 instance and deploy the `phonebook` application on that instance. Since we are using a single instance, we will use SQLite as the database.

3

Managing Amazon Web Services with Ansible

In this chapter, we will cover the following recipes:

- Creating an RDS instance
- Creating and deleting records in Route53
- Managing S3 objects
- Managing Lambda
- Managing IAM users
- Using dynamic inventory
- Deploying the sample application

Introduction

Amazon Web Services offers a wide range of services, other than EC2, to host our applications in a most advanced and secured fashion. In this chapter, we will be discussing a few of the key services required for deploying a production-class application:

- **Relational Database Service (Amazon RDS):** A hosted relational database service from AWS in an isolated environment. It provides several database engines, such as MySQL, PostgreSQL, Oracle, and Microsoft SQL Server.
- **Route53:** A scalable **domain name system (DNS)** by AWS.

- **Simple Storage Service (S3):** An object storage service by AWS. S3 provides high durability, compliance capabilities, and security. It is widely used for creating CDN backends, backups, and storing log files, as well as hosting static websites.
- **Lambda:** AWS Lambda is an event-driven, serverless platform for cloud computing. It's a computing service that involves a lambda function to perform an action based on received events.
- **Identity Access Management (IAM users):** The IAM service helps administrators to manage access for multiple groups of people and applications accessing AWS resources.
- **Dynamic inventory:** The dynamic nature of the cloud comes with its own limitations and use cases, but with the help of dynamic inventory in Ansible, we can target hosts with AWS tagging, resource IDs, AZ, regions, hostnames, and so on.

We will need to configure Ansible in the exact same way as described in [Chapter 2, Using Ansible to Manage AWS EC2](#), under the recipe *Preparing Ansible to work with AWS*.

Creating an RDS instance

Amazon Web Services, along with compute, offers a **relational database service (RDS)**. The RDS makes it easier to set up, scale, and operate relational databases in the cloud.

Getting ready

Before we get started with Amazon RDS, we need to familiarize ourselves with a few terms:

- **DB instance:** A DB instance is a basic component of Amazon RDS. It's an isolated database engine in the cloud. It can be accessed using standard clients, provided our security group allows for it. Amazon RDS has different offerings in terms of memory and computation power, which is determined by its `DBInstance` class.
- **Regions and availability zones:** AWS offers various hosting options across the globe called regions and multiple availability zones inside each region (also referenced as different physical data centers within the same region). This feature enables us to deploy highly available services.
- **Security groups:** We have discussed security groups in the previous chapter. Security groups can also be attached to a DB instance and restrict access to limited IP addresses, subnets, or even instances in other security groups.

- **DB Parameters Groups:** In order to configure DB engine parameters, RDS provides a service called DB Parameter Groups, which allows us to configure DB engine parameters and use the same DB Parameter Group with multiple DB instances.

How to do it...

We will be using the RDS module to create an RDS instance. We will be creating a MySQL database instance in our task:

```
- name: Create RDS Instance
  rds:
    aws_access_key: "{{ access_key }}"
    aws_secret_key: "{{ secret_key }}"
    region: "{{ aws_region }}"
    command: create
    instance_name: my-first-rds-instance
    db_engine: MySQL
    size: 10
    instance_type: db1.m1.small
    username: cookbook-admin
    password: koobkooc
    security_groups: "{{ my_first_sg.group_id }}"
    multi_zone: yes
    backup_retention: 7
    tags:
      Environment: cookbook-prod
      Application: cookbook-test
  tags:
    - recipe1
```

How it works...

The preceding task will create (using a parameter command) a MySQL database instance with a disk size of 10 GB and a compute size of `m1.small`. The instance type defines the computation power of the instance, which can be changed depending upon the use case. We have also defined a few parameters that ensure the admin user is created with the appropriate credentials and an appropriate security group is applied. We have also defined a backup retention window. This ensures that we can restore the database, if required, from anywhere up to the last seven days of backup.

We should note that the security group we used here is the same security group that we generated in the previous chapter. One of the great features of an RDS instance is its high availability; even during maintenance (software or hardware), RDS can reduce the risks of downtime significantly by using a multi-availability zone (multi-az) feature. With multi-az, AWS runs multiple replicas of the database instance in different availability zones that can be promoted to master seamlessly during maintenance, downtime, or reboots of the master.

Creating and deleting records in Route53

Route53 is a scalable and highly available domain resolution service from AWS. With Route53, we can manage DNS records using the AWS console, as well as APIs, which enable us to automate several tasks where frequent changes in DNS records might be required. In this recipe, we will create a Route53 record and delete an existing Route53 record.

Getting ready

Before we do that, let us shed some light on a few keywords:

- **DNS zone:** This refers to a certain part of the existing global DNS name. Every DNS zone represents a boundary over certain DNS entities, the authority of which belongs to a legal entity or an organization.
- **DNS type:** There are various types of DNS record, for example, a type A record and a type CNAME record. An A type record maps a name to an IP address and a CNAME records the map's name with another name.

How to do it...

We can use the Route53 module for creating, retrieving, and deleting a Route53 record. In the following task, we have assumed that the DNS zone `example.com` belongs to us and has been registered with the AWS Route53 service:

1. To create a Route53 record, do the following:

```
- name: Create Route53 record
  route53:
    state: present
    zone: example.com
```



```
record: app.example.com
type: A
ttl: 7200
value:
  - 1.1.1.1
  - 2.2.2.2
  - 3.3.3.3
tags:
  - recipe2
```

2. To get an existing Route53 record, do the following:

```
- name: Get existing Route53 record
route53:
  state: get
  zone: example.com
  record: dns.example.com
  type: A
  register: record
tags:
  - recipe2
```

3. To delete a Route53 record, do the following:

```
- name: Delete existing Route53 record
route53:
  state: absent
  zone: example.com
  record: "{{ record.set.record }}"
  ttl: "{{ record.set.ttl }}"
  type: "{{ record.set.type }}"
  value: "{{ record.set.value }}"
tags:
  - recipe2
```

How it works...

In the preceding task, we are creating a new type A DNS record `app.example.com` under zone `example.com` and mapping it to multiple IP addresses.

We have used the parameter **ttl (time to live)**, which can be seen as the lifespan or lifetime that a record can stay with the client without initiating a new DNS query for the same DNS record.

In the second task, we are retrieving an existing DNS record using a parameter called `state` with the input value `get` and registering the value we obtain in the variable name `record`. And later, in the third task, once we have that DNS record information in the variable, using the `state` parameter with the value `absent`, we delete that DNS record.

We should note that once we register the DNS record into the variable called `record`, the details of the required DNS record will change to the following:

- Record as `set.record`
- Ttl as `set.ttl`
- Type as `set.type`
- Value as `set.value`



We should note here that the preceding tasks will fail if they are executed without modifying the correct value of the `zone` parameter in the task.

Managing S3 objects

AWS **Simple Storage Service (S3)** is a storage service that can be used as storage on the internet. It allows us to download and upload any amount of data any time, with various use cases such as storing videos, images, documents, log files, assets (such as images, JavaScripts, and CSS) of a website, and so on.

S3 allows us to store our documents in private and public modes, which enables us to store secret data over S3 for personal use, as well as files that are publicly available. S3 uses object storage instead of flat file systems. Object storage is storage designed to store data as an object. An object contains data and metadata, allowing us to store unstructured data.

In this recipe, we will be creating an S3 object using the `PUT` operation and then we will retrieve that object with the `GET` operation. Amazon S3 stores our data as objects inside buckets.

How to do it...

We will use the S3 module for creating an empty bucket with public read permission, and then we will store our file as a data object in that bucket:

1. To create an S3 bucket, do the following:

```
#Creating an empty s3 bucket
- name: Creating S3 bucket
  s3:
    bucket: my_first_bucket
    mode: create
    permission: public-read
  tags:
    - recipe3
```

2. Put the object in the S3 bucket:

```
#Putting object
- name: Put object in S3 bucket
  s3:
    bucket: my_first_bucket
    object: /location/of/text.txt
    src: file.txt
    mode: put
  tags:
    - recipe3
```

3. Get the object:

```
#Retrieving an object
- name: Get object
  s3:
    bucket: my_first_bucket
    object: /location/of/text.txt
    dest: file.txt
    mode: get
  tags:
    - recipe3
```

How it works...

In the preceding tasks, we are creating a bucket and naming it `my_first_bucket`, and making it publicly readable using the permission `public-read`. In the next task, we are storing our data file present in the files of our role object inside that newly created bucket.

We should note that we are using a parameter called `object`, which is the key of the object (virtual folders inside the bucket).

In the last task, we are retrieving the same object created in the second task, present at the same location inside the bucket (key), and storing that object in the destination file `file.txt`.



We have used the bucket name `my_first_bucket`, which needs to be globally unique; executing this task without defining a unique bucket name will result in a failure.

Managing Lambda

AWS Lambda is a serverless compute service, also known as function as a service, which can be invoked in response to events. AWS Lambda functions can be invoked for various events generated by multiple AWS services, such as the creation of an S3 object, running an EC2 instance, or receiving HTTP requests through an Amazon API gateway. AWS Lambda currently supports many popular programming languages, such as Java, Node.js, C#, and Python. AWS Lambda also lets us control various platform parameters, such as the allowed execution time, memory utilization, and environment variables.

In this recipe, we will be creating a lambda function that will print `Hello World` on its execution. To keep this task simple, we will be executing a lambda function using the Ansible module `execute_lambda`.

Getting ready

We should understand the following concepts involved in AWS Lambda before we jump into writing a task for our lambda function.

- **Packaging our code for Lambda:** AWS Lambda accepts `.zip` format packages, which includes code and libraries associated with code. We will be using Python 2.7 as our programming language to print a simple `Hello World`.

- **Role for Lambda:** If our lambda function needs to access any other AWS resource, such as S3, RDS, and so on, then roles come in handy. We will be using an existing IAM role here called `first_iam_role` and discuss it in more detail in the upcoming recipe *Managing IAM users*.
- **Handler:** This is the entry point for the execution of the lambda function. It's a way to tell Lambda which function is our main one. For example, if we create a file called `hello_world.py` and define a function called `my_handler`, then our handler will look like `hello_world.my_handler`.
- **Runtime:** This is the environment for our lambda function. AWS Lambda supports multiple platforms, as we discussed earlier. In our case, this will be Python 2.7.

How to do it...

1. Before we start writing our task, let's start with our code first:

```
#hello_world.py
def my_handler(event, context):
    return "Hello World"
```

2. We will use the `lambda` module to create our first lambda function:

```
- name: Creating first lambda function
  lambda:
    name: MyFirstLambda
    state: present
    zip_file: myfirstzip.zip
    runtime: 'python2.7'
    role: 'arn:aws:iam::<account-no>:role/lambda'
    handler: 'hello_world.my_handler'
    vpc_subnet_ids:
      - "{{ my_private_subnet.subnet.id }}"
    vpc_security_group_ids:
      - "{{ my_first_sg.group_id }}"
  tags:
    - recipe4
```

3. Now we will execute our lambda function using the `execute_lambda` module:

```
- name: executing lambda function
  execute_lambda
    name: MyFirstLambda
    payload:
      foo: test
      value: hello
    register: response
  tags:
    - recipe4
```

4. We can use the debug module of Ansible to verify the response generated by the lambda function, as shown in the following example:

```
- debug: msg="{{ response }}"
```

How it works...

In *step 1*, we have written a basic Python program to print `Hello World`, which will be our lambda function. We should note here that our program requires two parameters as arguments that provide useful information to our program. The context provides information about the platform to our function that can be used in our program for various use cases, such as memory information, the remaining time to execute, and so on, and the event is the object that provides information about the event that invoked our lambda function, which could be a list, str, dictionary or hash, or `NoneType`. Once we have our code ready, we will ZIP it and name our package `myfirstzip.zip`.

In *step 2*, we have created our lambda function with the name `MyFirstLambda` and packaged the code we created previously. We are using subnet IDs and the security group IDs created in [Chapter 2, Using Ansible to Manage AWS EC2](#).

In *step 3*, we have invoked our lambda function. We are using a dummy payload here to test our lambda function using the parameter `payload`. This payload will be serialized automatically during the execution of the lambda function. We are storing the response generated by the lambda function in the variable called `response`.

Managing IAM users

AWS **Identity and Access Management (IAM)** enables the AWS administrator to control access across AWS resources in a more efficient and managed manner. IAM allows us to create multiple users in an AWS account with different access levels and privileges. IAM users can be allowed to access AWS through the web console and API. In this recipe, we will be creating IAM users and attaching policies that define access for those users. We will also create IAM roles that can be applied to various AWS services, such as Lambda, EC2, and so on.

How to do it...

1. We will be using the IAM module to create IAM users. In the following task, we will be creating an IAM user. We will also set a password for the user being created. We will be storing this information in our `secrets.yml`, which is protected by Ansible Vault. We have discussed Ansible vault in the [Chapter 1, Getting Started with Ansible and Cloud Management](#), and in [Chapter 2, Using Ansible to Manage AWS EC2](#):

```
- name: Create IAM users
  iam:
    iam_type: user
    name: "{{ item }}"
    state: present
    password: "{{ iam_pass }}"
  with_items:
    - cookbook-admin
    - cookbook-two
  tags:
    - recipe5
```

2. Let's first create a JSON policy for providing access to all AWS resources in the AWS account. Note that we are only giving this policy as an example; we certainly do not recommend using this unless we want unrestricted access to be granted:

```
{
  "Version": "2017-10-01",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "*",
```

```
        "Resource": "*"
    }
  ]
}
```

3. Let's save this as `iam_admin.json`:

```
- name: Assign IAM policy to user
  iam_policy:
    iam_type: user
    iam_name: cookbook-admin
    policy_name: Admin
    state: present
    policy_document: iam_admin.json
```

How it works...

In *step 1*, we should note that `iam_pass` is a variable defined in our variables file, protected by Ansible Vault for storing secrets.

In *step 2*, we wrote an IAM policy and saved that file with the name `iam_admin.json`.

In *step 3*, we created an IAM policy with the name `Admin` and attached that policy to the user `cookbook-admin`.

We should note here that we have used `iam_type` as a user here, which can be changed to a group or role. In a production environment, it's good practice to create IAM groups with attached policies and add users to the group. Also, changing `iam_type` to a role will create a role with a defined policy and can be used with various AWS resources.

Using dynamic inventory

In our previous recipes and the previous chapter, we have seen how easily we can create AWS resources. When running production loads on AWS Cloud, we tend to scale out and scale in the resources according to traffic and workloads. But with this dynamic nature of the cloud to scale in and out, managing a static inventory file for Ansible is an operational task and overhead, and it becomes really difficult if we want to implement features such as autoscaling.

With the help of dynamic inventory, we can solve this. Ansible provides an EC2 dynamic inventory script called

`ec2.py` (<https://github.com/ansible/ansible/blob/devel/contrib/inventory/ec2.py>).

How to do it...

1. We will be running Ansible ad hoc commands using the EC2 dynamic inventory and shell module to execute `/bin/uname -a`:

```
ansible -i ec2.py tag_Name_my_first_instance -m shell -a  
"/bin/uname -a"
```

2. Similarly, we can use the dynamic inventory to target the following specific features:

- Instance IDs

```
ansible -i ec2.py <instance-id> -m shell -a "/bin/uname -a"
```

- Regions

```
ansible -i ec2.py us-east-1 -m shell -a "/bin/uname -a"
```

- All

```
ansible -i ec2.py all -m shell -a "/bin/uname -a"
```

3. AWS dynamic inventory, generated by `ec2.py`, is grouped by default with various parameters, for example, tags, instance_ids, regions, and so on. We can also use this same approach for running an Ansible playbook (with `hosts: us-east-1`), for example:

```
ansible-playbook -i ec2.py my_sample.yml
```

We should note that `ec2.py` uses Boto for connecting AWS Cloud, which will be using the Boto profile for AWS credentials. We can store these credentials under `~/.aws/credentials`.

Deploying the sample application

In this section, we will be deploying a sample phonebook application using AWS Cloud resources, such as RDS for the database instance and EC2 as our compute server where our code will be deployed.

How to do it...

This sample application will be deployed using a playbook, as follows:

```
---
- hosts: localhost
  roles:
    - phonebook
```

How it works...

This application will use the same password that we used while creating a DB instance for creating a database and the user for our application. This sample application playbook will create and provision an AWS EC2 instance and will deploy the application on it. Along with this, it will create an RDS instance with the master password defined in our secrets protected by Ansible Vault. After creating an RDS instance, it will create a database with a user for the application.

Once our application is deployed, we will be able to access the phonebook application on port 8080 of our AWS instance.

4

Exploring Google Cloud Platform with Ansible

In this chapter, we will cover the following recipes:

- Preparing to work with Google Cloud Platform
- Creating GCE instances
- Attaching persistent disks
- Creating snapshots for backup
- Tagging an instance
- Managing network and firewall rules
- Managing load balancer
- Managing GCE images
- Creating instance templates
- Creating managed instance groups
- Managing objects in Google Cloud Storage
- Creating a Cloud SQL instance (without Ansible module)
- Using dynamic inventory
- Deploying the phonebook application

Introduction

Google Cloud Platform is one of the largest and most innovative cloud providers out there. Google Cloud is used by various industry leaders such as Coca-Cola, Spotify, and Philips. Amazon Web Services and Google Cloud are always involved in a price war, which benefits consumers greatly. Google Cloud Platform covers 12 geographical regions across four continents with new regions coming up every year. Google Compute Engine, abbreviated as GCE, is one of the most popular components of Google Cloud. GCE can help users to:

- Create a virtual machine
- Create virtual machine templates
- Create and manage a group of virtual machines
- Create and manage backups as snapshots
- Manage health checks

Other than GCE, we are also going to cover network services and Google Cloud Storage in this chapter. Unfortunately, Ansible modules for Google Cloud Platform are not as comprehensive as the modules for Amazon Web Services. We will see what to do if we encounter a missing module.

Preparing to work with Google Cloud Platform

Among various Google Cloud modules, a bunch of Python modules are required to make sure the Ansible modules work smoothly. These modules are as follows:

- `apache-libcloud`
- `boto`
- `google-api-python-client`
- `google-auth`
- `google-auth-httpplib2`

These libraries can be obtained by using Python's `pip` tool:

```
$ pip install apache-libcloud boto google-api-python-client google-auth  
google-auth-httpplib2
```

Along with these libraries, we need to get credentials from Google Cloud. All of the modules, except for the one used for Google Cloud Storage, can use a service account credential JSON file. Service accounts are special accounts typically associated with a non-human entity like a server or an application. This account can be used by applications and servers to interact with various APIs provided by Google Cloud. The advantage of using Service Account is that it helps us to avoid embedding any user credentials in any way, improving security. Let us create a service account for Ansible modules.

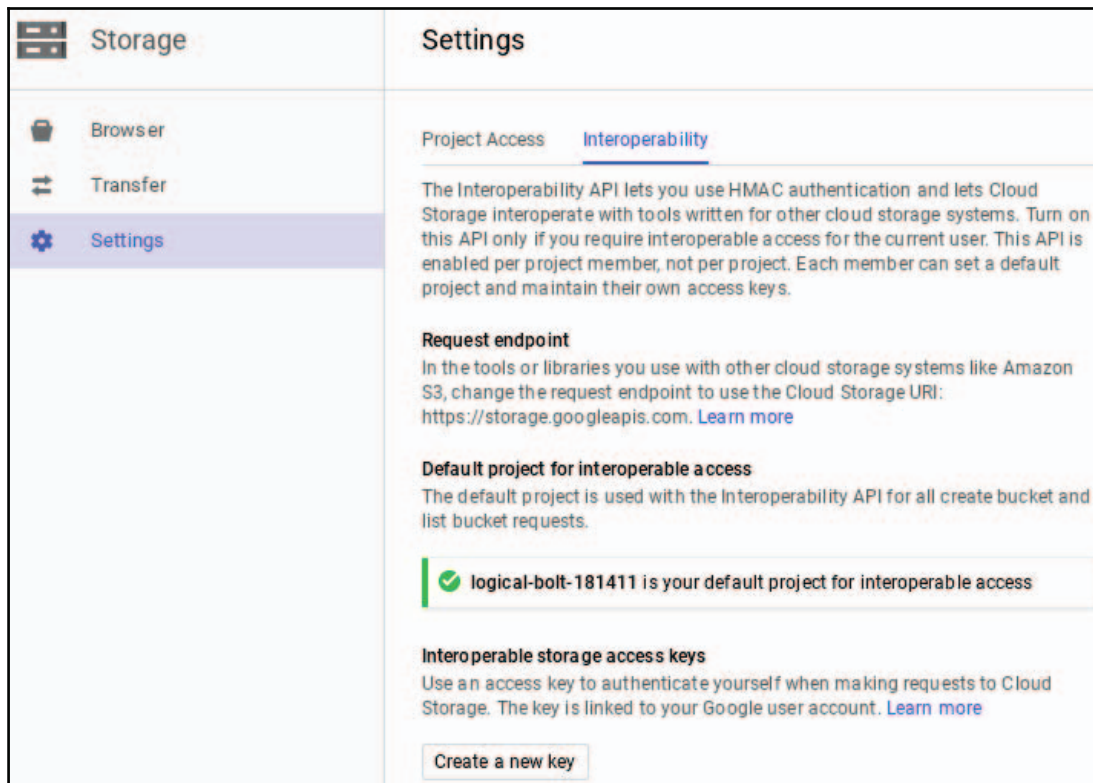
How to do it...

1. We should open the Google Cloud dashboard and select our project from the top header.
2. After that, from the left sidebar, select **IAM & Admin** and then go to the **Service Accounts** section. Here we can create a new service account by clicking on the **Create Service Account** button at the top. From the pop-up, we can create a service account and download the JSON credentials file:



The screenshot shows the 'Create service account' dialog box. It has a title bar 'Create service account'. Below the title bar, there are two input fields: 'Service account name' with the value 'ansible' and 'Role' with a dropdown menu showing 'Editor'. Below these, there is a 'Service account ID' field with the value 'ansible' and a domain field with the value '@logical-bolt-181411.iam.gserviceaccount.com'. Below the ID field, there is a message: 'You don't have permission to furnish a new private key.' followed by a checked checkbox 'Furnish a new private key' and a description: 'Downloads a file that contains the private key. Store the file securely because this key can't be recovered if lost.' Below this, there is a 'Key type' section with two radio buttons: 'JSON' (selected) and 'P12' (unselected). Below the 'P12' option, there is a note: 'For backward compatibility with code using the P12 format'. Below the key type section, there is another message: 'You don't have permission to modify the domain-wide delegation setting You don't have permission to modify the product name for the consent screen' followed by an unchecked checkbox 'Enable G Suite Domain-wide Delegation' and a description: 'Allows this service account to be authorized to access all users' data on a G Suite domain without manual authorization on their part. Learn more'. At the bottom right, there are two buttons: 'CANCEL' and 'CREATE'.

3. We should keep the JSON file safely since it is not possible to recover it, if it is lost. However, we can create new credentials, if required. We are saving it in our home directory `/home/packt/gce-default.json`.
4. For Google Cloud Storage, which uses Boto library, we need an access key and a secret key just like Amazon Web Service. To obtain that, we need to go to the **Storage** option on the left sidebar on the Google Cloud dashboard and select **Settings**. A button on the bottom of the page, named **Create a new key**, will get us the key pair that we want:



5. Once we have the credentials, we should put them in `roles/gce/vars/secrets.yml`:

```
---
gs_access_key: GOOG4TSK2MSO2RDQGMXT
gs_secret_key: 49I6NhRHKFEhfw2nopr9YZNJcfrzQqmilJUVB1Vf
```

6. Now, we encrypt them:

```
$ ansible-vault encrypt roles/gce/vars/secrets.yml
```

7. Before we can start creating resources in Google Cloud, we will add some variables in `roles/gce/vars/main.yml` which will define connection parameters and JSON credentials details:

```
---
service_account_email: "ansible@logical-
bolt-181411.iam.gserviceaccount.com"
credentials_file: "/home/packt/gce-default.json"
zone: "us-west1-a"
project_id: "logical-bolt-181411"
```

The `service_account_email` is the ID of the service account that we created just now and the `credentials_file` is the JSON credentials file that we downloaded. We are going to boot our resources in `us-west1-a`. Lastly, `project_id` defined the project that we will connect to.

Creating GCE instances

Creating a virtual machine is the most basic building block of the GCE. A *default* network is created for us to boot the virtual machine in. However, in many cases, we would want to create a separate virtual network for the sake of isolation.

How to do it...

1. Let us start by creating a network in the `us-west1` region where we will boot our instances:

```
- name: Create Custom Network
  gce_net:
    name: my-network
    mode: custom
    subnet_name: "public-subnet"
    subnet_region: us-west1
    ipv4_range: '10.0.0.0/24'
    state: "present"
    service_account_email: "{{ service_account_email }}"
    project_id: "{{ project_id }}"
```

```
    credentials_file: "{{ credentials_file }}"
tags:
- recipe2
```

2. Now we will get the static IP:

```
- name: create public ip
  gce_eip:
    name: app
    region: us-west1
    state: present
    service_account_email: "{{ service_account_email }}"
    project_id: "{{ project_id }}"
    credentials_file: "{{ credentials_file }}"
  register: app_eip
tags:
- recipe2
```

3. The preceding task will return the external IP address which we have saved in the variable `app_eip`. Once we have the external IP, we can create and start our virtual machine:

```
- name: create and start the instance
  gce:
    instance_names: app
    zone: "{{ zone }}"
    machine_type: f1-micro
    image: centos-7
    state: present
    disk_size: 15
    tags: http
    metadata: "{{ instance_metadata }}"
    network: my-network
    subnetwork: public-subnet
    external_ip: "{{ app_eip.address }}"
    service_account_email: "{{ service_account_email }}"
    credentials_file: "{{ credentials_file }}"
    project_id: "{{ project_id }}"
tags:
- recipe2
```


4. Further, we specify the network details and the access requirements like credentials. We will update our `roles/gce/vars/main.yml` to add

`instance_metadata:`

```
instance_metadata: '{"sshKeys":"aditya:ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQACbg83WYIxUfXWJ4bQiYfZYHceDwMJxnGfJqgY
tHL/DategVY+Nm8MX3CRZYisfskt0m9CQ6y/Ux10ITTz+O11fgxLJcroZmKJbWW0K39
gfhVfQr7FIe0zuJaxqUQUuyc0i6RCBRiZPiQQOPes2yDtHfHgDWx3q9knS3ZXIAXcGLZ
rgfC1XnIK8CLAnZDved9Rue2bhsCnO9Mleh9g/CTtehMDAzD4NeSv9eETlHYkYSpJg8
gFA3BFICpBxTqWSjf1mMQGSmiudFOhRjHIxL1Tvh"}'
```

How it works...

In *step 1*, we have supplied credentials, network name, subnet name, IP range and region. Now we have a network named `my-network` and a subnet within that named `public-subnet`. GCE instances, by default, start up with an ephemeral external public IP. However, that IP can change between instance reboot which might not be desirable at times. We can request a static external IP and attach it to our instance and we did so in *step 2*. We booted an instance in *step 3*. For that we had to specify a name for the instance. We booted an f1-micro instance running CentOS 7 with a 15 GB disk attached to it. We also specified tags which would help us in organizing our instances as well as in managing firewall rules. The metadata helps to specify the SSH public key so that we can SSH into instances or run Ansible playbooks for further configuration. *Step 4* creates a user `aditya` and specifies a SSH key. Now we have the instance ready, we can use SSH to configure it further.

Attaching persistent disks

Along with the disk attached as the bootable disk, we can attach additional disks to an instance. Since the size of these disks can be easily increased as per requirement, and can be detached and reattached to another instance, they provide great flexibility in terms of usage. We have seen them being used in all sorts of databases and network storage's like NFS.

How to do it...

Let us create a 10 GB standard persistent disk and mount it on a virtual machine:

```
- name: attach persistent disk
  gce_pd:
    disk_type: pd-standard
    size_gb: 10
    instance_name: app
    zone: "{{ zone }}"
    service_account_email: "{{ service_account_email }}"
    credentials_file: "{{ credentials_file }}"
    project_id: "{{ project_id }}"
    name: app-pd
  tags:
  - recipe3
```

How it works...

The task creates a 10 GB standard persistent disk, the other option being a SSD disk and it attaches the disk to the app. We can log into the app instance to format the disk and use it. Typically the disk can be found as `/dev/sdb`, `/dev/sdc`, and so on.

Creating snapshots for backup

Often we will need to create snapshots of instances. These can be used as backup or to create more instances to scale up. Typically, a daily or weekly cron of the Ansible task would help in maintaining a periodic backup.

How to do it...

We can use the `gce_snapshot` module to create a snapshot:

```
- name: create snapshot of instance
  gce_snapshot:
    instance_name: app
    snapshot_name: app-snapshot
    state: present
    service_account_email: "{{ service_account_email }}"
    project_id: "{{ project_id }}"
    credentials_file: "{{ credentials_file }}"
```

```
tags:
- recipe4
```

How it works...

In this case, we just need to specify the instance name and the snapshot name. When running this as a cron, we can set the snapshot name to the current date to maintain periodic backup. One thing to note here is that snapshots are incremental. So they consume very little space and thus, save significant cost on storage.

Tagging an instance

Tagging an instance is useful for organization as well as it is helpful while creating firewall rules. When we start using dynamic inventory, then tagging will be helpful in running playbooks against a certain set of instances tagged with a certain string.

How to do it...

Tagging an instance is straightforward using the `gce_tag` module:

```
- name: set tag for the instance
  gce_tag:
    instance_name: app
    tags: http
    zone: "{{ zone }}"
    state: present
    pem_file: "{{ credentials_file }}"
    service_account_email: "{{ service_account_email }}"
    project_id: "{{ project_id }}"
  tags:
  - recipe5
```

Here we have just tagged the instance as `http`. We can specify any number of tags by giving a comma separated list.

Managing network and firewall rules

By default, inbound connections are not allowed to any of the instances. One way to allow the traffic is by allowing incoming connections to a certain port of instances carrying a particular tag. For example, we can tag all the web servers as `http` and allow incoming connections to port 80 and 8080 for all the instances carrying the `http` tag.

How to do it...

1. We will create a firewall rule with source tag using the `gce_net` module:

```
- name: Create Firewall Rule with Source Tags
  gce_net:
    name: my-network
    fwname: "allow-http"
    allowed: tcp:80,8080
    state: "present"
    target_tags: "http"
    subnet_region: us-west1
    service_account_email: "{{ service_account_email }}"
    project_id: "{{ project_id }}"
    credentials_file: "{{ credentials_file }}"
  tags:
- recipe6
```

2. Using tags for firewalls is not possible all the time. A lot of organizations whitelist internal IP ranges or allow office IPs to reach the instances over the network. A simple way to allow a range of IP addresses is to use a source range:

```
- name: Create Firewall Rule with Source Range
  gce_net:
    name: my-network
    fwname: "allow-internal"
    state: "present"
    src_range: ['10.0.0.0/16']
    subnet_name: public-subnet
    allowed: 'tcp'
    service_account_email: "{{ service_account_email }}"
    project_id: "{{ project_id }}"
    credentials_file: "{{ credentials_file }}"
  tags:
- recipe6
```

How it works...

In *step 1*, we have created a firewall rule called `allow-http` to allow incoming requests to TCP port 80 and 8080. Since our instance `app` is tagged with `http`, it can accept incoming traffic to port 80 and 8080.

In *step 2*, we have allowed all the instances with IP `10.0.0.0/16`, which is a private IP address range. Along with connection parameters and the source IP address CIDR, we have defined the network name and subnet name. We have allowed all TCP connections. If we want to restrict it to a port or a range of ports, then we can use `tcp:80` or `tcp:4000-5000` respectively.

Managing load balancer

An important reason to use a cloud is to achieve scalability at a relatively low cost. Load balancers play a key role in scalability. We can attach multiple instances behind a load balancer to distribute the traffic between the instances. Google Cloud load balancer also supports health checks which helps to ensure that traffic is sent to healthy instances only.

How to do it...

Let us create a load balancer and attach an instance to it:

```
- name: create load balancer and attach to instance
  gce_lb:
    name: loadbalancer1
    region: us-west1
    members: ["{{ zone }}/app"]
    httphealthcheck_name: hc
    httphealthcheck_port: 80
    httphealthcheck_path: "/"
    service_account_email: "{{ service_account_email }}"
    project_id: "{{ project_id }}"
    credentials_file: "{{ credentials_file }}"
  tags:
  - recipe7
```

For creating a load balancer, we need to supply a comma separated list of instances. We also need to provide health check parameters including a name, a port and the path on which a GET request can be sent.

Managing GCE images

Images are a collection of a boot loader, operating system, and a root filesystem. There are public images provided by Google and various open source communities. We can use these images to create an instance. GCE also provides us capability to create our own image which we can use to boot instances.

It is important to understand the difference between an image and a snapshot. A snapshot is incremental but it is just a disk snapshot. Due to its incremental nature, it is better for creating backups. Images consist of more information such as a boot loader. Images are non-incremental in nature. However, it is possible to import images from a different cloud provider or datacenter to GCE.

Another reason we recommend snapshots for backup is that taking a snapshot does not require us to shut down the instance, whereas building an image would require us to shut down the instance. Why build images at all? We will discover that in subsequent sections.

How to do it...

1. Let us create an image for now:

```
- name: stop the instance
  gce:
    instance_names: app
    zone: "{{ zone }}"
    machine_type: f1-micro
    image: centos-7
    state: stopped
    service_account_email: "{{ service_account_email }}"
    credentials_file: "{{ credentials_file }}"
    project_id: "{{ project_id }}"
    disk_size: 15
    metadata: "{{ instance_metadata }}"
  tags:
- recipe8

- name: create image
  gce_img:
    name: app-image
    source: app
    zone: "{{ zone }}"
    state: present
    service_account_email: "{{ service_account_email }}"
```

```
    pem_file: "{{ credentials_file }}"
    project_id: "{{ project_id }}"
  tags:
  - recipe8

- name: start the instance
  gce:
    instance_names: app
    zone: "{{ zone }}"
    machine_type: f1-micro
    image: centos-7
    state: started
    service_account_email: "{{ service_account_email }}"
    credentials_file: "{{ credentials_file }}"
    project_id: "{{ project_id }}"
    disk_size: 15
    metadata: "{{ instance_metadata }}"
  tags:
  - recipe8
```

How it works...

In these tasks, we are stopping the instance first and then creating the image. We just need to supply the instance name while creating the image, along with the standard connection parameters. Finally, we start the instance back. The parameters of these tasks are self-explanatory.

Creating instance templates

Instance templates define various characteristics of an instance and related attributes. Some of these attributes are:

- Machine type (f1-micro, n1-standard-1, custom)
- Image (we created one in the previous tip, app-image)
- Zone (us-west1-a)
- Tags (we have a firewall rule for tag http)

How to do it...

Once a template is created, we can use it to create a managed instance group which can be auto-scale based on various parameters. Instance templates are typically available globally as long as we do not specify a restrictive parameter like a specific subnet or disk:

```
- name: create instance template named app-template
  gce_instance_template:
    name: app-template
    size: f1-micro
    tags: http,http-server
    image: app-image
    state: present
    subnetwork: public-subnet
    subnetwork_region: us-west1
    service_account_email: "{{ service_account_email }}"
    credentials_file: "{{ credentials_file }}"
    project_id: "{{ project_id }}"
  tags:
  - recipe9
```

We have specified the machine type, image, subnets, and tags. This template can be used to create instance groups.

Creating managed instance groups

Traditionally, we have managed virtual machines individually. Instance groups let us manage a group of identical virtual machines as a single entity. These virtual machines are created from an instance template, like the one which we created in the previous tip. Now, if we have to make a change in instance configuration, that change would be applied to all the instances in the group.

How to do it...

Perhaps, the most important feature of an instance group is auto-scaling. In event of high resource requirements, the instance group can scale up to a predefined number automatically:

```
- name: create an instance group with autoscaling
  gce_mig:
    name: app-mig
    zone: "{{ zone }}"
```



```
service_account_email: "{{ service_account_email }}"
credentials_file: "{{ credentials_file }}"
project_id: "{{ project_id }}"
state: present
size: 2
named_ports:
- name: http
  port: 80
template: app-template
autoscaling:
  enabled: yes
  name: app-autoscaler
  policy:
    min_instances: 2
    max_instances: 5
    cool_down_period: 90
    cpu_utilization:
      target: 0.6
    load_balancing_utilization:
      target: 0.8
tags:
- recipe10
```

How it works...

The preceding task creates an instance group with an initial size of two instances, defined by `size`. We have named port 80 as `HTTP`. This can be used by other GCE components to route traffic. We have used the template that we created in the previous recipe. We also enable autoscaling with a policy to allow scaling up to five instances. At any given point, at least two instances would be running. We are scaling on two parameters, `cpu_utilization`, where 0.6 would trigger scaling after the utilization exceeds 60% and `load_balancing_utilization` where the scaling will trigger after 80% of the requests per minutes capacity is reached. Typically, when an instance is booted, it might take some time for initialization and startup. Data collected during that period might not make much sense. The parameter, `cool_down_period`, indicates that we should start collecting data from the instance after 90 seconds and should not trigger scaling based on data before.

Managing objects in Google Cloud Storage

Google Cloud Storage is an object storage tool which can handle various kinds of data, from live data to data archiving at various levels of redundancy. In its simplest form, Google Cloud Storage can be used to store and retrieve data from storage buckets, similar to AWS S3. Google Cloud Storage is present in five continents (North and South Americas, Europe, Asia, and Australia). For data that needs quick retrieval, we can store data as close to our clients as possible while for disaster recoveries, we can store data in a different continent altogether (subject to compliance restrictions). Over a period of time, we have observed two very popular use cases of Cloud Storage:

- Store assets like images and JavaScripts in a bucket and designate that bucket as a backend for a CDN service.
- Upload periodic backups of databases (often as an encrypted blob) in a different continent.

Basic workflow of both of these use cases are quite similar; create a bucket and upload a file. Optionally, we can retrieve and delete the files as well.

How to do it...

1. Let us start by creating a bucket:

```
- name: create a bucket
  gc_storage:
    bucket: packt-mybucket
    mode: create
    gs_access_key: "{{ gs_access_key }}"
    gs_secret_key: "{{ gs_secret_key }}"
  tags:
  - recipe11
```

2. We created a bucket named `packt-mybucket` in previous task. We should note that bucket names are global. This means that if another Google Cloud Storage user has taken a name, then it won't be available to us. Uploading a file is easy enough. We just need to specify the path of the file that we want to upload and the object name which would identify the file in the bucket. The file with name `putmyfile.txt` would be available in the `packt-mybucket` as `key.txt`. By default, the uploaded files are private which is good for backups but might not be desirable for a CDN service. So we can set it as `public-read`, if we want:

```
- name: upload an object
  gc_storage:
    bucket: packt-mybucket
    object: key.txt
    src: /home/packt/ansible/putmyfile.txt
    mode: put
    permission: public-read
    gs_access_key: "{{ gs_access_key }}"
    gs_secret_key: "{{ gs_secret_key }}"
  tags:
  - recipe11
```

3. Downloading an object is as simple as uploading. We need to specify the object that we want to download and supply a path for the same:

```
- name: download an object
  gc_storage:
    bucket: packt-mybucket
    object: key.txt
    dest: /home/packt/ansible/getmyfile.txt
    mode: get
    gs_access_key: "{{ gs_access_key }}"
    gs_secret_key: "{{ gs_secret_key }}"
  tags:
  - recipe11
```

Lastly, we should note that the `gc_storage` module uses Boto to interact with Google Cloud Storage. Technically, Boto is AWS SDK but Google provides certain interoperability capabilities due to which Boto library can interact with Google Cloud Storage.

Creating a Cloud SQL instance (without Ansible module)

Google Cloud has several tools and features for which Ansible modules are not available at the time of writing. In such cases we can use a shell module and Google Cloud SDK's `gcloud` command line tool. One such component of Google Cloud is Cloud SQL which provides managed MySQL and PostgreSQL instances.

How to do it...

Let us build an instance of MySQL using Ansible and gcloud:

1. We will start by configuring the `gcloud` tool. In order to keep our `main.yml` clean, we will move the `gcloud` configuration tasks into a different YAML file, `configure_gcloud.yml`, and import it in `main.yml`. In this YAML file, we have Ansible tasks to take care of installing the Google SDK repository on various Linux distributions, including Red Hat Enterprise Linux, CentOS, Fedora, Debian, and Ubuntu. These need super user privileges. We have included the repository files in the code base. Our `configure_gcloud.yml` would look like this:

```
---
- name: enable Google SDK repo for RedHat/Centos
  copy:
    src: google-cloud-sdk.repo
    dest: /etc/yum.repos.d/google-cloud-sdk.repo
  when: ansible_os_family == "RedHat"

- name: enable Google SDK repo for Debian/Ubuntu
  template:
    src: google-cloud-sdk.list
    dest: /etc/apt/sources.list.d/google-cloud-sdk.list
  when: ansible_os_family == "Debian"

- name: install Google Cloud's public key for Debian/Ubuntu
  apt_key:
    url: https://packages.cloud.google.com/apt/doc/apt-key.gpg
    state: present
  when: ansible_os_family == "Debian"

- name: install gcloud SDK
  package:
    name: google-cloud-sdk
    state: present

- name: configure gcloud with service account
  command: gcloud auth activate-service-account --key-
file=/home/packt/gce-default.json
```

2. Let us include the `configure_gcloud.yml` in `main.yml`:

```
- name: Configure Google Cloud SDK
  import_tasks: configure_gcloud.yml
  tags:
  - recipe12
```

3. Now that we have the `gcloud` tool configured, let us start by enabling the API to manage the Google Cloud SQL instance:

```
- name: enable Cloud SQL APIs
  command: gcloud service-management enable sqladmin.googleapis.com
  tags:
  - recipe12
```

4. After executing the command to enable the API, we need to wait a couple of minutes because it might take a short time for the changes to take effect:

```
- name: wait for a couple of minutes
  pause:
    minute: 2
  tags:
  - recipe12
```

5. The command to enable the API is idempotent in nature. So we can execute it multiple times without any problems. However, not every task will be idempotent. In such cases we need to build pseudo idempotency by executing tasks based on certain checks and parameters. We need to create a Cloud SQL instance based on a check of availability of the instance. The first task will check if the instance already exists. If not, it will fail with a non-zero return code. In the second task we check for the return code and create an instance, if it does not exist:

```
- name: check if the instance is up
  command: gcloud sql instances describe app-db
  register: availability
  ignore_errors: yes
  tags:
  - recipe12

- name: boot SQL instance if it not present
  command: gcloud sql instances create app-db --tier=db-f1-micro --
region=us-west1 --authorized-networks={{ eip.address }}/32
  when: availability.rc != 0
  tags:
  - recipe12
```

6. Once the instance is created, we can set the password for the root user using `gcloud` tool:

```
- name: set root password
  command: gcloud sql users set-password root % --instance app-db -
- password {{ mysql_root_password }}
tags:
- recipe12
```

Now we are ready to use MySQL Cloud SQL instance. We will be using it while deploying our phonebook application.

Using dynamic inventory

Managing inventory in a text file can be a problem. A large cluster could overwhelm the maintainer of the inventory. On top of that, often the instances can be scaled at short notice. Keeping this in mind, we can use dynamic inventory to build and run playbooks on instances based on specific parameters. A list of instances is generated in real time, eliminating the need for maintaining an inventory manually.

How to do it...

1. Ansible's Github repository has scripts for various infrastructure providers, including a script for Google Compute Engine (<https://github.com/ansible/ansible/blob/devel/contrib/inventory/gce.py>). We need to download this script and configure it to use our credentials:

```
$ curl
https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/gce.py > gce.py

$ chmod +x gce.py
```

2. Once we have set the executable bit, we need to create a `secrets.py` which will consist of details about credentials and connection parameters:

```
GCE_PARAMS = ('ansible@logical-
bolt-181411.iam.gserviceaccount.com', '/home/packt/gce-
default.json')
```

```
GCE_KEYWORD_PARAMS = {'project': 'logical-  
bolt-181411', 'datacenter': 'us-west1'}
```

3. Now, let us test if our dynamic inventory is configured correctly:

```
$ ./gce.py --list  
{  
  "us-west1-a": [  
    "app"  
  ],  
  "f1-micro": [  
    "app"  
  ],  
  "status_running": [  
    "app"  
  ],  
  "tag_app": [  
    "app"  
  ],  
  "network_my-network": [  
    "app"  
  ],  
  "_meta": {  
    "stats": {  
      "cache_used": false,  
      "inventory_load_time": 1.2765889167785645  
    },  
    "hostvars": {  
      "app": {  
        "gce_uuid": "e251d614f1bb1854ae3dace4921bf182b3d0e02f",  
        "gce_public_ip": "35.199.156.156",  
        "ansible_ssh_host": "35.199.156.156",  
        "gce_private_ip": "10.0.0.2",  
        "gce_id": "8721025482999434353",  
        "gce_image": "centos-7-v20171003",  
        "gce_description": null,  
        "gce_machine_type": "f1-micro",  
        "gce_subnetwork": "public-subnet",  
        "gce_tags": [  
          "app",  
          "http"  
        ],  
        "gce_name": "app",  
        "gce_zone": "us-west1-a",  
        "gce_status": "RUNNING",  
        "gce_network": "my-network",  
        "gce_metadata": {
```

```

"sshKeys": "aditya:ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQACbg83WYIxUfXWJ4bQiYfZYHceDwMJxnGfJqgY
tHL/DAtegVY+Nm8MX3CRZYisfskt0m9CQ6y/Ux1OITTz+O11fgxLJcroZmKJbWW0K39
gfhvFqR7FIe0zuJaxqUQUuyc0i6RCBRiZPiQQOPes2yDt fHgDWx3q9knS3ZXIAXcGLZ
rgfC1XnIK8CLAnZDved9Rue2bhsCnO9Mleh9g/CTtehMDAzD4NeSv9eETlHYkYSpJg8
gFA3BFICpBxTqWSjflmMQGSmiudFOhRjHIxL1Tvh+pnjSoL/jrLcP3RtMVuG0ZU0qko
Ats1qpTwmyAJUz9Ts2EeyDJ0tXsFAiOFbmuMd"
}
}
},
"tag_http": [
"app"
],
"35.199.156.156": [
"app"
],
"10.0.0.2": [
"app"
],
"centos-7-v20171003": [
"app"
]
}

```

4. The inventory script presents with a wide variety of ways to execute Ansible commands and playbooks on servers based on tags, network, instances size, network zone, and so on. Let us run a simple Ansible command to verify this:

```

$ ansible -i gce.py tag_app -u aditya -m ping
app | SUCCESS => {
  "changed": false,
  "failed": false,
  "ping": "pong"
}

```

5. We can execute playbook by using one of the keys printed by the `./gce.py --list` command in `hosts` key in `playbook`:

```

---
- hosts: tag_app
  roles:
    - myrole

```


Using dynamic inventory, we eliminate the need to maintain the inventory manually. However, so far, Ansible has dynamic inventory only for GCE and not the other components of Google Cloud Platform. If we really want, we can write code for dynamic inventory for other components and use it just like we used `gce.py`.

Deploying the phonebook application

Let us deploy our phonebook app on Google Cloud Platform. We will use the GCE app instance and the MySQL Cloud SQL instance created in previous recipes. We will also use the dynamic inventory that we just used in the previous section.

How to do it...

Using the roles that we wrote in Chapter 1, *Getting Started with Ansible and Cloud Management*, we can create a playbook like this:

```
---
- hosts: tag_app
  roles:
    - phonebook
```

We need to update the `vars/secrets.yml` with the IP of the Cloud SQL instance's, root password that we setup in `recipe12`. We also need to set up an app password which will be used while creating the app user and the database. We execute the playbook and should be able to browse the application on port 8080 of the GCE instance.

We may need to authorize the IP address of the GCE instance in the Cloud SQL so that it can connect to the database.

How it works...

Behind the scenes, this playbook will find the app instance and the database, and it will create an app user in MySQL for the application to use. An important point to note here is that we would need to execute this with a user which has superuser privileges because certain parts of the role installs software packages.

5

Building Infrastructure with Microsoft Azure and Ansible

In this chapter, we will cover the following recipes:

- Preparing Ansible to work with Azure
- Creating an Azure virtual machine
- Managing network interfaces
- Working with public IP addresses
- Using public IP addresses with network interfaces and virtual machines
- Managing an Azure network security group
- Working with Azure Blob storage
- Using a dynamic inventory
- Deploying a sample application

Introduction

Microsoft Azure, formerly known as Windows Azure, is a cloud service from Microsoft that offers **Infrastructure as a Service (IaaS)**, **Platform as a Service (PaaS)**, and **Software as a Service (SaaS)** through its more than 600 services.

Microsoft Azure is present in 36 regions across the world. It was initially released in 2010 as Windows Azure, specifically designed for Windows virtual machines. In 2014, it was renamed Microsoft Azure; along with this, some of Microsoft's Linux services were also renamed. Some of the popular Azure services includes:

- Virtual machines
- SQL databases
- Functions
- Storage
- Azure Active Directory

Preparing Ansible to work with Azure

Ansible interacts with the Azure resource manager's REST APIs to manage infrastructure components using Python SDK provided by Microsoft Azure, which requires credentials of an authorized user to interact with Azure REST APIs.

How to do it...

Ansible packages cloud modules to interact with Azure resource manager. These modules require Azure Python SDK to interact with the Azure resource manager's APIs. We will now prepare our host to run Ansible using Azure cloud modules and put it together so that it can create and manage Azure resources:

1. We should start by installing Azure SDK on a host running Ansible:

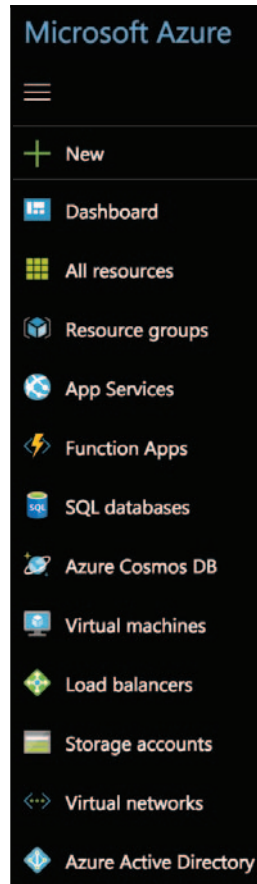
```
$ pip install ansible[azure]
```

2. We need to set up credentials for the Azure resource manager to interact with Azure APIs. Azure provides two ways to authenticate with Azure:
 - Active Directory username and password
 - Service principle credentials

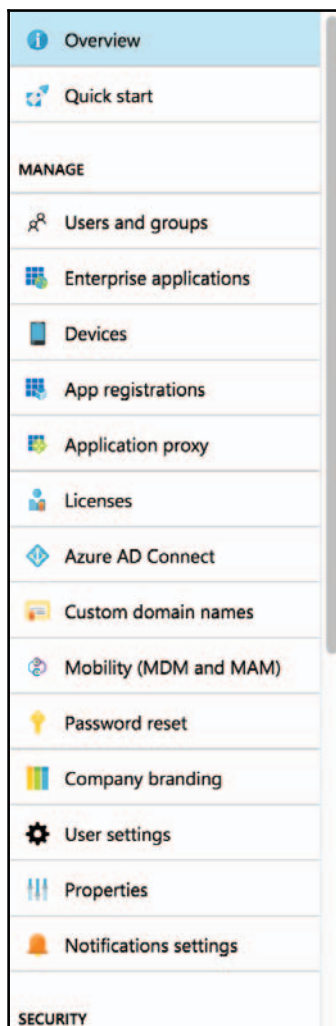
We will use service principle credentials to authenticate with the Azure resource manager. We should note that in order to create service principle credentials to authenticate with Azure, a user must have the required permission in the Azure Active Directory.

Let's ensure that we have the correct permissions to create our application for Ansible:



1. Log into the Azure portal and select **Azure Active Directory**:



2. Select **User settings**, as shown in the following screenshot:



3. Check whether the **App Registrations** setting is set to **Yes**. If it's set to **Yes**, then anyone other than the admin can register an application and we can proceed. If it's set to **No**, either we need to be the admin in order to register an application or we should get it enabled by the admin of the Azure account:

 Save  Discard

Enterprise applications

Users can consent to apps accessing company data on their behalf ⓘ ☒ Yes ☐ No

Users can add gallery apps to their Access Panel ⓘ ☐ Yes ☒ No

App registrations

Users can register applications ⓘ ☒ Yes ☐ No

External users

Guest users permissions are limited ⓘ ☒ Yes ☐ No

Admins and users in the guest inviter role can invite ⓘ ☒ Yes ☐ No

Members can invite ⓘ ☒ Yes ☐ No

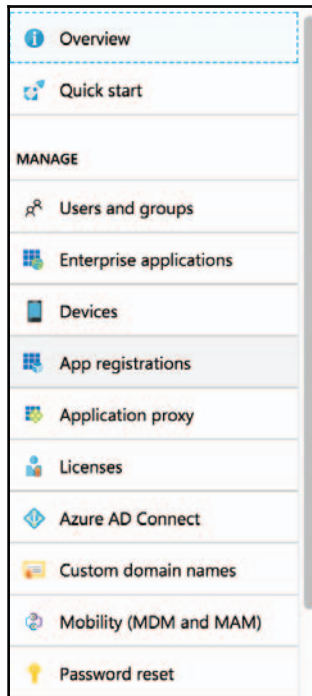
Guests can invite ⓘ ☒ Yes ☐ No

Administration portal

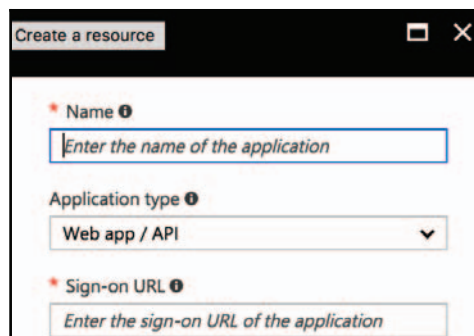
Restrict access to Azure AD administration portal ⓘ ☐ Yes ☒ No

Once we have the required permissions, we will register an application.

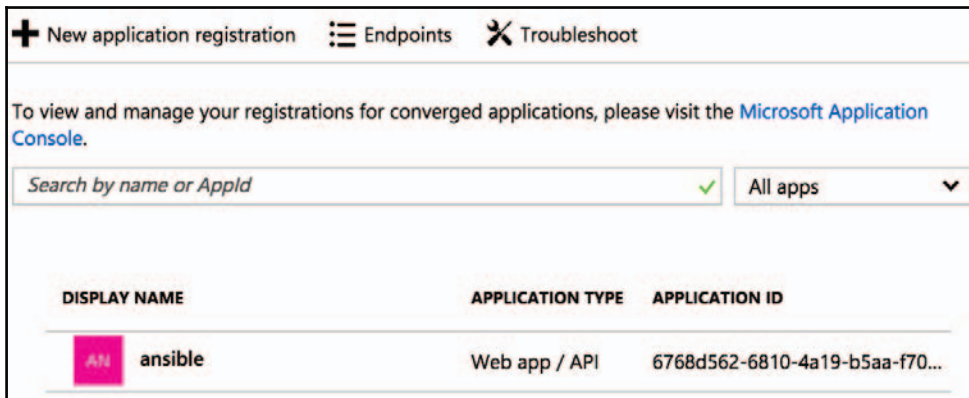
1. Select **Azure Active Directory** and select **App Registrations**:



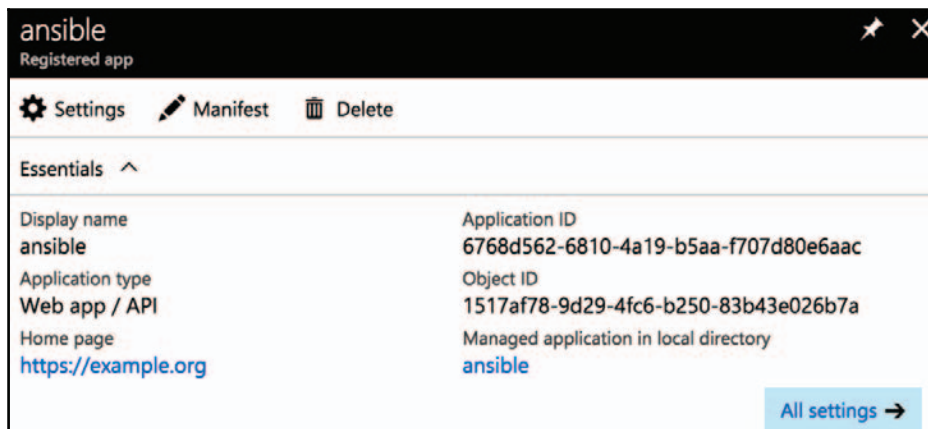
2. Select **New Application Registrations** and we will see a dialog box, as shown in the following screenshot:



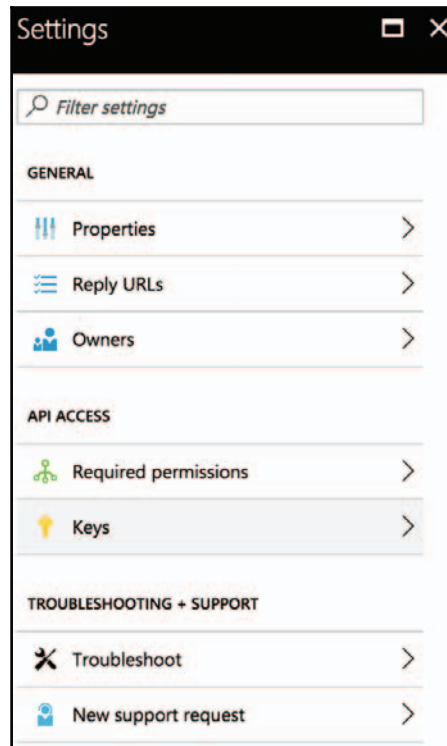
3. Enter a name and URL for the application.
4. We will need the application ID and authorization key for our registered application. Navigate to **App Registrations** and select our application:



5. We can copy the **Application ID**:



6. Let's generate an authorization key. Navigate to **Keys**:

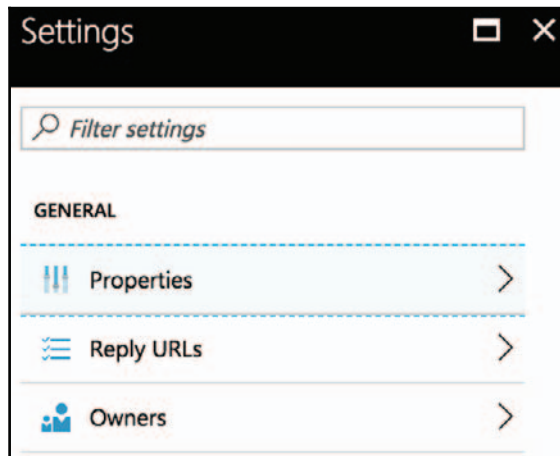


7. Enter the key's name, select the expiration for this key, and note down the auto-generated authentication key:

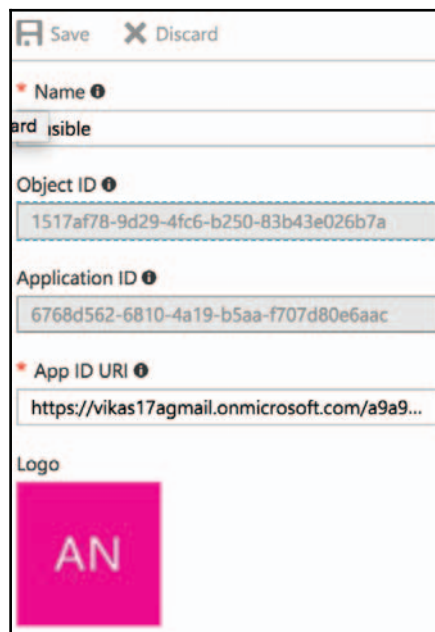


While accessing the Azure resource manager, it also requires the tenant ID. Let's look for the tenant ID.

1. Navigate to **Azure Active Directory** and select **Properties** to get the tenant ID:

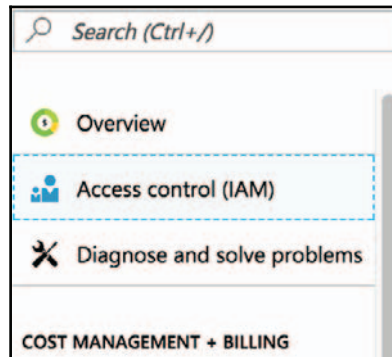


2. The **Application ID** here is our tenant ID:



Now our application is ready, but it needs access to create new resources. We will now assign a role to our application using **Access Control (IAM)**.

1. Select **Subscriptions** and navigate to the current **Azure Subscription | Access Control (IAM)**:



2. Add a new IAM control, using the role from a predefined list and by selecting **Application** under **Resources**. We can select roles or create one depending on what we want to achieve through Ansible automation. To keep this step simple, we are using the **Owner** role.
3. Once we have all the required credentials, we can create a credential file inside `.azure` in the home directory:

```
$ vim ~/.azure/credential
[default]
subscription_id=xxxxxx-xxxxx-xxxxxx-xxxx
client_id=xxxxxx-xxxx-xxxx-xxxxxx
secret=xxxxxxxxxx
tenant=xxxxx-xxxx-xxx-xxx-xxx
```

Creating an Azure virtual machine

In this recipe, we will create a virtual machine using the `azure_rm_virtualmachine` module. Azure provides two types of virtual machines: Windows and Linux. In this recipe, we will be spawning a Linux virtual machine.

Before we jump into creating a Linux VM, we should know the following terms with respect to Azure:

- **Resource groups:** These are logical containers where Azure resources are deployed. We can deploy resources into a specific resource group for a specific use case. For example, we can have resource groups named production for all the production resources and staging for all the resources required for staging.
- **Image:** Azure Marketplace has various images for creating virtual machines. We can select an image of our choice, based on the use case, and create our virtual machine. In this recipe, we will be using an Ubuntu Server image. There are four parameters linked to an image:
 - **Offer** defines the distribution type. For example, Ubuntu, Debian, or CoreOS.
 - **Publisher** refers to the organization that created the image. For example, canonical, credativ, or CoreOS.
 - **SKU** defines the instance of the image offered. For example, 16.04-LTS for Ubuntu Server.
 - **Version** defines the version of an image SKU. When defining an image, we can set the version to **Latest** to select the latest SKU of an image.
- **Storage account:** The Azure Storage Account is a Microsoft-managed cloud service which provides scalable, highly available, and redundant storage. Azure Storage offers three kinds of storage:
 - **Blob storage:** This stores our files in the same way that they are stored on local computers. For example, images, PDFs, and so on. The storage can consist of a **virtual hard disk (VHD)** format, large log files, and so on. This kind of storage is used for creating disks attached to a virtual machine.
 - **File storage:** This is the network file share, which can be accessed through standard **Server Message Block (SMB)** protocol.
 - **Queue storage:** This is a messaging storage system. A single queue can store millions of messages, and each message can be up to 64 KB in size.
- **Location:** This is a region where we can deploy our resources in Azure Cloud. All of the recipes in this chapter will be deploying resources in the westus region (California, USA). We will define this location as `azure_region` in `group_vars` for the playbook:

```
azure_region: westus
```

How to do it...

1. Create a resource group for deploying resources:

```
- name: Create resource group
  azure_rm_resourcegroup:
    name: example
    location: "{{azure_region}}"
    tags:
      env: testing
  tags:
    - recipe4
```

2. Create a storage account for our VM disk:

```
- name: Create a storage account
  azure_rm_storageaccount:
    resource_group: example
    name: examplestorage01
    type: Standard_RAGRS
    location: "{{azure_region}}"
    tags:
      - env: testing
  tags:
    - recipe4
```

3. Let's create our first VM in Azure Cloud:

```
- name: Create VM
  azure_rm_virtualmachine:
    resource_group: example
    name: first_vm
    location: "{{azure_region}}"
    vm_size: Standard_D4
    storage_account: examplestorage01
    admin_username: cookbook
    ssh_public_keys:
      - path: /home/admin/.ssh/authorized_keys
        key_data: "ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQDQ8ddP3LGD8r586Nj19lqScZvakv4DvGPsK9PN
Cw+MWaLZsSovUECLm1v3IxfBhbGUrbQMFAbfff0Piie9+6aas5vSfaqn1LMhEyVNjJka
Faztg/FiYbhcSzb4zc7hrKyZriUyou2gj68o9113g38wh0tK6TSjfQ+DrN2HiV8bo4j
LYmGnh+A3O6HMWR1ceCclN5c3g4RRjrLzSC9YolufMDLzs4CWxjEDLufYwsPqafOrvc
XU1LeAzfjYrG8Re82sH6uE8Zw1WHRDk9hhRZU8s5jFCtepLeHL0jgftMXHGEP7F/cFX
Zb9KzdO1sqIie7OMfQ44hAPAcA1KexEPt6gb1"
    image:
```

```
offer: UbuntuServer
publisher: Ubuntu
sku: '16.04-LTS'
version: latest
tags:
  - recipe4
```

How it works...

In *step 1*, we created a resource group for deploying resources in our defined location. We will be using this resource group name in subsequent tasks to deploy all of the resources in this resource group.

In *step 2*, we created a storage account, which will be required for the OS disk of our virtual machine. Azure offers multiple storage types depending upon the use case and availability- **Locally Redundant Storage (LRS)**, **Geo-Redundant Storage (GRS)**, **Zone-Redundant Storage (ZRS)**, or **Read Access Geo-Redundant Storage (RAGRS)**. We are using **Standard_RAGRS**.

Finally, in *step 3*, we created our first virtual machine in Azure Cloud. This task will take care of setting up an admin user and deploying its public keys at the required places. We have set the `admin_username` as `cookbook`. Once our virtual machine is ready, we can connect to our virtual machine using the SSH protocol and the private key associated with the public key we used in the task.

Managing network interfaces

An Azure virtual machine can be attached to multiple network interface cards. With a network interface card, the virtual machine can access the internet and communicate with other resources both inside and outside Azure Cloud. In the recipe, *Creating an Azure virtual machine*, while creating a virtual machine, it creates a default NIC card for the VM with default configurations. In this recipe, we will see how to create, delete, and manage a NIC with custom settings.

Before we move ahead and create a NIC, we should be aware of the following term:

- **Virtual network:** This is a logical separation of the network within the Azure Cloud. A virtual network can be assigned a CIDR block. Any virtual network has the following attributes associated with it:
 - Name
 - Address Space
 - Subnet

How to do it...

1. Create a virtual network:

```
- name: Create Virtual Network
  azure_rm_virtualnetwork:
    name: vnet01
    resource_group: example
    address_prefixes_cidr:
      - "10.2.0.0/16"
      - "172.1.0.0/16"
    tags:
      env: testing
  state: present
  tags:
    - recipe2
```

2. Create a subnet:

```
- name: Create subnet
  azure_rm_subnet:
    name: subnet01
    virtual_network_name: my_first_subnet
    resource_group: example
    address_prefix_cidr: "10.2.0.0/24"
    state: present
  tags:
    - recipe2
```

3. Create a Network Interface Card:

```
- name: Create network interface card
  azure_rm_networkinterface:
    name: nic01
    resource_group: example
    virtual_network_name: vnet01
    subnet_name: subnet01
    public_ip: no
    state: present
  register: network_interface
  tags:
    - recipe2
```

4. Access the private IP address:

```
- name: Show private ip
  debug:
    msg:
      "{{network_interface.ip_configuration.private_ip_address}}"
  tags:
    - recipe2
```

How it works...

In *step 1*, we created a virtual network with the name `vnet01` within the same resource group we used in the first recipe, *Creating an Azure virtual machine*. Since we are using a resource group, the resources can pick the default location of the resource group. We have defined the CIDR network addresses as `10.2.0.0/24` and `172.1.0.0/16`. We can also define multiple network addresses using YAML syntax.

We should note here that if we want to delete an existing virtual network, we can set the state to `absent`. By default, most of the Ansible modules have their state set to `present`.

In *step 2*, we created a subnet using the `azure_rm_subnet` module inside the virtual network `vnet01`.

In *step 3*, we created a network interface card using `azure_rm_networkinterface` and named it `nic01`. We specified the `public_ip` as `no`, which will ensure that Azure will not allocate any public IP address to the NIC, but also that NIC will be associated with a private IP from the subnet address space.

After creating the NIC, we register the output of this module in the `network_interface` variable to display its properties in *step 4*.

In *step 4*, we use the `debug` module to print the private IP address allocated to the network interface using the variable registered in *step 3*.

Working with public IP addresses

We just created a NIC, but without the public IP, we can't reach that network interface and the virtual machine associated with it from the public internet. In this recipe, we will create a public IP address and associate it with the network interface.

Azure allocates the public IP address using one of two methods, static or dynamic. An IP address allocated with the static method will not change, irrespective of the power cycle of the virtual machine; whereas, an IP address allocated with the dynamic method is subject to change. In this recipe, we will create a public IP address allocated with the `Static` method.

How to do it...

1. Create a public IP:

```
- name: Create Public IP address
  azure_rm_publicipaddress:
    resource_group: example
    name: pip01
    allocation_method: Static
    domain_name: test
    state: present
    register: publicip
  tag:
    - recipe3
```

2. Display a public IP:

```
- name: Show Public IP address
  debug:
    msg: "{{ publicip.ip_address }}"
  tag:
    - recipe3
```

How it works...

In *step 1*, we created a public IP address using the `azure_rm_publicipaddress` module, named it `pip01`, and registered the result with the `publicip` variable.

In *step 2*, we used the `debug` module to print the public IP address stored in the `publicip` variable.

Using public IP addresses with network interfaces and virtual machines

In this recipe, we will create a virtual machine and network interface using the public IP we created in the recipe, *Working with public IP addresses*. After creating a virtual machine with the public network interface, we will log into it using SSH. We have already seen how to create a virtual machine in the recipe, *Creating an Azure virtual machine*. We will also learn how this Ansible task differs from what we learned in the first recipe, *Creating an Azure virtual machine*.

How do it...

1. Create a NIC with the existing public IP address:

```
- name: Create network interface card using existing public ip
  address
  azure_rm_networkinterface:
    name: nic02
    resource_group: example
    virtual_network_name: vnet01
    subnet_name: subnet01
    public_ip_address_name: pip01
    state: present
  register: network_interface02
  tags:
    - recipe4
```

2. Create a virtual machine with the existing network interface:

```
- name: Create VM using existing virtual network interface card
  azure_rm_virtualmachine:
    resource_group: example
    name: first_vm
    location: "{{azure_region}}"
    vm_size: Standard_D4
    storage_account: examplestorage01
    admin_username: cookbook
    ssh_public_keys:
      - path: /home/admin/.ssh/authorized_keys
        key_data: "ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQDQg8ddP3LGDr586Njl9lqScZvakv4DvGPsK9PN
Cw+MWaLZsSovUECLm1v3IxfBhbGUrBQMFAbfff0Piie9+6aas5vSFAqn1LMhEyVNjJka
Faztg/FiYbhcSzb4zc7hrKyZriUyou2gj68o9113g38wh0tK6TSjfQ+DrN2HiV8bo4j
LYmGnh+A3O6HMWR1ceCclN5c3g4RRjrLzSC9YolufMDLzs4CWxjEDLufYwsPqafOrvc
XU1LeAzfjYrG8Re82sH6uE8ZW1WHRDk9hhRZU8s5jFCTepLeHL0jgftMXHGEP7F/cFX
Zb9KzdO1sqIie7OMfQ44hAPAcA1KexEPt6gb1"
    image:
      offer: UbuntuServer
      publisher: Ubuntu
      sku: '16.04-LTS'
      version: latest
    network_interfaces: nic02
  tags:
    - recipe4
```

3. Log into the VM using the public IP from the recipe, *Working with public IP addresses*:

```
$ ssh -i ~/.ssh/cookbook.pem cookbook@13.33.23.24
```

How it works...

In the first two steps, we created a NIC with an existing public IP, created in the recipe *Working with public IP addresses*, and a virtual machine using that NIC.

In *step 3*, we logged into the virtual machine created with the public NIC.

We should note here that when we create a network interface without specifying a network security group, it uses the default network security group. In the upcoming recipe, we will see how to create a network security group.

Managing an Azure network security group

In Azure, a network security group is an **access control list (ACL)**, which allows and denies network traffic to subnets or an individual NIC. In this recipe, we will create a network security group with some basic rules for allowing web (HTTP and HTTPS) and SSH (port 22) traffic and denying the rest of the traffic. Since a network security group is the property of the network and not the virtual machine, we can use subnets to group our virtual machines and keep them in the same network security group for the same ACL.

How to do it...

1. Create a network security group:

```
- name: Create network security group
  azure_rm_securitygroup:
    resource_group: example
    name: msg01
    purge_rules: yes
    rules:
      - name: 'AllowSSH'
        protocol: TCP
        source_address_prefix: *
        destination_port_range: 22
        access: Allow
        priority: 100
        direction: Inbound
      - name: 'AllowHTTP'
        protocol: TCP
        source_address_prefix: *
        destination_port_range: 80
        priority: 101
        direction: Inbound
      - name: 'AllowHTTPS'
        protocol: TCP
        source_address_prefix: *
        destination_port_range: 443
        priority: 102
        direction: Inbound
      - name: 'DenyAll'
        protocol: TCP
        source_address_prefix: *
        destination_port_range: *
        priority: 103
        direction: Inbound
```

```
tags:
  - recipe5
```

2. Attach the subnet to the security group:

```
- name: Create subnet
  azure_rm_subnet:
    name: subnet01
    virtual_network_name: my_first_subnet
    resource_group: example
    address_prefix_cidr: "10.2.0.0/24"
    state: present
    security_group_name: msg01
  tags:
    - recipe5
```

3. Attach the NIC card to the security group:

```
- name: Create network interface card using existing public ip
  address and security group
  azure_rm_networkinterface:
    name: nic02
    resource_group: example
    virtual_network_name: vnet01
    subnet_name: subnet01
    public_ip_address_name: pip01
    security_group_name: msg01
    state: present
  register: network_interface02
  tags:
    - recipe5
```

How it works...

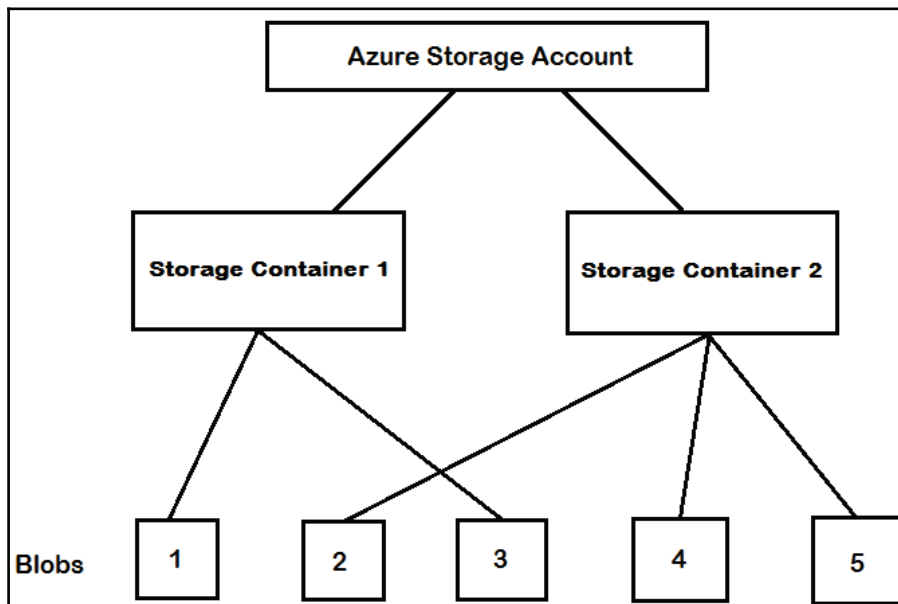
We used `azure_rm_securitygroup` to create a security group and defined rules with unique priorities by which rules are applied to the Azure network. In the later steps, we associated the network security group with a subnet and also with a NIC.

We should note here that the parameter we used for this module (`purge_rules` as `true`) will override the existing security group. We should avoid using this parameter for an existing security group that is not managed by Ansible.

Working with Azure Blob storage

The Azure Storage Account offers Blob storage, which operates in the same way as files stored on our local systems, such as pictures, videos, PDFs, and so on. In this recipe, we will learn how to use the Azure Storage Account for storing blobs.

Before we jump into creating blob objects, we should know some more about the blob service concept. Look at the following diagram:



- The **Azure Storage Account** is an access point through which Azure storage can be accessed.
- The **Storage Container** is a set of specific storage blobs. Every blob needs to be part of a container.
- The **Blobs** is a file that could be anything, such as a PDF, image, video, and so on.

How to do it...

1. Create a storage container:

```
- name: Create storage container
  azure_rm_storageblob:
    resource_group: example
    storage_account_name: examplestorage01
    container: cookbook
    state: present
```

2. Add a blob to the container:

```
- name: Upload a file to existing container
  azure_rm_storageblob:
    resource_group: example
    storage_account_name: examplestorage01
    container: cookbook
    blob: myfile.png
    src: /tmp/myfile.png
    public_access: blob
    content_type: 'application/image'
```

3. Download the uploaded file:

```
- name: Download blob object
  azure_rm_storageblob:
    resource_group: example
    storage_account_name: examplestorage01
    container: cookbook
    blob: myfile.png
    dest: /tmp/download_file.png
```

How it works...

In *step 1*, we created an Azure Storage Container and named it `cookbook`.

In *step 2*, we uploaded an image file with the PNG format, located at `/tmp/myfile.png` on the host running the Ansible task, to the `cookbook` container. We also set the `public_access` parameter as a `blob`. We granted this blob the public read permission.

In *step 3*, we downloaded the same file we just uploaded to the `/tmp/download_file.png` location on the Ansible host.

Using a dynamic inventory

In this recipe, we will use dynamic inventory to target hosts and build an Ansible inventory using the Azure Resource Manager API. Dynamic inventory enables us to leverage its dynamic nature by not hardcoding the host's addresses in a static inventory file.

Ansible packages Azure's inventory script within an Ansible repository. We can find the inventory script at

https://github.com/ansible/ansible/blob/devel/contrib/inventory/azure_rm.py. The inventory script also requires the same credentials located at `~/.azure/credentials`, as required for other Azure Cloud modules.

Ansible can target the hosts present in a group, and by default, the Azure inventory creates the following groups:

- **azure (all hosts):** This will contain all hosts, and can be accessed using `hosts: azure`.
- **location name:** This will group hosts by their location—for example, `westus`, `eastus`, and so on. This can be accessed using `hosts: westus`.
- **name of resource group:** This will group hosts deployed in a resource group. It can be accessed using `hosts: example`.
- **security group name:** This will group hosts deployed with the same security groups. For example, `mysg01`. It can be accessed using `hosts: mysg01`.
- **tag key:** This will group hosts by their respective tag key—for example, `env`. It can be accessed by `hosts: env`.
- **tag key_value:** This will group hosts by their tag's key and value. For example, a host may have the tag key `env` and the value `testing`. It can be accessed using `hosts: tag_env_testing`.

We can define a few other parameters during configuration to restrict hosts in the inventory. We could define specific groups and filters, such as resource groups, using the following inventory configuration file:

```
[azure]

# Controls resource groups to include in inventory.
#resource_groups=

# Control which tags are included. Set tags to a comma separated list of
keys or key:value pairs
#tags=
```



```
# Control which locations are included. Set locations to a comma separated
list of locations.
#locations=

# Include powerstate. If you don't need powerstate information, turning it
off improves runtime performance.
include_powerstate=yes

# Control grouping with the following boolean flags.
group_by_resource_group=yes
group_by_location=yes
group_by_security_group=yes
group_by_tag=yes
```

How to do it...

1. Use Ansible dynamic inventory with the following ad hoc command:

```
#Target all hosts in azure group
$ ansible -i azure_rm.py azure -m ping

#Target all hosts in example resource group
$ ansible -i azure_rm.py example -m ping
```

2. Use `ansible-playbook` with dynamic inventory:

```
$ ansible-playbook -i azure_rm.py playbook.yml

# Content of playbook.yml
$ cat playbook.yml
---
- host: tag_env_preprod
  tasks:
    - name: Test Dynamic Inventory
      shell: /bin/uname -a
```

3. Use Azure tags to target hosts in Ansible:

```
$ ansible tag_env_preprod -m ping
```

Deploying a sample application

We will now deploy a simple phonebook application in Azure Cloud. In this application, we will create a resource group (phonebook), a virtual network (phonebook-vnet01), a subnet (phonebook-net01), a security group (phonebook) allowing HTTP and SSH traffic, a network interface with a public IP, and finally a virtual machine.

How to do it...

Using recipes from this chapter, we can create and save our playbook as `phonebook.yml`:

```
#Playbook for deploying phonebook application in Azure Cloud
---
- hosts: tag_Name_first_vm
  gather_facts: no
  roles:
    - phonebook
```

We should note that the hosts in this playbook are set as `tag_Name_first_vm` (Ansible dynamic inventory), which will create an inventory for our phonebook application at runtime. We can execute our playbook using the following command:

```
$ ansible-playbook -i azure-rm.py --become phonebook.yml
```

We used a command-line inventory argument for our playbook, since we are creating our resources in the playbook itself. After successful completion of this playbook, we will be able to access our application on port 8000 and the host IP of the virtual machine created in this playbook, which will be displayed in the last task using the `debug` module.

6

Working with DigitalOcean and Ansible

In this chapter, we will cover the following recipes:

- Preparing to work with DigitalOcean
- Adding SSH keys to a DigitalOcean account
- Creating Droplets
- Managing Block Storage
- Attaching a Floating IP
- Using a Load Balancer
- Adding an A DNS Record
- Using dynamic inventory
- Deploying a sample application

Introduction

DigitalOcean is one of the fastest-growing cloud providers. A major reason for the tremendous growth of DigitalOcean is its simplicity and appeal to the developer community. Creating resources on DigitalOcean needs only the smallest amount of preparation, and usually takes less than five minutes. Another reason why DigitalOcean is very popular is its cost, which is easily predictable, starting at US\$5. It consists of 12 datacenters spread across 7 countries, in 3 continents. To maintain simplicity, DigitalOcean sometimes compromises on features. For example, at the time of writing, there are no managed databases. Also, DigitalOcean does not support Windows.

However, all the building blocks for a typical application are available:

- Droplets (virtual machines)
- Volumes (attachable block storage)
- Load balancers
- DNS
- Spaces (object storage)

Along with various DigitalOcean services, we are also going to handle the problem of non-idempotency in one of the Ansible modules. We are also going to use dynamic inventory and set up our SQLite-based phonebook app.

Preparing to work with DigitalOcean

DigitalOcean has a very minimal setup. Some of the Ansible modules requires the `dopy` Python module.

How to do it...

The `dopy` Python module can be used for various DigitalOcean operations, so we have to install one dependency:

```
$ pip install dopy
```

Just like other Ansible modules for cloud providers, DigitalOcean modules need to interact with DigitalOcean APIs. Along with `dopy`, we need an authentication token from DigitalOcean to interact with its API:

1. To generate a token, we need to visit the API settings URL in the DigitalOcean dashboard (<https://cloud.digitalocean.com/settings/api/tokens>).
2. Click on the **Generate New Token** button. A form will ask us for the name of the token and scope.
3. Since we are going to build droplets and other resources, we are required to select the optional **Write** scope as well.

- Once we click on the **Generate Token** button on this form, a token will be generated. We need to save this token securely in our `secrets.yml`. If lost, the token cannot be recovered. However, we can always generate a new token:

- Now that we have the token, let's create `roles/digitalocean/vars/secrets.yml`:

```
---
DO_OAUTH_TOKEN:
"5fcd456fb9f5hg5f29a17df55678j4fd4ecbdfghjkgf4c215fghjbbaa39bah7tc47m
"
```

- Now, we can encrypt this:

```
$ ansible-vault encrypt roles/digitalcoean/vars/secrets.yml
```

We are ready to create resources in DigitalOcean now.

Adding SSH keys to a DigitalOcean account

There are two ways to log into a DigitalOcean droplet. If public key information is supplied during the creation of the droplet, then it will be added to the root user of the droplet. Otherwise, a one-time password is emailed to the registered user ID. Because of security implications, we strongly recommend using public keys and not passwords for logging in.

How to do it...

1. For adding an SSH public key to our account, we will use the `digital_ocean_sshkey` module:

```
- name: Add SSH key to DO account
  digital_ocean_sshkey:
    name: "cookbook-key"
    ssh_pub_key: "{{ ssh_public_key }}"
    oauth_token: "{{ DO_OAUTH_TOKEN }}"
```

We have supplied a name along with the key. We need to add the token for the authentication as well.

2. We will store the public key in the `roles/digitalocean/vars/main.yml` as a variable:

```
--
ssh_public_key: ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQCbG83WYIxUfXWJ4bQiYfZYHceDwMJxnGfJqgY
tHL/DAteGVY+Nm8MX3CRZYisfskt0m9CQ6y/Ux1OITTz+O11fgxLJcroZmKJbWW0K39
gfHvFqR7FIe0zuJaxqUQUuyc0i6RCBRiZPiQQOPes2yDtfHgDWx3q9knS3ZXIAXcGLZ
rgfC1XnIK8CLAnZDved9Rue2bhscn09Mleh9g/CTtehMDAzD4NeSv9eETlHYkYSpJg8
gFA3BFICpBxTqWSjflmMQGSmiudFOhRjHIXL1Tvh+pnjSoL/jrLcP3RtMVuG0ZU0qko
Ats1qpTwmyAJUz9Ts2EeyDJ0tXsFAiOfbmuMd aditya@devopsnexus.com
```

Creating Droplets

We can create droplets using the `digital_ocean` module. One thing to note about this module is that it is idempotent in a different way. The module identifies an instance by an ID. If we know the `Droplet ID`, we have to put it in the task and then the task will not create another droplet. However, without `Droplet ID`, the multiple execution of this task will keep on creating more droplets. We have to figure out a way to create a droplet only if one does not already exist.

Holding that thought in mind, let's take a small detour and have a look at a nice utility called `doctl`. DigitalOcean provides the `doctl` command-line tool to create and manage resources. For authentication, we can create a `config` file or supply the token along with the command. Since we have the token saved as a secret, it would be easier for us to pass it along with the command using Ansible tasks.

How to do it...

1. Let's download the `doctl` tool from GitHub's release page (<https://github.com/digitalocean/doctl/releases/tag/v1.7.1>) and extract the archive in the binary path, such as `/usr/bin`:

```
$ wget
https://github.com/digitalocean/doctl/releases/download/v1.7.1/doctl-1.7.1-linux-amd64.tar.gz
$ tar -xzf doctl-1.7.1-linux-amd64.tar.gz
$ mv doctl /usr/bin/
$ chmod +x /usr/bin/doctl
```

2. Using `doctl`, we can query DigitalOcean as to whether a droplet exists. We will use this to create idempotency for our task. We should start by listing the instances:

```
- name: check for the droplet
  shell: doctl compute droplet list --output json --access-token
"{{ DO_OAUTH_TOKEN }}"
  register: droplets
```

3. Executing the preceding task will get us a list of existing droplets in JSON format. We save this list in a variable called **droplets**. Ansible provides us with the capability to parse JSON and extract values. We are going to search for the ID of a droplet with the name `app`. We can start with the stored list of droplets in the variable `droplets`. That data is stored as a string in `droplets.stdout`. We can get a JSON out of it since we used `--output json` in the previous task. We have to pipe this JSON to a `json_query` filter, which filters out IDs and names. Finally, we have to pipe this further to get the ID with the name `app`. We will store this in a variable called `app`. If the instance exists, then this will have a valid ID; otherwise, this would be empty:

```
- name: find the id of droplet
  debug:
    msg: "{{ droplets.stdout | from_json | json_query('[].{id: id, name: name} | [?name=='app'].id \')}}"
  register: app
```

Now that we have the ID of the droplet, let's execute the task to boot the droplet:

```
- name: Creating app droplet
  digital_ocean:
    id: "{{ app.msg[0] | default('0') }}"
```

```
state: present
command: droplet
name: app
size_id: 512mb
region_id: blr1
image_id: centos-7-x64
wait_timeout: 500
ssh_key_ids: "{{ result.data.ssh_key.fingerprint }}"
api_token: "{{ DO_OAUTH_TOKEN }}"
register: app_droplet
```

For the preceding task, we have specified a default value of 0 to handle the case where the `app` variable registered in the previous task returns an empty value, signifying the non-existence of the droplet. In such a case, a droplet with the name `app` would be created. We have also supplied values for the command to use, the name, and specific details of the droplet, such as region, image, and size. For `ssh_key_ids`, we have used the value from the variable that we registered in the first recipe, *Adding SSH keys to a DigitalOcean account*. This is just the md5 fingerprint of the key, which can be generated manually as well. If we want to go the manual route, then we can actually skip the tasks that we executed to get the ID of the droplet and just put this ID in this task itself.

Managing Block Storage

Often, the default disk available on the droplet might not be enough. For such scenarios, DigitalOcean provides a block storage service that can be attached to a droplet as an additional disk. This storage is highly available and easily extendable. Using Ansible, we can create and attach the block storage to a droplet.

How to do it...

1. Let's start by creating a 10 GB block. We have to specify the `command`, `region`, `block_size`, and `volume_name`. The authentication token is also required along with these details. Note that this task will only create the block but will not attach it to any of the droplets:

```
- digital_ocean_block_storage:
  state: present
  command: create
  region: blr1
```



```
block_size: 10
volume_name: cookbook1
api_token: "{{ DO_OAUTH_TOKEN }}"
```

2. Once we have the block, let's attach it to the app droplet that we created in the recipe *Creating droplets*:

```
- digital_ocean_block_storage:
  state: present
  command: attach
  volume_name: cookbook1
  region: blr1
  droplet_id: "{{ app_droplet.droplet.id }}"
  api_token: "{{ DO_OAUTH_TOKEN }}"
```

Here, we attached a volume named `cookbook1` to the droplet with the ID obtained from the registered variable from the previous recipe. We also need to pass the authentication token as well.

Attaching a Floating IP

Floating IP addresses are movable IPs that can be moved around from droplet to droplet as required. This can be helpful to route traffic away to a healthy droplet in case a droplet turns bad.

How to do it...

1. Let's create a floating IP and attach it to the droplet:

```
- name: attach a floating IP
digital_ocean_floating_ip:
  state: present
  droplet_id: "{{ app_droplet.droplet.id }}"
  api_token: "{{ DO_OAUTH_TOKEN }}"
```

This task takes `droplet_id` as a parameter and attaches a floating IP to it.

Using a Load Balancer

A load balancer can help in improving uptime and distributing requests among droplets. DigitalOcean provides a load balancer with a bunch of features:

- Load balancing algorithms
- Health checks
- SSL redirection
- Port forwarding

The problem here is that there is no Ansible module for load balancers.

How to do it...

1. As with previous chapters, we will use the `shell` module and the `doctl` command-line tool to write an Ansible task to create a load balancer:

```
- name: create a load balancer
  shell: doctl compute load-balancer create --name lb1 --region
blr1 --droplet-ids {{ app_droplet.droplet.id }} --forwarding-rules
{{ forwarding_rules }} --health-check {{ health_check }} -t {{
DO_OAUTH_TOKEN }}
```

A load balancer can support a lot of options, but we have used the bare minimum to make it easier to understand. The `name` and `region` parameters specify the name and region of the load balancer. The name has to be unique and the region must be the same as the droplets. The `droplet-ids` parameter can take a comma-separated list of droplet IDs. We can define forwarding rules and health check rules as well.

2. Let's first create two variables in `roles/digitalocean/vars/main.yml` so that we can understand them better:

```
---
forwarding_rules:
entry_protocol:http,entry_port:8080,target_protocol:http,target_por
t:8080
health_check:
protocol:http,port:8080,path:/,check_interval_seconds:10,response_t
imeout_seconds:5,healthy_threshold:3,unhealthy_threshold:3
```

The `forwarding_rules` defines an `entry_protocol`. The load balancer is expecting to receive requests in this protocol, which is HTTP in our case, but it could be anything else, such as TCP. The `entry_port` is the port on the load balancer that will be exposed to our users. Similarly, the `target_protocol` and the `target_port` are the protocol and the port for the droplet. The health check rule defines parameters for executing a periodic health check on every droplet. Our variable signifies that we want to issue an HTTP request on port 8080 and path /. This request would execute every 10 seconds on each droplet, with a timeout of 5 seconds. Three consecutive successes or failures are required to add or remove the droplet from the load balancer. This health check helps us ensure that the requests of our users won't go to a droplet that is down for any reason.

Adding an A DNS record

DigitalOcean provides a DNS service where we can add our domains and various related records. The Ansible module for the DNS is available; however, it supports only basic operations; that is, creating an A record.

How to do it...

We can use `digital_ocean_domain` module to create an A record.

```
- name: associate domain and IP
digital_ocean_domain:
  state: present
  name: ansiblecloudbook.com
  ip: "{{ app_droplet.droplet.ip_address }}"
  api_token: "{{ DO_OAUTH_TOKEN }}"
```

For this task, we have supplied a domain name and IP address for creating the record. This is as far as this module goes. For advanced operations, we can use the `doctl` command-line tool or the web dashboard.

Using dynamic inventory

As we have established in previous chapters, managing a text-based inventory can be a pain. There are chances that a manually managed inventory can consist of stale data that can impact our applications. Ansible provides dynamic inventory to address this.

Ansible's GitHub repository has scripts for various infrastructure providers, including a script for DigitalOcean (https://github.com/ansible/ansible/blob/devel/contrib/inventory/digital_ocean.py).

How to do it...

1. We need to download this script and configure it to use our credentials:

```
$ curl
https://github.com/ansible/ansible/raw/devel/contrib/inventory/digital_ocean.py > digital_ocean.py
$ chmod +x digital_ocean.py
$ curl
https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/digital_ocean.ini > digital_ocean.ini
```

2. We need to append the following line in the `digital_ocean.ini` for authentication:

```
api_token =
5fcd456fb9f5hg5f29a17df55678j4fd4ecbdfghjkgf4c215fghjbbaa39bah7tc47m
```

3. Let's test our inventory script by listing the resources:

```
$ ./digital_ocean.py --list
{
  "image_7.4_x64": {
    "hosts": [
      "139.59.5.202"
    ],
    "vars": {
      }
    },
    "all": {
      "hosts": [
        "139.59.5.202"
      ],
      "vars": {
        }
    },
    "distribution": "CentOS",
    "type": "snapshot",
```

```
        "public":true,
        "size_gigabytes":0.4
    }
}
}
}
```

Note that we have truncated the output of the command to improve readability.

4. The dynamic inventory script lets us execute Ansible tasks and modules just by knowing the name of the instance:

```
$ ansible -i digital_ocean.py app -u root -m ping
139.59.5.202 | SUCCESS => {
  "changed": false,
  "failed": false,
  "ping": "pong"
}
```

5. Along with the name, we can also call the inventory based on operating system, the region, and so on.

Using a dynamic inventory in a playbook is easy as well:

```
---
- hosts: app
  roles:
    - myrole
```

Deploying a sample application

Let's deploy our phonebook application. Since DigitalOcean does not offer a managed database service, we will deploy the SQLite variant of our application. Since we are going to use dynamic inventory, we will create the playbook as follows:

```
---
- hosts: app
  roles:
    - phonebook
```

How to do it...

To execute the playbook, along with dynamic inventory, we should execute the following command:

```
$ ansible-playbook -i digital_ocean.py --become phonebook.yml
```

This command installs the phonebook on our droplet. We can leverage the load balancer and provide the IP address of the load balancer instead of the droplet. Keep in mind that we are using SQLite here, not MySQL; SQLite is not accessible over the network.

7

Running Containers with Docker and Ansible

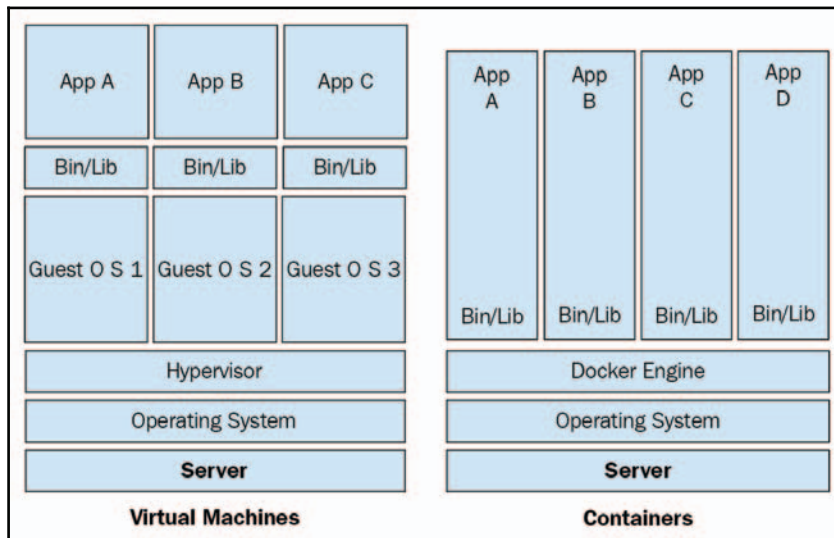
In this chapter, we will cover the following recipes:

- Preparing Ansible to work with Docker
- Running a container
- Downloading Docker images
- Mounting volumes in containers
- Setting up Docker Registry
- Logging into Docker Registry
- Using Docker Compose to manage services
- Scaling a Compose-based service
- Deploying a sample application

Introduction

Containers are lightweight, standalone software packages that contain all the dependencies it requires to run, such as the code itself, libraries, configurations, and so on. It allows us to run an application along with its dependencies in isolation. Containers enable the deployment of an application quickly and seamlessly, and also gives us more control over system resources. Unlike a virtual machine, which has its own kernel, containers don't need their own kernel for different containers. A container will share its kernel with its host.

Docker is one of the most popular containerization platforms, and was developed by company called Docker, Inc. Docker is a tool that can package code and its dependencies into a container, which in turn can run on multiple platforms.



As a platform, Docker has evolved from a simple containerization tool. Now it consists of supporting tools for running multi-container applications, clustering, logging, and many more features that can help throughout an application's life cycle.

In this chapter, we will be using Ansible to deploy, manage, and scale Docker containers. Ansible modules allow us to use a local Docker engine using a Unix socket, and they also support the use of remote Docker engines that use a TCP connection over network. In this chapter, we will be using the local Docker engine. We are also going to look at Docker Compose, which is a great tool for running multi-container applications.

Preparing Ansible to work with Docker

We need to make sure that Docker engine is up and running. Using the Ansible module with Docker requires installation of the following Python packages:

- docker-py
- docker-compose

How to do it...

1. Let's install the pip package `docker-py` in a version greater than 1.7.0. Ansible will use this Python library to interact with the Docker API:

```
$ pip install 'docker-py>=1.7.0'
```

2. Let's install the `docker-compose` pip package in a version greater than 1.7.0. Ansible will use this to interact with Docker compose:

```
$ pip install 'docker-compose>=1.7.0'
```

3. The Ansible module connects with the Docker API, as we mentioned earlier. Using parameters passed with tasks, we will be using a local Docker engine to create and manage Docker containers. It connects to a `docker` parameter using the parameter called `docker_host`. We can also set the environment variable to `DOCKER_HOST`.

We should note that the default host used by the Ansible module is set as `unix:///var/run/docker.sock`, which allows the Docker module to interact with local Docker engines running on the same host.

Running a container

In this recipe, we will be deploying a basic container using Ansible module's `docker_container`. Before we jump into creating a task, however, we should look at the following concepts, which are required to understand Docker containers:

- **Docker images:** A Docker image is a read-only template that consists of all the dependencies necessary to run a piece of code. A container is actually a runtime instance of a Docker image.
- **Docker Hub:** Docker Hub is the official repository of Docker images. Docker images created by organizations and the wider community are available on Docker Hub for general public use.

We will be using an Ubuntu image to create a Docker container. The `docker_container` module will download the image from Docker Hub and run the container.

How to do it...

1. Let's define the task for creating a Docker container:

```
- name: Create and start a container using Ubuntu Image
  docker_container:
    name: myfirstcontainer
    state: started
    image: ubuntu:14.04
    command: sleep infinity
  tags:
    - recipe2
```

2. Now, verify that the container is up and running:

```
$ docker ps -a
```

3. Let's now run a command in our Docker container:

```
$ docker exec myfirstcontainer ls
```

How it works...

In the preceding task, we created and started a Docker container using the parameter `state`. If we set `state` as `present`, it will only create a container and will not start it. We are using the `command` parameter with the value `sleep infinity` to keep our container running. Docker containers will exit if the foreground process exits. The command `sleep infinity` ensures that the foreground process runs till infinity (theoretically, of course).

Once the Docker container was ready, we used a Docker command to look at the running containers. The command then gave us a list of the running containers; after that, we executed the `ls` command in our Docker container.

Downloading Docker images

In this recipe, we will download a Docker image. In a production environment, while creating a Docker image for an application, we need to add code and the configurations of our application over the top of an existing Docker image (sometimes referred to as a base image), such as Ubuntu, CentOS, and so on. When we deploy a container to a critical environment, we want to minimize the risk of delays. Therefore, it is better to download the image locally before using the `docker_container` module, so as to reduce the deployment time. In this recipe, we will also learn how to create a Docker image using Dockerfile.

How to do it...

1. Let's define the task to pull a Docker image:

```
- name: Download docker image
  docker_image:
    name: ubuntu
  tags:
    - recipe3
```

2. Let's now create a dummy project to build a Docker image:

```
# Let us create our project directory aside our playbook
$ mkdir docker_files

# Let us create our Dockerfile
$ vim Dockerfile

FROM ubuntu:latest
CMD ["sleep", "infinity;"]
```

3. Let's define the task to build an image for our Docker project:

```
- name: Build docker image
  docker_image:
    name: ansible-built
    tag: firstbuilt
    path: docker_files/
    state: present
  tags:
    - recipe3
```

How it works...

In *step 1*, we downloaded a Docker image from Docker Hub by setting the parameter name as provider and the image version. Here, we downloaded an Ubuntu image from Docker Hub.

In *step 2*, we created a very basic project that allows us to build our own Docker image. The project directory contained a Dockerfile, where we defined statements to configure our image.

We used the base image `ubuntu:latest`, and once the container has been created it will execute `sleep infinity` upon startup.

In *step 3*, we used the Ansible module `docker_image` to build a Docker image using Dockerfile from our basic Docker project. We can verify this image using `docker image -a`.

Mounting volumes in containers

In this recipe, we will learn how to mount volumes into a container. Before we start writing tasks, let's take a look at Docker's storage offering. Docker provides a writable layer that is tightly coupled with a host. Data written on the writable layer of a container will be inaccessible once the container has stopped, and will be lost when the container is deleted.

Docker provides three ways of mounting data into a container:

- **Volumes** are the most commonly used mounts for Docker containers. They exist on a specific path of a host's filesystem, which cannot be modified by another process.
- **Bind mounts** can be present anywhere in the filesystem of the host, including files or directories, and they can also be modified by another process running outside the container.
- **Tmpfs mounts** are hosted in memory storage, which never goes into a host's filesystem.

In this recipe, we will create and mount volumes in a container.

How to do it...

1. Let's create a Docker volume for our container:

```
- name: Create volume for docker instance
  docker_volume:
    name: first_volume
  tags:
    - recipe4
```

2. Mount the Docker volume to a defined path in the container:

```
- name: Mount volume to container
  docker_container:
    name: myfirstcontainer
    state: started
    image: ubuntu:14.04
    volumes:
      - first_volume:/app
    command: sleep infinity
  tags:
    - recipe4
```

How it works...

In *step 1*, we created a volume for a Docker container using an Ansible module called `docker_volume`. The same module can be used to delete an existing volume by setting the parameter `state` as `absent`.

In *step 2*, we mounted the volume created in *step 1* to a Docker container, using the Ansible module `docker_container` with the additional parameter `volume`. We specified the volume name created in *step 1* and a mount path inside the container.

This can be verified by using the command `docker inspect myfirstinstance`. This will display the mounts of a container and details about them.

Setting up Docker Registry

Docker Registry is a scalable server where we can store our Docker images. The benefit of running Docker Registry for storing images instead of using Docker Hub is that we can own the images created. Our build or deploy pipelines don't need to rely on Docker Hub. Some other prominent advantages of using a self-hosted Docker Registry are as follows:

- We can co-locate the registry with the application servers, which will reduce the time of deployment
- It helps save bandwidth
- It can help in compliance, in case we need to satisfy something that Docker Hub is missing

In this recipe, we will write a task to create a Docker Registry using a Docker container.

How to do it...

1. Let's define a task to create a Docker Registry:

```
- name: Setting up a docker registry
  docker_container:
    name: registry
    image: registry:2
    exposed_ports:
      - 5000
    ports:
      - 5000:5000
    tags:
      - recipe5
```

2. Let's push a Docker image to the Docker registry created in *step 1*:

```
# Lets pull an image from docker hub
$ docker pull ubuntu
# Tag image to local docker registry
$ docker tag ubuntu localhost:5000/firstimage
# Push image to docker registry
$ docker push localhost:5000/firstimage
```

How it works...

In *step 1*, we used the Docker image `registry` and tag `2` to create a Docker container for setting up Docker Registry. We exposed port `5000` on the Docker container and mapped that port using the parameter `ports` as `5000:5000`, which signify that port `5000` on the host will be mapped to port `5000` on the Docker container.

We should note that the registry application inside our container listens on port `5000`.

In *step 2*, we used Docker commands to test local the Docker Registry by pushing an image to the local registry.

Logging into Docker Registry

In this recipe, we will write a task to log in to Docker Registry. If we are using a private registry and want to push or pull Docker images, we first need to log in to that Docker Registry.

In the following task we will use Docker Hub as the Docker Registry. We will be logging into Docker Hub and will push an image to Docker Hub's private repository.

How to do it...

1. Let's define a task for logging into Docker Registry:

```
- name: Log into DockerHub
  docker_login:
    username: vikas17a
    password: XXXXXXXXX
    email: vikas17a@gmail.com
  tags:
    - recipe6
```

2. Let's try to push an image to Docker Hub's private repository using the login:

```
- name: Push image to Dockerhub
  docker_image:
    repository: vikas17a/cookbook
    name: cookbook
    tag: firstbuilt
    path: docker_files/
    push: yes
  tags:
    - recipe6
```

How it works...

In *step 1*, we used the Ansible module `docker_login` to log in to a Docker Registry, in our case, Docker Hub. We should note that `docker_login` uses Docker Hub as the default registry; that is, `https://index.docker.io/v1/`. We can change the Docker Registry to a privately-hosted Docker Registry by using an additional parameter: `registry_url`.

In *step 2*, we pushed a Docker image to a private Docker Registry. We should note that if we skip `docker_login` and try to push an image to Docker Hub, it will fail for private registries.

Using Docker Compose to manage services

In this recipe, we will learn about how to use Docker Compose for managing services. Before we jump into defining a task for Docker Compose, we should know what Docker Compose is about.

Docker Compose is a tool that can define multiple containers that are required for a service in a single file. It can bring up all containers, network links, volumes, and so on with just a single command.

Docker Compose's main functionality is to support microservice architecture, that is, bringing up containers and the links between them. Docker Compose's scope, however, is limited to a single host. If we want to use multiple hosts, we should instead look at projects, such as Kubernetes and Docker Swarm.

How to do it...

1. Let's create a basic Docker Compose project in a directory named `docker_compose`:

```
$ mkdir -p docker_compose && cd docker_compose
```

2. We will create a sample web application that uses Python Flask and Redis to count hits on a website:

```
$ vim app.py
from flask import Flask
from redis import Redis
app = Flask(__name__)
redis = Redis(host='redis', port=6379)
@app.route('/')
def hello():
    count = redis.incr('hits')
    return 'Hello World! I have been seen {} times.\n'.format(count)
if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

3. We will add the `requirements.txt` file to our application in the same directory:

```
$ vim requirements.txt
flask
redis
```

4. Let's create a `Dockerfile` for the application:

```
$ vim Dockerfile

FROM python:3.4-alpine
ADD . /code
WORKDIR /code
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

5. Finally, we will create our Docker Compose file:

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
```

```
redis:
  image: "redis:alpine"
```

6. Let's now define an Ansible task to bring our application up using `docker-compose.yml`:

```
- name: Create application using compose
  docker_service:
    project_src: docker_compose
    state: present
  tags:
    - recipe7
```

7. Verify that the service is up and running using `docker-compose.yml`:

```
$ curl localhost:5001
Hello World! I have been seen 1 times
```

How it works...

From *step 1* through to *step 5*, we created a basic application inside the directory `docker_compose` along with our Ansible task. We added a simple Flask application to serve requests with the `Hello World!` response. To demonstrate a use-case of Docker Compose, we added a dependency to our application, using Redis to display a website's total number of hits. We created a `Dockerfile` containing statements to build our application container. Finally, we created `docker-compose.yml`, where we added dependencies for the application, that is, Redis.

Here, we have defined two services: one is a web-based service, which is a Flask application, and the other is Redis, which acts as a data store for the web service.

In *step 6*, we used the Ansible module `docker_service` and set our project path as `docker_compose`. The Ansible module will find `docker-compose.yml` present in the project path, and will start building and running the required containers using `docker-compose`. Then, our application will be ready.

In *step 7*, we verified that our application was up and running using the `curl` command.

Scaling up Compose-based service

In this recipe, we will scale up our service, which we created in the recipe, *Using Docker Compose to manage a service*. Scaling up a Compose-based service simply means adding more containers for a particular service defined in `docker-compose.yml`. We will be using Ansible to scale the web service of an application. Before we jump further into this topic and scale a web service, we should understand the concept of load balancers. Usually, web applications that are horizontally scalable like ours (our web service) need a load balancer to distribute requests among all instances of the application. In our example, we have not yet created a load balancer, so we need to create a load balancer using a Docker Hub image.

How to do it...

1. Let's add a load balancer, links, and a network in `docker-compose.yml`:

```
version: '2'
services:
  web:
    build:
    ports:
      - "5000"
    networks:
      - l1-tier
      - l2-tier
  redis:
    image: "redis:alpine"
    links:
      - web
    networks:
      - l2-tier
  lb:
    image: dockercloud/haproxy
    ports:
      - "5001:80"
    links:
      - web
    networks:
      - l1-tier
      - l2-tier
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock

networks:
```

```
l1-tier:
  driver: bridge
l2-tier:
  driver: bridge
```

2. Let's run `docker_service` to create Docker Compose again. This will re-build new components for our compose application:

```
$ ansible-playbook -i localhost, docker.yml -t recipe7
```

3. Let's write a task to scale the web application:

```
- name: Scale up web service of compose application
  docker_service:
    project_src: docker_compose
    scale:
      web: 2
  tags:
    - recipe8
```

4. Now, verify if both web containers are serving requests:

```
# Lets send multiple requests
$ for i in `seq 1 10`; do curl localhost:5001; sleep 1; done
# Lets verify logs of both web service containers
$ docker logs dockercompose_web_1
$ docker logs dockercompose_web_2
```

How it works...

In *step 1*, we modified the existing `docker-compose.yml` to add a load balancer and network configurations for our application. We should note that the load balancer we are using here is a container build from an image fetched from Docker Hub, which looks for a Docker engine socket and load balances the traffic between containers linked on the same network.

In *step 2*, we used an existing task from the recipe *Using Docker Compose to manage a service* to rebuild our application using `docker-compose.yml`.

In *step 3*, we defined a task for scaling out a web service using the Ansible module `docker_service` and the parameter `scale` with its values defined as name of service and count of containers to scale out.

In *step 4*, we verified if both of our containers were serving requests, firstly by initiating 10 requests using the `bash` for loop and `curl` command, and secondly by looking into logs of our web service containers using the `docker logs` command.

Deploying a sample application

In this recipe, we will be deploying our favorite phonebook application using Docker containers. Referring to the code files, we have created a `Dockerfile` and added the steps to bake our application into a Docker image. Then, we created `docker-compose.yml`, in which we have added our service as a phonebook and mapped it to the port of the host container, which is 8080.

How to do it...

Let's write a playbook for the application on how to build and run a phonebook service using `ansible-playbook`:

```
---
- hosts: localhost
  roles:
    - phonebook

$ ansible-playbook --become phonebook.yml
```

How it works...

After we have run `ansible-playbook`, Ansible will execute `docker-compose` using the module `docker_service` and will map the host port 8080 to the container's port, 8080. Once the playbook has finished executing, we can browse `localhost:8080`, and our application will be ready.

8

Diving into OpenStack with Ansible

In this chapter, we will cover the following recipes:

- Preparing Ansible to work with OpenStack
- Adding a keypair
- Managing security groups
- Managing network resources
- Managing Nova compute instances
- Creating a Cinder volume and attaching it to a Nova compute instance
- Managing objects in Swift
- User management
- Creating a flavor
- Adding an image
- Dynamic inventory
- Deploying the phonebook application

Introduction

In previous chapters, we looked at a bunch of popular cloud providers. One of the biggest advantages of using a third-party cloud provider is that we can get started in minutes; provisioning the actual hardware is no longer our problem. There are several other advantages of using a third-party cloud provider. However, there is a major drawback: the cloud is not customized to our needs. It may lack flexibility, which we can gain if we control the underlying hardware and network. OpenStack can help us address this problem.

OpenStack is software that can help us build a system similar to popular cloud providers, such as AWS or GCP. OpenStack provides an API and a dashboard to manage the resources that it controls. Basic operations, such as creating and managing virtual machines, block storage, object storage, identity management, and so on, are supported out of the box. In the case of OpenStack, we control the underlying hardware and network, which comes with its own pros and cons. We have to keep the following points in mind while managing the OpenStack setup:

- We need to maintain the underlying hardware. Consequently, we have to do capacity planning since our ability to scale resources would be limited by underlying hardware. However, since we control the hardware, we can get customized resources in virtual machines.
- We can use custom network solutions. We can use economical equipment or high-end devices, depending upon the actual need. This can help us get the features that we want and may end up saving money.
- We need to regularly update the hypervisor and other OpenStack dependencies, especially in the case of a security-related issue. This can be a time-consuming task because we might need to move the running virtual machines around to ensure that users do not face a lot of trouble.
- OpenStack can be helpful in cases where strict compliance requirements might not allow us to use a third-party cloud provider. A typical example of this is that certain countries require financial and medical data to stay inside their jurisdiction. If any third-party cloud is not able to fulfill this condition, then OpenStack is a great choice.



Caution: Although OpenStack can be hosted on premises, several cloud providers provide OpenStack as a service. Sometimes these cloud providers may choose to turn off certain features or provide add-on features. Sometimes, even while configuring OpenStack on a self-hosted environment, we may choose to toggle certain features or configure a few things differently. Therefore, inconsistencies may occur. All the code examples in this chapter are tested on a self-hosted OpenStack released in August 2017, named Pike. The underlying operating system was CentOS 7.4.

To help us get started, we have compiled a table that shows us the approximate equivalent projects of Amazon Web Services and OpenStack.

Utility	AWS	OpenStack
Virtual machines	EC2	Nova
Block storage	EBS	Cinder
Object storage	S3	Swift
Identity management	IAM	Keystone
Network management	VPC	Neutron
Dashboard	AWS Web Console	Horizon

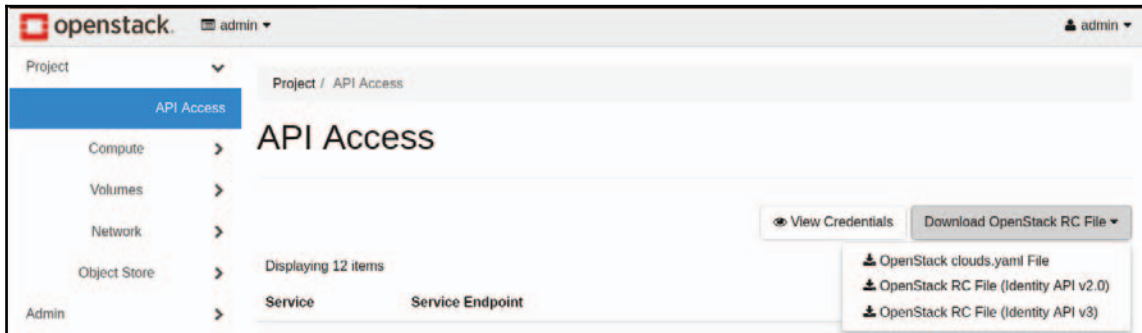
Preparing Ansible to work with OpenStack

OpenStack exposes a set of web APIs that can be used to manage resources. In order to interact with these APIs, Ansible needs to authenticate with OpenStack. The `shade` Python library helps in authenticating with the OpenStack server. So let's go ahead and install the required dependency using the following command:

```
# pip install shade
```


How to do it...

We need to set up a few environment variables on the server that we will execute the Ansible code from. These variables will be required for authenticating with OpenStack. An easy way to do this is to download the environment variable file from the OpenStack dashboard (called **Horizon**).



From the Horizon dashboard, go to **API Access** from the left sidebar. From here, click on the **Download OpenStack RC File | OpenStack RC File (Identity API v3)** file. This file consists of all the required variables that we need to set up in the environment. We can just execute this Bash script and all the variables will be exported. It may ask for the OpenStack password of the user, which will be set as an environment variable as well.

In case the dashboard is not available, here are the variables from our setup:

```
OS_AUTH_URL=http://192.168.0.102:5000/v3
OS_ENDPOINT_TYPE=publicURL
OS_TENANT_ID=3252fc69b58240ea86b010684fd12ff1
OS_REGION_NAME=RegionOne
OS_USER_DOMAIN_NAME=Default
OS_IDENTITY_API_VERSION=2
OS_TENANT_NAME=admin
OS_PASSWORD=8a569982407a4298
OS_USERNAME=admin
```

Once we have the environment variables set, we are ready to use Ansible on OpenStack.

Adding a keypair

A keypair is used to log into a virtual machine through SSH created by OpenStack. Once we have a key, or several of them, then we can choose a public key while creating a virtual machine, and this key will be added to the default user of the operating system.

How to do it...

1. For this, we need to have a key added to OpenStack before we boot the virtual machine. The name of the key has to be unique. This is what we will specify while booting an instance. So let's add our public key as follows:

```
- name: adding public key
  os_keypair:
    name: aditya
    public_key: {{ aditya_pub_key }}
```

2. We have used the variable for improving the readability of the code. It also helps if we need to change the keys quickly. We will add the value of the `aditya_pub_key` variable to the `vars/main.yml` file, as follows:

```
---
aditya_pub_key: ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQCbG83WYIxFXWJ4bQiYfZYHceDwMJxnGfJqgY
tHL/DAtEGvY+Nm8MX3CRZYisfskt0m9CQ6y/Ux1OITTz+O11fgxLJcroZmKJbWW0K39
gfHvFqR7Fie0zuJaxqUQUuyc0i6RCBRiZPiQQOPes2yDtfHgDWx3q9knS3ZXIAXcGLZ
rgfC1XnIK8CLAnZDved9Rue2bhsCnO9M1eh9g/CTtehMDAzD4NeSv9eETlHYkYSpJg8
gFA3BFICpBxTqWSjff1mMQGSmiudFOhRjHIXL1Tvh+pnjSoL/jrLcP3RtMVuG0ZU0qko
Ats1qpTwmyAJUz9Ts2EeyDJ0tXsFAiOFbmuMd aditya@devopsnexus.com
```

3. If we ever need to, we can delete an existing key from OpenStack by using the `state` parameter with the task, as follows:

```
- name: delete a public key
  os_keypair:
    name: aditya
    state: absent
```



We should note that this will not remove the keys installed in already-running virtual machines—we need to do that on our own. Of course, Ansible is a great tool to do that as well.

Managing security groups

Security groups are the firewalls that can be used to allow or disallow the flow of traffic. They can be applied to virtual machines. Security groups and virtual machines have a many-to-many relationship. A single security group can be applied to multiple virtual machines and a single virtual machine can have multiple security groups.

How to do it...

1. Let's create a security group as follows:

```
- name: create a security group for web servers
  os_security_group:
    name: web-sg
    state: present
    description: security group for web servers
```

The `name` parameter has to be unique. The `description` parameter is optional, but we recommend using it to state the purpose of the security group. The preceding task will create a security group for us, but there are no rules attached to it. A firewall without any rules is of little use. So let's go ahead and add a rule to allow access to port 80 as follows:

```
- name: allow port 80 for http
  os_security_group_rule:
    security_group: web-sg
    protocol: tcp
    port_range_min: 80
    port_range_max: 80
    remote_ip_prefix: 0.0.0.0/0
```

2. We also need SSH access to this server, so we should allow port 22 as well:

```
- name: allow port 80 for SSH
  os_security_group_rule:
    security_group: web-sg
    protocol: tcp
    port_range_min: 22
    port_range_max: 22
    remote_ip_prefix: 0.0.0.0/0
```

How it works...

For this module, we need to specify the name of the security group. The rule that we are creating will be associated with this group. We have to supply the protocol and the port range information. If we just want to whitelist only one port, then that would be the upper and lower bound of the range. Lastly, we have to specify the allowed addresses in the form of CIDR. The address `0.0.0.0/0` signifies that port 80 is open for everyone. This task will add an ingress type rule and allow traffic on port 80 to reach the instance. Similarly, we have to add a rule to allow traffic on port 22 as well.

Managing network resources

A network is a basic building block of the infrastructure. Most of the cloud providers will supply a sample or default network. While setting up a self-hosted OpenStack instance, a single network is typically created automatically. However, if the network is not created, or if we want to create another network for the purpose of isolation or compliance, we can do so using the `os_network` module.

How to do it...

1. Let's go ahead and create an isolated network and name it private, as follows:

```
- name: creating a private network
  os_network:
    state: present
    name: private
```

2. In the preceding example, we created a logical network with no subnets. A network with no subnets is of little use, so the next step would be to create a subnet:

```
- name: creating a private subnet
  os_subnet:
    state: present
    network_name: private
    name: app
    cidr: 192.168.0.0/24
    dns_nameservers:
      - 8.8.4.4
      - 8.8.8.8
```

```
host_routes:
  - destination: 0.0.0.0/0
    nexthop: 104.131.86.234
  - destination: 192.168.0.0/24
    nexthop: 192.168.0.1
```

How it works...

The preceding task will create a subnet named `app` in the network called `private`. We have also supplied a CIDR for the subnet, `192.168.0.0/24`. We are using Google DNS for nameservers as an example here, but this information should be obtained from the IT department of the organization. Similarly, we have set up the example host routes, but this information should be obtained from the IT department as well.

After successful execution of this recipe, our network is ready to use.

Managing a Nova compute instance

Managing Nova compute instances, or virtual machines, is one of the main operations that we will be doing while working with OpenStack. This is where we run our applications. All of the previous recipes were steps toward creating a compute instance.

How to do it...

1. Let's begin by creating a compute instance, as follows:

```
- name: Deploy an instance
  os_server:
    state: present
    name: webserver
    image: cirros
    key_name: aditya
    security_groups: web-sg
    wait: yes
    flavor: m1.tiny
    auto_floating_ip: yes
    network: private
  meta:
    hostname: webserver.localdomain
```

2. If at any point we need to stop the instance, then we can use the `os_server_action` module as follows:

```
- name: stop the webserver
  os_server_action:
    action: stop
    server: webserver
```

How it works...

In *step 1*, for creating a compute instance, we have specified a name that will be used to refer to the virtual machine at all times. We have also specified an image called `cirros`. Cirros (<https://launchpad.net/cirros>) is a Linux flavor that specializes in running on the cloud. We can choose any available image, such as CentOS or Fedora, provided they are available on our OpenStack setup. We can add images of operating systems that we want—we will look at how to do this later. We have also specified the key and the security group created in the previous tasks. We have set the `wait` parameter to **yes**. Because of this, Ansible will wait for the instance to be created. This is good for synchronous tasks. The `flavor` parameter, `m1.tiny`, is predefined and manages the resources allocated to the compute instances. By default, it is 512 MB RAM and 1 core of virtual CPU. We have also requested for a floating IP address to be attached to this instance. This instance will be booted in the private network that we created in a previous task. Finally, we specify a metadata key to set the hostname of the instance.

In *step 2*, the task is useful when we want to save resources by shutting down an instance, but we do not want to delete it. To start the instance again, we just need to set the **action** parameter to **start**.

Creating a Cinder volume and attaching it to a Nova compute instance

Cinder provides a block storage mechanism, which is ideal to use with Nova for mounting disks. One disk volume is created when we create a Nova compute instance. However, often we need to attach multiple disk volumes to a compute instance. Cinder lets us create more volumes, and we can connect them to our compute instances.

How to do it...

Let's start by creating a 5 GB volume:

```
- name: create 5G test volume
  os_volume:
    state: present
    size: 5
    display_name: data
```

The `os_volume` module takes a size and a display name as parameters and creates a volume. This volume is not attached to the compute instance yet. We'll attach this volume to our webserver compute instance in the next task.

Let's go ahead and attach the volume created in the previous task to the `webserver` compute instance that we created before:

```
- name: attach volume to host
  os_server_volume:
    state: present
    server: webserver
    volume: data
```

The preceding task is going to attach the `volume` with the name `data` to a compute instance named `webserver`. We can log into the instance and verify this. We may format the disk for a suitable filesystem, such as `ext4`.

Managing objects in Swift

Swift provides object storage for OpenStack. Here, we create containers and upload objects inside the containers. Remember that a container is a virtual entity for organizing the data better.

How to do it...

1. Let's begin with creating a container, as follows:

```
- name: create an object container
  os_object:
    state: present
    container: backup
```

2. Now let's push a sample object here, as follows:

```
- name: upload backup.tar to backup container
  os_object:
    filename: /opt/backup.tar
    container: backup
    name: backup.tar
    state: present
```

How it works...

Step 1 will create a container named `backup` where we can push any object. This is a very typical use case for storing daily backups of databases.

In *step 2*, we specify the path of the file we want to upload to the container. We will also supply a container name. The `name` parameter should be a name of the key to which the file will be uploaded. This can be a path within the container.



As a side note, we recommend encrypting and numbering the backups.

User management

OpenStack provides an elaborate user management mechanism. If we are coming from a typical third-party cloud provider, such as AWS or GCP, then it can look overwhelming. The following list explains the building blocks of user management:

- **Domain:** This is a collection of projects and users that define an administrative entity. Typically, they can represent a company or a customer account. For a self-hosted setup, this could be done on the basis of departments or environments. A user with administrative privileges on the domain can further create projects, groups, and users.
- **Group:** A group is a collection of users owned by a domain. We can add and remove privileges from a group and our changes will be applied to all the users within the group.

- **Project:** A project creates a virtual isolation for resources and objects. This can be done to separate departments and environments as well.
- **Role:** This is a collection of privileges that can be applied to groups or users.
- **User:** A user can be a person or a virtual entity, such as a program, that accesses OpenStack services.



For a complete documentation of the user management components, go through the OpenStack Identity document at <https://docs.openstack.org/keystone/pike/admin/identity-concepts.html>.

How to do it...

Let's go ahead and start creating some of these basic building blocks of user management. We should note that, most likely, a default version of these building blocks will already be present in most of the setups:

1. We'll start with a domain called `demodomain`, as follows:

```
- name: creating a demo domain
  os_keystone_domain:
    name: demodomain
    description: Demo Domain
    state: present
    register: demo_domain
```

2. After we get the domain, let's create a role, as follows:

```
- name: creating a demo role
  os_keystone_role:
    state: present
    name: demorole
```

3. Projects can be created, as follows:

```
- name: creating a demo project
  os_project:
    state: present
    name: demoproject
    description: Demo Project
    domain_id: "{{ demo_domain.id }}"
    enabled: True
```

4. Once we have a role and a project, we can create a group, as follows:

```
- name: creating a demo group
  os_group:
    state: present
    name: demogroup
    description: "Demo Group"
    domain_id: "{{ demo_domain.id }}"
```

5. Let's create our first user:

```
- name: creating a demo user
  os_user:
    name: demouser
    password: secret-pass
    update_password: on_create
    email: demo@example.com
    domain: "{{ demo_domain.id }}"
    state: present
```

6. Now we have a user and a group. Let's add the user to the group that we created before:

```
- name: adding user to the group
  os_user_group:
    user: demouser
    group: demogroup
```

7. We can also associate a user or a group with a role:

```
- name: adding role to the group
  os_user_role:
    group: demo2
    role: demorole
    domain: "{{ demo_domain.id }}"
```

How it works...

In *step 1*, the `os_keystone_domain` would take a name as a mandatory parameter. We also supplied a description for our convenience. We are going to use the details of this domain, so we saved it in a variable called `demo_domain`.

In *step 2*, the `os_keystone_role` would just take a name and create a role. Note that a role is not tied up with a domain.

In *step 3*, the `os_project` module would require a name. We have added the description for our convenience. The projects are tied to a domain, so we have used the `demo_domain` variable that we registered in a previous task.

In *step 4*, the groups are tied to domains as well. So, along with the name, we would specify the description and domain ID like we did before. At this point, the group is empty, and there are no users associated with this group.

In *step 5*, we supply `name` along with a password for the user. The `update_password` parameter is set to `on_create`, which means that the password won't be modified for an existing user. This is great for the sake of idempotency. We also specify the email ID. This would be required for recovering the password and several other use cases. Lastly, we add the domain ID to create the user in the right domain.

In *step 6*, the `os_user_group` module will help us associate the `demouser` with the `demogroup`.

In *step 7*, the `os_user_role` will take a parameter for `user` or `group` and associate it with a role.

A lot of these divisions might not be required for every organization. We recommend going through the documentation and understanding the use case of each of them. Another point to note is that we might not even see the user management bits on a day-to-day basis. Depending on the setup and our responsibilities, we might only interact with modules that involve managing resources, such as virtual machines and storage.

Creating a flavor

When we boot a compute instance, we provide a flavor as a parameter. A flavor defines the resources of a compute instance, such as CPU, memory, and storage. On the Pike release of OpenStack, the default flavors are set as follows:

Flavor	VCPUs	Disk (in GB)	RAM (in MB)
m1.tiny	1	1	512
m1.small	1	20	2048
m1.medium	2	40	4096
m1.large	4	80	8192
m1.xlarge	8	160	16384

However, in many cases, the default flavors are not suitable for our needs.

How to do it...

OpenStack lets us create our own flavors, as follows:

```
- name: Create a custom flavor
  os_nova_flavor:
    name: custom1
    ram: 1024
    vcpus: 1
    disk: 10
    ephemeral: 10
    state: present
```

The `os_nova_flavor` module takes a name and parameters for the desired resources to create a custom flavor. Once the flavor is created, we can use the dashboard or `os_server` Ansible module to create a compute instance in the usual way, but with the new flavor.

Adding an image

We need to update our OpenStack setup periodically with the latest releases of our operating systems. This is a good practice to get various bug fixes and security patches. Most of the popular Linux projects, such as CentOS, Fedora, Debian, and Ubuntu, release `qcow2` images or other compatible formats to support private clouds such as OpenStack. For our convenience, OpenStack documents some of the major projects on their website (<https://docs.openstack.org/image-guide/obtain-images.html>).

How to do it...

As an example, we will work with the Fedora Project's cloud image. To do so, we will download the `qcow2` image from the Fedora Project's download page, at <https://alt.fedoraproject.org/cloud/>:

```
- name: adding Fedora 27 as an image
  os_image:
    name: fedora-cloud-27
    container_format: bare
    disk_format: qcow2
    id: "{{ ansible_date_time.epoch | to_uuid }}"
    filename: /opt/Fedora-Cloud-Base-27-1.6.x86_64.qcow2
    state: present
```

How it works...

The `os_image` module will take the user-defined name of the image as a parameter. We will define the container format and disk format. The ID parameter takes a UUID, which has to be unique. We can manually supply one or use Ansible's `to_uuid` filter. If we are supplying one, then the easiest way to get a UUID in the correct format is to use the `uuidgen` command on shell. Lastly, we specify the path of the downloaded `qcow2` image on our local system. This `qcow2` image will be uploaded to OpenStack and an image will be created from which we can boot compute instances.

Dynamic inventory

Managing the inventory file manually can be a problem. In an environment where instances are booted up and destroyed very frequently, a text-based inventory file can easily go out of sync from the actual infrastructure. In such cases, we can use dynamic inventory, which will help us generate the list of compute instances and other resources in real time. Ansible's GitHub repository has a dynamic inventory script for OpenStack.

How to do it...

Let's download the dynamic inventory script and set the executable bit as follows:

```
$ curl
https://raw.githubusercontent.com/ansible/ansible/devel/contrib/invento
ry/openstack.py > openstack.py
$ chmod +x openstack.py
```

After we set the executable bit, we need to set the environment variables so that the inventory script can authenticate with the OpenStack keystone. The easiest way to set up the environment variables is to execute the script that we downloaded from the **Horizon** dashboard in the first recipe, *Preparing Ansible to work with OpenStack*.

Once the environment variables are set, we can execute the following command and check whether or not all the resources are getting reported:

```
$ ./openstack.py --list
{
  "RegionOne": [
    "667af6f2-51ed-4718-a525-1d990936b518"
  ],
  "RegionOne_nova": [
    "667af6f2-51ed-4718-a525-1d990936b518"
  ],
  "_meta": {
    << snipped >>
  },
  "envvars": [
    "667af6f2-51ed-4718-a525-1d990936b518"
  ],
  "envvars_RegionOne": [
    "667af6f2-51ed-4718-a525-1d990936b518"
  ],
  "envvars_RegionOne_nova": [
    "667af6f2-51ed-4718-a525-1d990936b518"
  ],
  "flavor-m1.tiny": [
    "667af6f2-51ed-4718-a525-1d990936b518"
  ],
  "image-cirros": [
    "667af6f2-51ed-4718-a525-1d990936b518"
  ],
  "instance-667af6f2-51ed-4718-a525-1d990936b518": [
    "667af6f2-51ed-4718-a525-1d990936b518"
  ],
  "meta-hostname_webserver.localdomain": [
```

```
    "667af6f2-51ed-4718-a525-1d990936b518"
  ],
  "nova": [
    "667af6f2-51ed-4718-a525-1d990936b518"
  ],
  "webserver": [
    "667af6f2-51ed-4718-a525-1d990936b518"
  ]
}
```

We have left out a part of the output to make it more readable. As we can see, dynamic inventory gives us several groups that we can use while invoking our playbooks, so we can execute the playbook using any of these groups, or even a combination of them. For example, if we want to execute the playbook on all the servers running `cirros`, perhaps to apply a critical security patch, then we can just create the playbook like this:

```
---
- hosts: image-cirros
  roles:
    - myrole
```

This way, we do not need to build and maintain an inventory manually. This also creates some common groups that we would not have created in most normal cases, such as groups based on flavors and regions.

Deploying the phonebook application

We are going to deploy our phonebook app to OpenStack. For this setup, we are going to use the phonebook with SQLite. Our phonebook role is created for CentOS and uses package names and repositories related to CentOS. For the following recipe, we can download the CentOS image from their website (<http://cloud.centos.org/centos/7/images/>) and add it to OpenStack using the `os_image` module in a similar way to the method we used in the recipe *Adding an image*. We can use the `os_server` module just as we used it in the recipe *Managing a Nova compute instance*, and we can create a compute instance using that image. Let's name this instance `phonebook`. To target this instance in our playbook, we will be using dynamic inventory.

How to do it...

1. Let's create the playbook for the phonebook app that we created in Chapter 2, *Using Ansible to Manage AWS EC2*:

```
---  
- hosts: phonebook  
  roles:  
    - phonebook
```

2. We will execute this playbook with dynamic inventory:

```
$ ansible-playbook -i openstack.py --become playbook.yml
```

Once we finish the execution of the playbook successfully, we will be able to browse the phonebook on port 8080 of the webserver instance. The `--become` flag is required because the phonebook role installs certain packages, that requires superuser privileges.

9

Ansible Tower

In this chapter, we will cover the following recipes:

- Installing Ansible Tower
- Getting started with Tower
- Adding a machine credential
- Building a simple inventory
- Executing ad-hoc commands
- Using Ansible Tower with a cloud provider
- Integrating Ansible roles with Tower
- Scheduling jobs
- Ansible Tower API
- Autoscaling using Callback

Introduction

Ansible is one of the simplest configuration management and orchestration tools out there. To make things even simpler, we are going to check out Ansible Tower in this chapter. Tower provides a web-based user interface that executes Ansible's codebase. On top of that, it comes with many features including access control, security, better logging, and workflows. It has an inbuilt mechanism to schedule jobs and notify users about key events. Our favorite feature, dynamic inventory, is pre-baked into Ansible Tower, making it easier to manage cloud setups.

Installing Ansible Tower

Ansible Tower is a licensed tool. However, it can be used to manage, at most, 10 servers using a free license. The Tower setup can be downloaded from Ansible's official website (<https://www.ansible.com/products/tower>). Once the compressed tar archive is obtained, then we can install Ansible Tower. It should not come as surprise that we have to use Ansible for installing Ansible Tower.

Another point to note is that Ansible Tower is only supported for the following operating systems:

- Red Hat Enterprise Linux 6 64-bit
- Red Hat Enterprise Linux 7 64-bit
- CentOS 6 64-bit
- CentOS 7 64-bit
- Ubuntu 12.04 LTS 64-bit
- Ubuntu 14.04 LTS 64-bit

More detailed requirements can be found inside Ansible Tower's documentation (<http://docs.ansible.com/ansible-tower/2.4.0/html/quickinstall/prepare.html#primary-requirements>).

How to do it...

We will go ahead and set up Ansible Tower on CentOS 7. As a prerequisite, we need to make sure that Ansible is already installed:

1. As we did in the first recipe *Install Ansible*, in the [Chapter 1, Getting Started with Ansible and Cloud Management](#), let us install Ansible using Python Package Manager (pip):

```
$ sudo pip install ansible==2.4.0.0
```

2. Next up, we extract the archive to get the Ansible Tower setup:

```
$ tar -xzf ansible-tower-setup-latest.tar.gz
```

3. Now we are going to get into the `setup` directory and set some password variables:

```
$ cd ansible-tower-setup-3.2.2
$ vim inventory
[tower]
localhost ansible_connection=local
[database]
[all:vars]
admin_password='mysecretpassword'
pg_host=''
pg_port=''
pg_database='awx'
pg_username='awx'
pg_password='mysecretpassword'
rabbitmq_port=5672
rabbitmq_vhost=tower
rabbitmq_username=tower
rabbitmq_password='mysecretpassword'
rabbitmq_cookie=cookiemonster
# Needs to be true for fqdns and ip addresses
rabbitmq_use_long_name=false
# Isolated Tower nodes automatically generate an RSA key for
authentication;
# To disable this behavior, set this value to false
# isolated_key_generation=true
```

4. Run the `setup` script, which will call the Ansible playbooks in the right order:

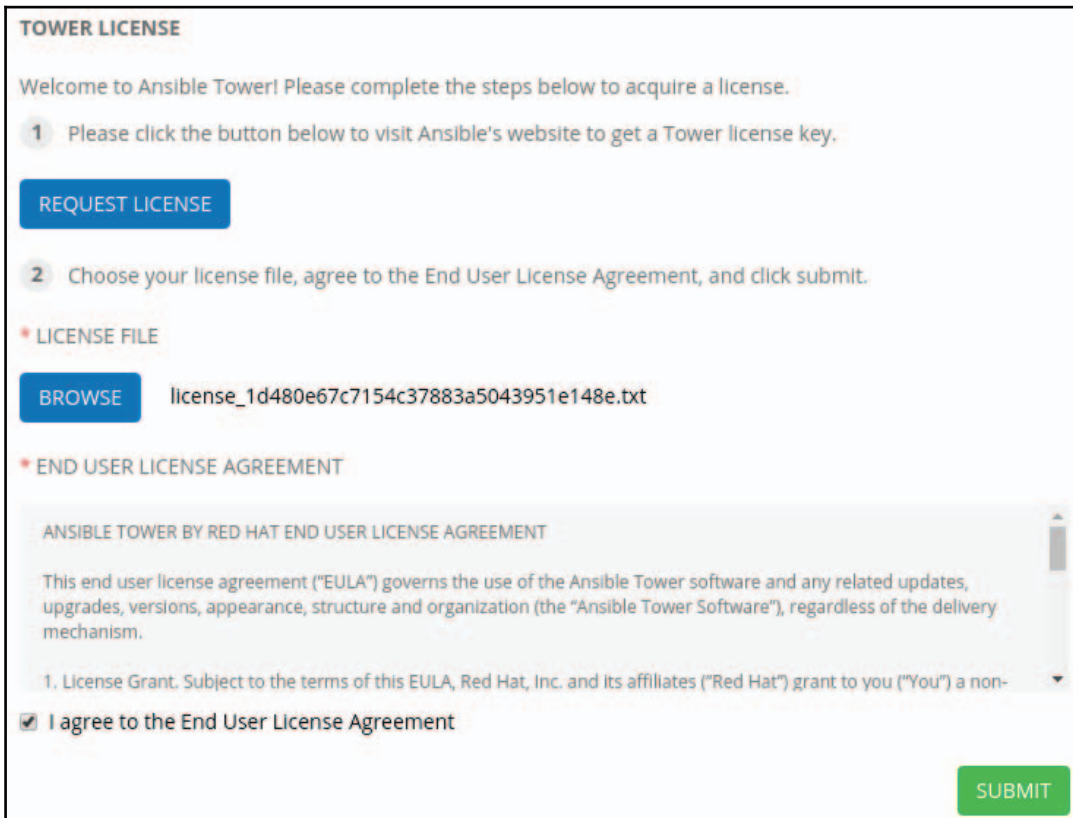
```
$ sudo ./setup.sh
```

Getting started with Tower

Once the setup script has finished, we can browse the dashboard using a web browser. We have to log in as an admin user, using the same password we just used in the inventory file. When we log in to Ansible Tower for the first time, it asks for a license. So, let's provide a license to Ansible Tower.

How to do it...

After the first login, we are required to supply a license file to Ansible Tower. To do this, we can click on the **REQUEST LICENSE** button and get a license. There are two kinds of license: the first is an Enterprise License, which unlocks features such as LDAP integration and enterprise support, but this license has a trial period of 30 days. The second one is the Limited Node License, which does not have a trial period associated with it, but can only manage up to 10 hosts. For this chapter, we are going to get the second license.



The screenshot shows the 'TOWER LICENSE' page in Ansible Tower. It contains a welcome message and two numbered steps. Step 1 instructs the user to click the 'REQUEST LICENSE' button. Step 2 instructs the user to choose a license file, agree to the EULA, and click submit. Under 'LICENSE FILE', there is a 'BROWSE' button and a text field containing 'license_1d480e67c7154c37883a5043951e148e.txt'. Under 'END USER LICENSE AGREEMENT', there is a scrollable area showing the 'ANSIBLE TOWER BY RED HAT END USER LICENSE AGREEMENT' text, followed by a checkbox labeled 'I agree to the End User License Agreement' which is checked. A green 'SUBMIT' button is located at the bottom right.

TOWER LICENSE

Welcome to Ansible Tower! Please complete the steps below to acquire a license.

- 1 Please click the button below to visit Ansible's website to get a Tower license key.

REQUEST LICENSE

- 2 Choose your license file, agree to the End User License Agreement, and click submit.

LICENSE FILE

BROWSE license_1d480e67c7154c37883a5043951e148e.txt

END USER LICENSE AGREEMENT

ANSIBLE TOWER BY RED HAT END USER LICENSE AGREEMENT

This end user license agreement ("EULA") governs the use of the Ansible Tower software and any related updates, upgrades, versions, appearance, structure and organization (the "Ansible Tower Software"), regardless of the delivery mechanism.

1. License Grant. Subject to the terms of this EULA, Red Hat, Inc. and its affiliates ("Red Hat") grant to you ("You") a non-

☒ I agree to the End User License Agreement

SUBMIT

Once we get the license text file, we can download it to our computer and click on the **BROWSE** button to supply the location of the license. Clicking **Submit** would present the dashboard:



Adding a machine credential

Ansible Tower uses credentials for various types of authentication. Some of the most common uses of credentials are:

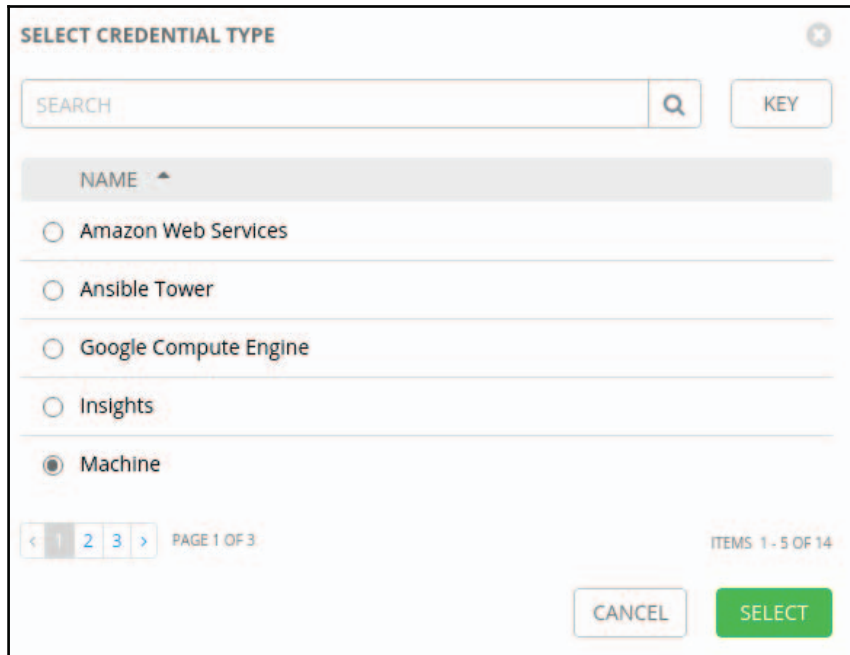
- SSH authentication for executing Ansible playbooks
- Synchronizing dynamic inventory
- Fetching Ansible playbooks from a version control

Let's now take a look at how we can use credentials for SSH authentication.

How to do it...

We can find the **Credentials** page by clicking on the settings button (the gear icon on top-right corner) and then selecting the **Credentials** section. We should notice that a **Demo Credential** has already been created by Ansible Tower. Let's instead create our own credential by clicking on the **+ADD** button.

After clicking the **+ADD** button, we need to supply the name for identifying the key and the credential type. For SSH, we have to set the **CREDENTIAL TYPE** as **Machine**:



The screenshot shows a modal window titled "SELECT CREDENTIAL TYPE". At the top, there is a search bar with the placeholder text "SEARCH" and a magnifying glass icon. To the right of the search bar is a button labeled "KEY". Below the search bar is a list of credential types, each with a radio button and a label: "Amazon Web Services", "Ansible Tower", "Google Compute Engine", "Insights", and "Machine". The "Machine" option is selected, indicated by a filled radio button. At the bottom of the list, there is a pagination control showing "< 1 2 3 >" and "PAGE 1 OF 3". To the right of the pagination control is the text "ITEMS 1 - 5 OF 14". At the bottom right of the modal are two buttons: "CANCEL" and "SELECT".

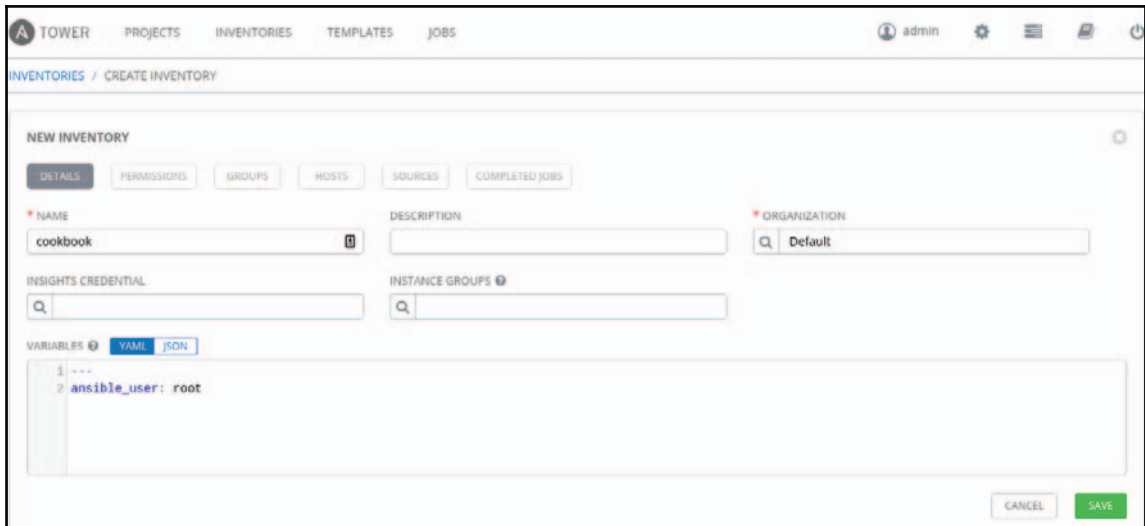
Once we have selected the type, the form on the page will extend and we should be able to supply a username and private key. Ansible Tower encrypts the key with 256 bit AES encryption, so our private key is secure with Ansible Tower. For our example, we are just going to set the username and private key, but it is also possible to set a password, private key passphrase, and privilege escalation details. We need to click on the **SAVE** button after we have input all the details. Once the private key or password has been saved, Ansible Tower will never show it to other users; it would only be decrypted as and when it actually needs to use the credentials.

Building a simple inventory

We need to have an inventory before we execute any commands or playbooks, so let's build an inventory and add a host to run some tasks.

How to do it...

The **INVENTORIES** tab is located on the top bar. A **Demo Inventory** has already been created for us. We will create a new inventory and add a host there. Let's click on the **+ADD** button and select inventory from the menu.



The screenshot shows the 'NEW INVENTORY' form in the Ansible Tower web interface. The top navigation bar includes 'TOWER', 'PROJECTS', 'INVENTORIES', 'TEMPLATES', and 'JOBS'. The user is logged in as 'admin'. The breadcrumb trail is 'INVENTORIES / CREATE INVENTORY'. The form has several tabs: 'DETAILS' (selected), 'PERMISSIONS', 'GROUPS', 'HOSTS', 'SOURCES', and 'COMPLETED JOBS'. The 'DETAILS' tab contains the following fields: 'NAME' (with a red asterisk) containing 'cookbook', 'DESCRIPTION', 'ORGANIZATION' (with a red asterisk) set to 'Default', 'INSIGHTS CREDENTIAL', and 'INSTANCE GROUPS'. At the bottom, there is a 'VARIABLES' section with a 'YAML' tab selected, showing a code editor with the following content:

```
1 ---
2 ansible_user: root
```

 At the bottom right of the form are 'CANCEL' and 'SAVE' buttons.

The new inventory page lets us define the name of the inventory and various parameters including variables. For example, for our cookbook inventory, we can add a variable, `ansible_user`, with the value as `root`. To add hosts and modify other tabs of the inventory, we'll have to save the inventory first.

Once we have saved the inventory, we can add a host by going to the **Hosts** tab and clicking the **+ADD HOST** button. Here, we would be able to add the IP or the hostname along with any host variables.

That's it! We have our inventory ready for use.

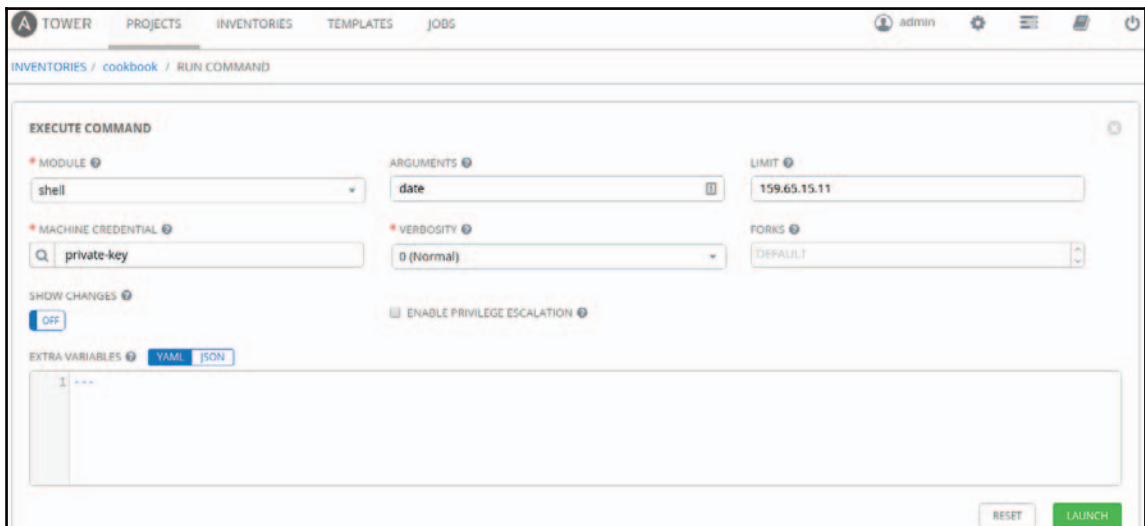
Executing ad-hoc commands

One of the simplest operations that we can do with Ansible is to execute ad-hoc commands, so let's go ahead and execute a command on the host that we added in the inventory.

How to do it...

To execute an ad-hoc command on any host, we need to do the following:

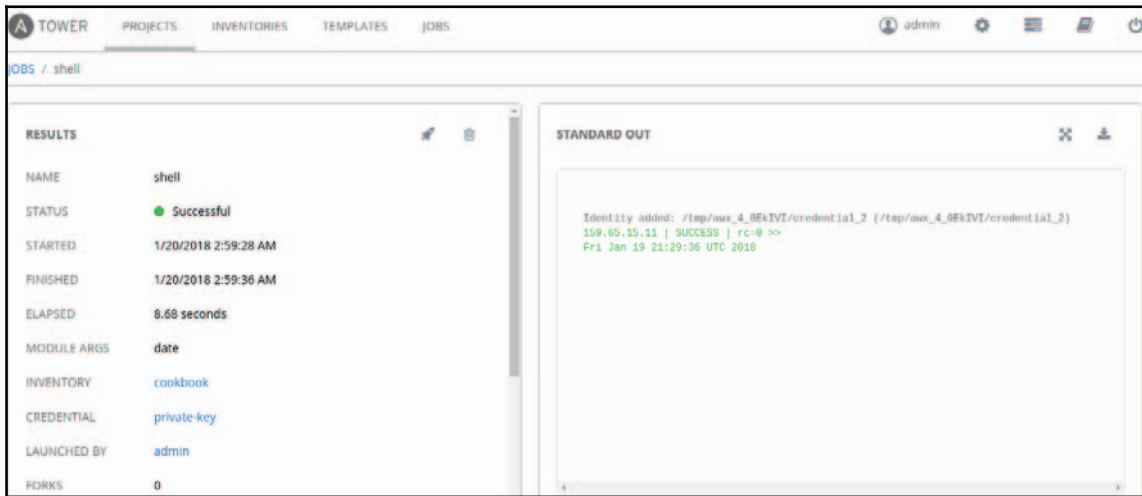
1. We need to go to the **Inventories** tab from the top navigation bar and select the **cookbook inventory**.
2. Once the inventory section opens up, we need to go to the **Hosts** tab and select the host on which we want to execute the command.
3. After selecting the host, click on the **RUN COMMANDS** button.
4. A form for executing commands will open up. We can choose the module to execute the command. For this example, we will select the shell module from the module dropdown and provide the command in the **ARGUMENTS** field.
5. We have to provide a credential so that Ansible can execute the command. We can click on the lens icon in the machine credential field and select the credential that we created in the recipe *Adding a machine credential*.



The screenshot shows the 'EXECUTE COMMAND' form in the Ansible Tower web interface. The form is titled 'EXECUTE COMMAND' and is located under the 'INVENTORIES / cookbook / RUN COMMAND' path. It contains several fields and controls:

- MODULE:** A dropdown menu with 'shell' selected.
- ARGUMENTS:** A text input field with 'date' entered.
- LIMIT:** A text input field with '159.65.15.11' entered.
- MACHINE CREDENTIAL:** A dropdown menu with 'private-key' selected.
- VERBOSITY:** A dropdown menu with '0 (Normal)' selected.
- FORKS:** A dropdown menu with 'DEFAULT' selected.
- SHOW CHANGES:** A toggle switch set to 'OFF'.
- ENABLE PRIVILEGE ESCALATION:** A checkbox that is unchecked.
- EXTRA VARIABLES:** A section with tabs for 'YAML' and 'JSON'. The 'YAML' tab is active, and the text area is empty.
- RESET** and **LAUNCH** buttons are at the bottom right.

6. To execute the command, we just have to click on the **LAUNCH** button. This creates a job. Logs of this job would be preserved, so we can refer to them at a later stage.



Using Ansible Tower with a cloud provider

So far, we have seen a few simple operations performed using Ansible Tower. Let's now start using Ansible Tower in a real-life situation.



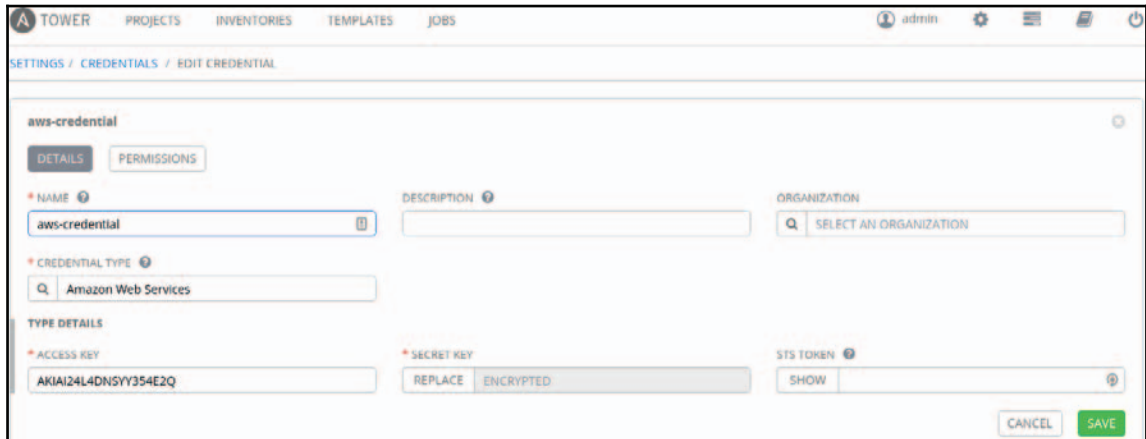
Note that our license will only let us manage up to 10 hosts. If our AWS account has more than 10 hosts, the inventory script will fail. If we want to manage more than 10 hosts, we should upgrade our license.

How to do it...

We will add an Amazon Web Services cloud to Ansible Tower and fetch the hosts present in the cloud dynamically. Let's start by creating a cloud credential.

1. We want to create a credential of the type Amazon Web Services, so go to the settings button and select the **CREDENTIALS** section.
2. In the **NEW CREDENTIAL** section, we can specify the name of the key. Here, we would set the **CREDENTIAL TYPE** as **Amazon Web Services**.

3. The form will expand so we can provide the AWS access key and secret key and save the credential.

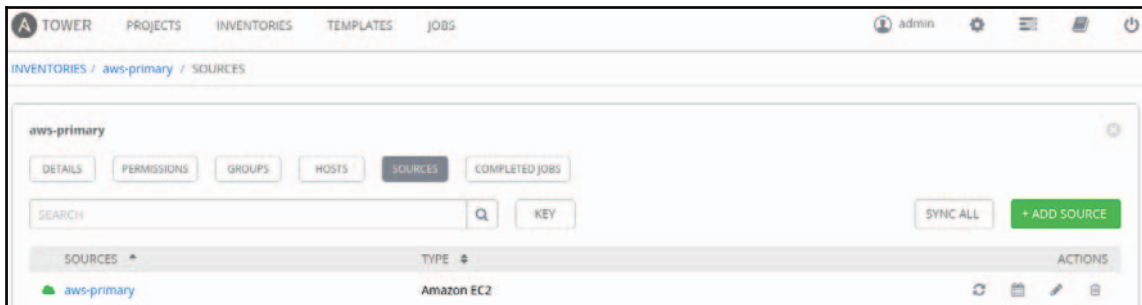


The screenshot shows the 'EDIT CREDENTIAL' form in Ansible Tower. The breadcrumb trail is 'SETTINGS / CREDENTIALS / EDIT CREDENTIAL'. The form title is 'aws-credential'. There are two tabs: 'DETAILS' (selected) and 'PERMISSIONS'. The form fields are as follows:

- NAME:** 'aws-credential' (with a copy icon).
- DESCRIPTION:** (empty).
- ORGANIZATION:** 'SELECT AN ORGANIZATION' (dropdown).
- CREDENTIAL TYPE:** 'Amazon Web Services' (dropdown).
- TYPE DETAILS:**
 - ACCESS KEY:** 'AKIAI24L4DNSY354E2Q'.
 - SECRET KEY:** 'REPLACE' and 'ENCRYPTED' buttons.
 - STS TOKEN:** 'SHOW' button and a text input field.

At the bottom right are 'CANCEL' and 'SAVE' buttons.

4. After the credential has been set up, we need to go to Inventories and add a new inventory by clicking on the **+ADD** button and selecting **Inventory** from the menu.
5. Before we can configure the inventory, we need to supply a name and save it.
6. After saving the inventory, we need to go to the **SOURCES** tab and click on the **+ADD SOURCE** button.
7. Here, we have to specify a name and set the source to **Amazon EC2** from the dropdown and add the credential using the lens button in the **CREDENTIAL** field. Save this source.
8. Lastly, from the **SOURCES** tab, click on the **SYNC ALL** button to sync the source. A green cloud icon next to the name of the source signifies that our inventory is ready to use.



Integrating Ansible roles with tower

So far, we have just executed ad-hoc commands, however, the real strength of Ansible lies in playbooks. Ansible allows us to add multiple Ansible code bases (playbooks, roles, variables, and so on). While Ansible Tower supports keeping the Ansible code in the Ansible Tower server's project directory, it is a practice that we would like to discourage. We strongly recommend using a version control system like Git for keeping Ansible code. For our example, we would use GitHub.

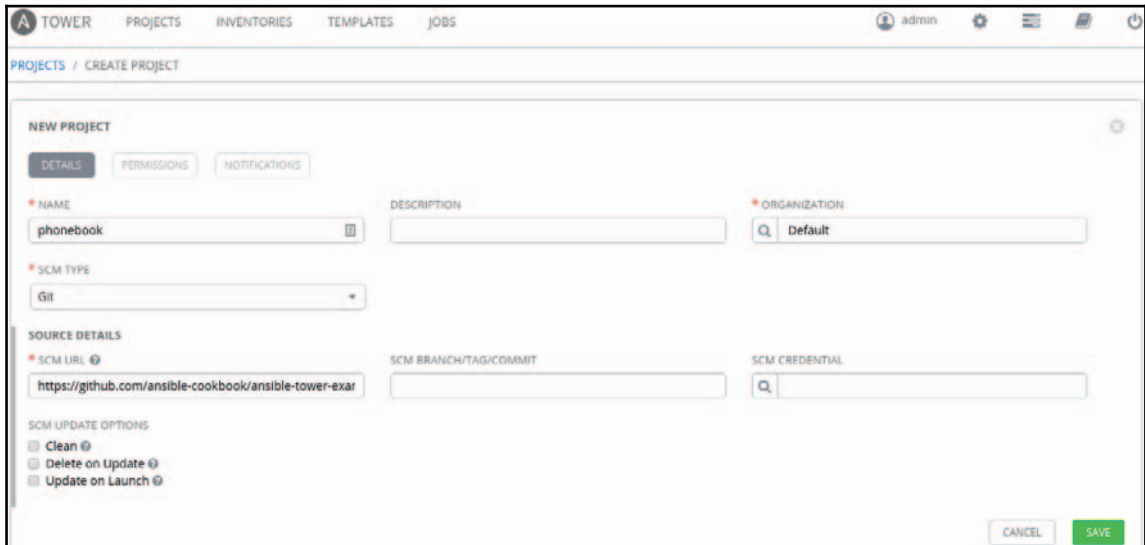
Let us integrate our GitHub repository (<https://github.com/ansible-cookbook/ansible-tower-example.git>) in Ansible Tower.

How to do it...

Executing playbooks from a Git repository involves two steps: first, we create a project, and then we create a template to execute the playbooks from the project. Let's start with creating a project:

1. Go to the **PROJECTS** tab from the top navigation bar. A **Demo Project** has already been created for our convenience.
2. Here, we need to click on the **+ADD** button. This will open a form for adding a new project.
3. We will supply a name for our project and set the **SCM TYPE** to **Git**. The form will expand so we can enter the details about Git repository.

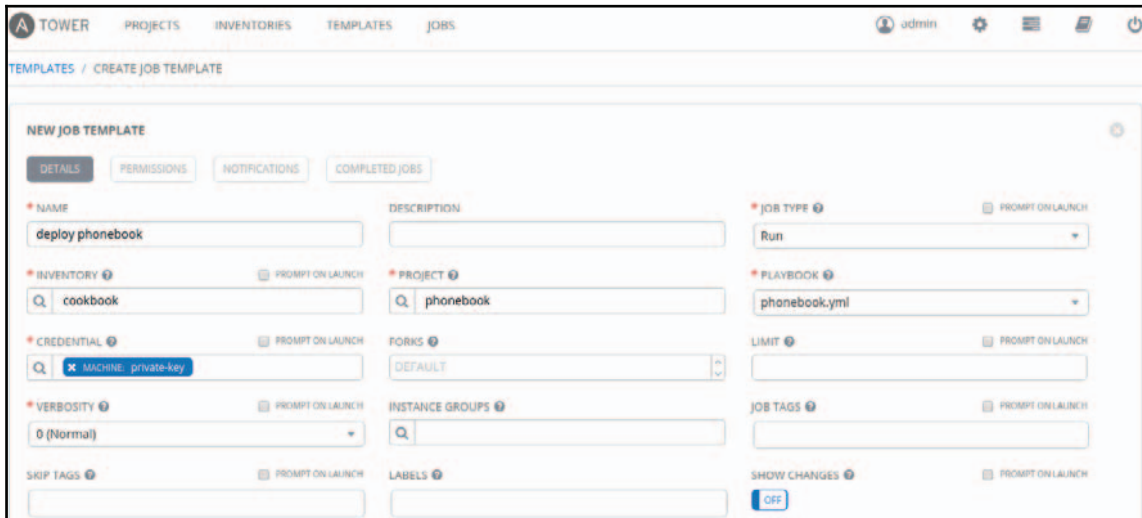
4. We will specify the GitHub URL ending in `.git` (`https://github.com/ansible-cookbook/ansible-tower-example.git`). If required, we can add a credential for cloning or pulling the code from GitHub. Save the project after adding GitHub details.



The screenshot shows the 'NEW PROJECT' form in the Ansible Tower web interface. The form is divided into several sections: 'DETAILS', 'PERMISSIONS', and 'NOTIFICATIONS'. The 'DETAILS' tab is active. It contains fields for 'NAME' (set to 'phonebook'), 'DESCRIPTION', and 'ORGANIZATION' (set to 'Default'). Below these is the 'SCM TYPE' dropdown, which is set to 'Git'. The 'SOURCE DETAILS' section includes 'SCM URL' (set to 'https://github.com/ansible-cookbook/ansible-tower-exar'), 'SCM BRANCH/TAG/COMMIT', and 'SCM CREDENTIAL'. At the bottom, there are 'SCM UPDATE OPTIONS' with checkboxes for 'Clean', 'Delete on Update', and 'Update on Launch'. The 'SAVE' button is highlighted in green at the bottom right.

5. After we click on the **SAVE** button, we can scroll down and click on the cloud icon with a downwards arrow to download an updated copy of the code. This can also be done from the **PROJECTS** page directly.
6. Once our project is ready, we should go to the **TEMPLATES** page from the top navigation bar.
7. On the **Templates** page, we need to click on the **+ADD** button and select **Job Template** from the dropdown menu.
8. Here, we have to specify a name, for example, `deploy phonebook`.
9. There are two possible values in the **JOB TYPE** dropdown. If we select the value as **Check**, then executing the job will be a dry run to make sure that the syntax of the playbook is correct; it won't make any changes on the target host. Instead, we will set the value as **Run** to execute this job and make the desired changes on the target host.
10. Now we need to set the inventory. Here, we are going to use the cookbook inventory that we created in the recipe, *Building a simple inventory*.
11. We will set the project as phonebook, which we created in *step 4*.

12. The **PLAYBOOK** dropdown will list all the playbooks found in the project that we chose. For us, there is only one playbook in our project called `phonebook.yml`, so we will select that.
13. We need to set a credential for SSH. We created a machine credential in the recipe *Adding a machine credential*. We will use that credential to authenticate with the target host.

The screenshot shows the 'NEW JOB TEMPLATE' form in the Ansible Tower web interface. The form is organized into a grid of fields. At the top, there are tabs for 'DETAILS', 'PERMISSIONS', 'NOTIFICATIONS', and 'COMPLETED JOBS'. The 'DETAILS' tab is active. The form fields include: 'NAME' (text input with 'deploy phonebook'), 'DESCRIPTION' (text input), 'JOB TYPE' (dropdown menu with 'Run'), 'INVENTORY' (dropdown menu with 'cookbook'), 'PROJECT' (dropdown menu with 'phonebook'), 'PLAYBOOK' (dropdown menu with 'phonebook.yml'), 'CREDENTIAL' (dropdown menu with 'MACHINE-private-key'), 'FORKS' (dropdown menu with 'DEFAULT'), 'LIMIT' (text input), 'VERBOSITY' (dropdown menu with '0 (Normal)'), 'INSTANCE GROUPS' (text input), 'JOB TAGS' (text input), 'SKIP TAGS' (text input), and 'LABELS' (text input). Each field has a 'PROMPT ON LAUNCH' checkbox. At the bottom right, there is a 'SHOW CHANGES' dropdown menu set to 'OFF'.

14. We should click on the **SAVE** button and go to the **Templates** page from the top navigation bar. Here, we would see a rocket icon next to our template. Clicking that would create a job and execute our task.

We should keep an eye on the **JOBS** page for the logs of the jobs that get executed.

Scheduling jobs

There are certain types of jobs that we need to execute periodically; taking a backup is one such job. Ansible allows us to schedule jobs at a set frequency or time.

How to do it...

1. We can use an existing project and template, or create a new one following the previous recipe.
2. On the **Templates** page, for the template that we want to schedule, we should click on the calendar icon.
3. A page with all the existing schedules will open. Since we are doing this for the first time, it would be empty for us. We need to click the **+ADD** button here.
4. Here, we can specify the **NAME**, **START DATE**, **START TIME**, and **LOCAL TIME ZONE**.
5. For daily execution, we can set the **REPEAT FREQUENCY** to **Day** and set the **EVERY** field to **1**. If we want a task to be executed every alternate day, then this should be set to **2**, and so on.
6. We can set an **END** to this schedule to **Never**, or after a certain number of occurrences, or even after a particular date.
7. Lastly, we should verify the schedule description to ensure that everything is as per our requirements, and then save the schedule.

The screenshot shows the configuration page for a schedule named "database daily backup". The fields are as follows:

- NAME:** database daily backup
- START DATE:** 01/20/2018
- START TIME (HH24:MM:SS):** 00:00:00
- LOCAL TIME ZONE:** UTC
- REPEAT FREQUENCY:** Day
- FREQUENCY DETAILS:**
 - EVERY:** 1 DAYS
 - END:** Never
- SCHEDULE DESCRIPTION:** every day
- OCCURRENCES (limited to first 10):** 1/20/2018 00:00:00 UTC, 1/21/2018 00:00:00 UTC, 1/22/2018 00:00:00 UTC
- DATE FORMAT:** LOCAL TIME (radio button) and UTC (radio button, selected)

Ansible Tower API

At times, we may need to invoke a certain operation on Ansible Tower programmatically. Ansible Tower provides a rich API to take care of most of the operations. Let's have a look at a few API operations using curl. All the API requests would require us to supply authentication information. Additionally, we would add `-s` for a silent output and `-k` for skipping SSL verification. The `-k` flag need not be used if the SSL has been set up correctly.

How to do it...

1. Let's first get the inventory end points:

```
$ curl -s -k -u admin:mysecretpassword  
https://192.168.0.102/api/v2/
```

2. This would output a JSON file. If it is too hard to read, we can pipe the output to the Python `json.tool` module, like this:

```
$ curl -s -k -u admin:mysecretpassword  
https://192.168.0.102/api/v2/ | python -m json.tool
```

3. We can list templates as well:

```
$ curl -s -k -u admin:mysecretpassword  
https://192.168.0.102/api/v2/job_templates/
```

4. From the output of the API response, we can get the ID of the template and execute the jobs:

```
$ curl -k -u admin:mysecretpassword  
https://192.168.0.102/api/v2/job_templates/7/launch/
```

5. We can also check out logs of a job by using a job ID obtained from the previous launch:

```
$ curl -k -u admin:mysecretpassword  
https://192.168.0.102/api/v2/jobs/28/
```

Autoscaling using Callback

So far, we have looked at methods where we have invoked the templates manually. However, in a cloud infrastructure, we often scale infrastructure dynamically. As a result, executing jobs manually can be time-consuming. Ansible Tower provides us a way so that instances can call the jobs themselves without actually using the password. The concept where an instance calls up a job is called a Callback, so let's go ahead and create a Callback URL.

How to do it...

1. We should go to the **Templates** page and click on the name of the template.
2. Here, we need to scroll down to the options and check **Allow Provisioning Callbacks**.
3. We have to provide a config key. Ideally, this should be a long string. We can also click on the wand button next to the host config key to generate a random string, and then we can click on the **SAVE** button.



OPTIONS

- ☐ Enable Privilege Escalation ?
- ☒ Allow Provisioning Callbacks ?
- ☐ Enable Concurrent Jobs ?
- ☐ Use Fact Cache ?

PROVISIONING CALLBACK URL ? HOST CONFIG KEY ?

https://192.168.0.102:443/api/v2/job_templates/9/callback 1df369954d97f6a9d0c2b2861ba14cc7

4. Any instance can call for the job using the Callback URL. Typically, this can be done through curl in `/etc/rc.local`, or by using the various mechanisms provided by cloud providers, such as `cloud-init`. The command that we should put is as follows:

```
$ curl -k --data "host_config_key=1df369954d97f6a9d0c2b2861ba14cc7"
https://192.168.0.102:443/api/v2/job_templates/7/callback/
```


Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

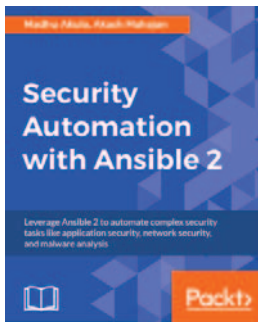


Implementing DevOps with Ansible 2

Jonathan McAllister

ISBN: 978-1-78712-053-2

- Get to the grips with the fundamentals of Ansible 2.2 and how you can benefit from leveraging Ansible for DevOps.
- Adapt the DevOps process and learn how Ansible and other tools can be used to automate it.
- Start automating Continuous Integration and Continuous Delivery tasks using Ansible
- Maximize the advantages of tools such as Docker, Jenkins, JIRA, and many more to implement the DevOps culture.
- Integrate DevOps tools with Ansible
- Extend Ansible using Python and create custom modules that integrate with unique specific technology stacks
- Connect and control the states of various third-party applications such as GIT, SVN, Artifactory, Nexus, Jira, Hipchat, Slack, Nginx, and others



Security Automation with Ansible 2

Madhu Akula, Akash Mahajan

ISBN: 978-1-78839-451-2

- Use Ansible playbooks, roles, modules, and templating to build generic, testable playbooks
- Manage Linux and Windows hosts remotely in a repeatable and predictable manner
- See how to perform security patch management, and security hardening with scheduling and automation
- Set up AWS Lambda for a serverless automated defense
- Run continuous security scans against your hosts and automatically fix and harden the gaps
- Extend Ansible to write your custom modules and use them as part of your already existing security automation programs
- Perform automation security audit checks for applications using Ansible
- Manage secrets in Ansible using Ansible Vault

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

- A DNS record
 - adding 117
- access control list (ACL) 102
- ad-hoc commands
 - executing, in Ansible Tower 162, 163
- Amazon Machine Image (AMI)
 - about 39
 - creating 39
- Amazon Web Services (AWS) 26
- Amazon Web Services
 - about 47
 - Dynamic inventory 48
 - Identity Access Management (IAM users) 48
 - Lambda 48
 - Relational Database Service (Amazon RDS) 47
 - Route53 47
 - Simple Storage Service (S3) 48
- Ansible command line
 - executing, to check connectivity 12, 13
- Ansible entities 11
- Ansible roles
 - integrating, with Ansible Tower 165, 166, 167
- Ansible Tower API 169
- Ansible Tower examples
 - reference link 165
- Ansible Tower requisites
 - reference link 156
- Ansible Tower
 - about 155
 - Ansible roles, integrating with 165, 166, 167
 - installing 156, 157
 - jobs, scheduling 167, 168
 - license, providing 157, 158
 - reference link 156
 - using, with cloud provider 163, 164

- Ansible Vault
 - secrets, managing 15, 17
- Ansible
 - about 9
 - advantages 9
 - Infrastructure as Code 10
 - installing 12
 - preparing, for working with AWS 26, 27
 - preparing, for working with Azure 85, 86, 87, 88, 89, 90, 93
 - preparing, with Docker 122, 123
 - preparing, with OpenStack 138, 139
- application
 - deploying 18, 19, 22
- auto scaling groups
 - creating 42, 45
- AWS EC2
 - auto scaling groups 42
 - launch configuration 42
 - scaling group policies 42
- Azure Blob storage
 - working 104, 105
- Azure inventory script
 - reference link 106
- Azure network security group
 - managing 102
- Azure Storage Account 104
- Azure virtual machine
 - creating 93, 94, 96
 - image 94
 - location 94
 - resource groups 94
 - storage account 94

B

- Blobs 104

block storage
managing 114

C

Callback
used, for autoscaling 170

Cinder volume
creating 144, 145

Cirros
URL 144

Classless InterDomain Routing (CIDR) 28

cloud provider
Ansible Tower, used for 163, 164
working 13

Cloud SQL instance
creating 77, 78, 79

Compose-based service
scaling up 133, 135

container
bind mounts 126
Docker Hub 123
Docker images 123
executing 123, 124
Tmpfs mounts 126
volumes 126
volumes, mounting 126, 127

D

DigitalOcean, API settings
URL 110

DigitalOcean
SSH keys, adding 111
working 110, 111

DigitalOceanq
SSH keys, adding 112

Docker Compose
used, for managing services 130, 131

Docker image
downloading 125, 126

Docker Registry
logging 129, 130
setting up 128, 129

Docker
Ansible, preparing 122, 123

doct1 tool
reference link 113

domain name system (DNS) 47
Domain Specific Language (DSL) 9

droplets
about 113
creating 112

dynamic inventory
about 151, 152
using 22, 23, 24, 58, 59, 80, 83, 106, 117, 118

E

EC2 dynamic inventory
reference link 59

EC2 instances
Amazon Machine Image (AMI) 34
attaching 40
creating 34
EC2 keypair 34
instance type 34

Elastic IPs
assigning 37
creating 37

Elastic Load Balancer (ELB)
about 40
creating 40

F

firewall rules
managing 70, 71

flavor
creating 149, 150

floating IP
attaching 115

G

GCE images
managing 72

GCE instances
creating 65, 67

Geo-Redundant Storage (GRS) 96

Google Cloud Platform
working 62, 63, 64, 65

Google Cloud Storage

- objects, managing 76, 77
- Google Compute Engine script
- reference link 80

H

- Horizon dashboard 152

I

- IAM users
 - managing 57
- Identity and Access Management (IAM) 57
- image
 - about 94
 - adding 150
 - offer 94
 - publisher 94
 - SKU 94
 - version 94
- Infrastructure as a Service (IaaS) 84
- Infrastructure as Code 10
- instance templates
 - creating 73, 74
- instance
 - tagging 69

K

- keypair
 - adding 140

L

- Lambda
 - code, packaging 54
 - handler 55
 - managing 54, 56
 - role 55
 - runtime 55
- load balancer
 - managing 71
 - using 116, 117
- Locally Redundant Storage (LRS) 96

M

- machine credential
 - adding, in Ansible Tower 159, 160

- managed instance groups
 - creating 74

N

- Network Address Translation (NAT) 29
- network interfaces
 - managing 96, 98, 99
 - public IP addresses, used 100, 101
 - virtual network 97
- network resources
 - managing 142
- network
 - managing 70, 71
- Nova compute instance
 - attaching 144, 145
 - managing 143, 144

O

- objects
 - managing, in Google Cloud Storage 76, 77
 - managing, in Swift 145, 146
- OpenStack, images
 - reference link 150
- OpenStack
 - Ansible, preparing 138, 139

P

- persistent disks
 - attaching 67, 68
- phonebook application
 - deploying 45, 46, 60, 83, 108, 119, 120, 135, 153, 154
- Platform as a Service (PaaS) 84
- playbooks
 - executing 14, 15
- public IP addresses
 - used, in network interfaces 100, 101
 - used, in virtual machine 100, 101
 - working 99, 100

Q

- qcow2 image
 - URL, for downloading 151

R

RDS instance

- creating 48, 49, 50
- DB instance 48
- DB Parameters Groups 49
- regions and availability zones 48
- security groups 48

Read Access Geo-Redundant Storage (RAGRS) 96

relational database service (RDS) 48

resource groups 94

Route53

- DNS type 50
- DNS zone 50
- records, creating 50, 51, 52
- records, deleting 50, 51, 52

S

S3 objects

- managing 52, 53, 54

security groups

- creating 32
- managing 32, 141, 142

Server Message Block (SMB) 94

simple inventory

- building, in Ansible Tower 160, 161

Simple Storage Service (S3) 52

snapshots

- creating, for backup 68

Software as a Service (SaaS) 84

SSH keys

- adding, to DigitalOcean 111, 112

Standard_RAGRS 96

storage account

about 94

Blob storage 94

file storage 94

queue storage 94

Storage Container 104

Swift

- objects, managing 145, 146

T

time to live (ttl) parameter 51

U

user management

- about 146, 147, 148
- fundamentals 146
- reference link 147

V

virtual hard disk (VHD) 94

virtual machine

- public IP addresses, used 100, 101

virtual network 97

Virtual Private Cloud (VPC)

- creating 28, 29, 30, 32
- managing 28, 29, 30, 32

volumes

- attaching, to instances 38, 39

Y

YAML

- URL 9

Z

Zone-Redundant Storage (ZRS) 96