



HashiCorp Infrastructure Automation Certification Guide

Pass the Terraform Associate exam and manage IaC to scale across AWS, Azure, and Google Cloud



HashiCorp Infrastructure Automation Certification Guide

Pass the Terraform Associate exam and manage IaC to scale across AWS, Azure, and Google Cloud

Ravi Mishra

Packt>

BIRMINGHAM—MUMBAI

HashiCorp Infrastructure Automation Certification Guide

Copyright © 2021 Packt Publishing

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

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

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

Group Product Manager: Wilson D'souza
Publishing Product Manager: Vijin Boricha
Senior Editor: Shazeen Iqbal
Content Development Editor: Romy Dias
Technical Editor: Shruthi Shetty
Copy Editor: Safis Editing
Project Coordinator: Shagun Saini
Proofreader: Safis Editing
Indexer: Subalakshmi Govindhan
Production Designer: Aparna Bhagat

First published: July 2021

Production reference: 1140721

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

ISBN 978-1-80056-597-5

www.packt.com

I would like to dedicate this book to my sweet and lovely daughter, Kaira Mishra, expecting her to be reading this book in the near future.

Contributors

About the author

Ravi Mishra (born in 1988) is a multi-cloud architect with a decade of experience in the IT industry. He started his career as a network engineer, then later, with time, he got the opportunity to work in the cloud domain, where he worked his hardest to become an expert in cloud platforms such as AWS, Azure, GCP, and Oracle. Along with the cloud, he has also developed a skillset in DevOps, containerization, Kubernetes, and Terraform. This was what inspired him to write this book.

Ravi has an electronics engineering degree, with a postgraduate diploma in IT project management.

Throughout his career, he has worked with multiple global MNCs.

He has more than 35 cloud certifications, including as a Microsoft Certified Trainer and a public speaker. You can find him on LinkedIn as [inmishrar](#).

Thanks to almighty God, who has given me the power and notion to draft this book.

Special thanks to my wife, Kavita Mishra, for her continued support and encouragement with everything that I do. I genuinely appreciate what you have done for us and I love you.

To my parents, Anil Mishra and Sarita Mishra, for their blessings throughout my life.

To my brother, Ratnesh Mishra, Hamid Raza (my Guru), and the whole Packt team, who helped me during this entire journey.

About the reviewers

Clara McKenzie is currently a support engineer for Terraform at HashiCorp with a long career in software, specializing in networking and embedded platforms. Born in San Francisco, she studied mathematics at Reed College in Portland, Oregon. Her work adventures include setting up the NFS/RPC group at Sun Microsystems, the NetWareForMac team at Novell, the Core Software Team at Ascend Communications, the Gracenote SDK Team, and creating test scenarios for Planets Dove Satellites. She goes by the name `cemckenzie` on GitHub.

Mehdi Laruelle is an automation and cloud consultant. Mehdi has had the opportunity to work for several years for major players in the industry. He has worked with DevOps culture and tools leading him to use HashiCorp software such as Terraform, Vault, and Packer. Knowledge is a passion that he likes to share, whether through training, articles, or meetups. He is the co-organizer of the HashiCorp User Group France meetup. This has led to him being recognized as a HashiCorp Ambassador and **AWS Authorized Instructor (AAI)**.

Table of Contents

Preface

Section 1: The Basics

1

Getting to Know IaC

Technical requirements	4	A comparison with other IaC	15
Introduction to IaC	4	CloudFormation versus Terraform	16
Advantages of IaC	5	Azure ARM templates versus Terraform	20
Simple and speedy	5	Google Cloud Deployment Manager versus Terraform	23
Configuration consistency	5		
Risk minimization	6	An understanding of Terraform architecture	27
Increased efficiency in software development	6	Terraform Core	28
Cost savings	7	Terraform plugins	28
Introduction to Terraform	7	Summary	31
What is Terraform?	7	Questions	31
Features of Terraform	8	Further reading	32
Terraform use cases	9		

2

Terraform Installation Guide

Technical requirements	34	Installing Terraform on Linux	39
Installing Terraform on Windows	34	Downloading Terraform	39
Downloading Terraform	34		

Installing Terraform on macOS	42	Summary	45
Downloading Terraform	42	Questions	45
		Further reading	46

Section 2: Core Concepts

3

Getting Started with Terraform

Technical requirements	50	Understanding Terraform output	70
Introducing Terraform providers	50	Terraform output	70
Terraform providers	50	Understanding Terraform data	79
Knowing about Terraform resources	60	Terraform data sources	79
Terraform resources	60	Summary	82
Understanding Terraform variables	63	Questions	82
Terraform variables	64	Further reading	84

4

Deep Dive into Terraform

Technical requirements	86	Understanding Terraform loops	105
Introducing the Terraform backend	86	The count expression	106
Terraform state	86	The for_each expression	113
The purpose of the Terraform state file	87	The for expression	119
Terraform backend types	88	Understanding Terraform functions	121
Understanding Terraform provisioners	96	Understanding Terraform debugging	123
Terraform provisioner use cases	97	Summary	125
Terraform provisioner types	99	Questions	126
		Further reading	127

5

Terraform CLI

Technical requirements	130	Terraform	146
Introduction to the Terraform CLI	130	Integrating with GCP	148
Integrating with Azure	133	Authentication using a Google service account by storing credentials in a separate file	148
Authentication using a Service Principal and a Client Secret	134	Provisioning GCP services using Terraform	152
Provisioning Azure services using Terraform	140	Understanding the Terraform CLI commands	154
Integrating with AWS	144	Summary	159
Authentication using an access key ID and secret	144	Questions	159
Provisioning AWS services using		Further reading	160

6

Terraform Workflows

Technical requirements	162	Terraform destroy	177
Understanding the Terraform life cycle	162	Understanding Terraform workflows using Azure DevOps	180
Terraform init	163	Summary	190
Terraform validate	167	Questions	190
Terraform plan	169	Further reading	192
Terraform apply	175		

7

Terraform Modules

Technical requirements	194	Writing Terraform modules for Azure	205
Understanding Terraform modules	194	Writing Terraform modules for AWS	214
source	196		
version	201		

Writing Terraform modules for GCP	220	Publishing a module	226
Publishing Terraform modules	225	Summary	227
Key requirements	225	Questions	228
		Further reading	229

Section 3: Managing Infrastructure with Terraform

8

Terraform Configuration Files

Technical requirements	234	Writing Terraform configuration files for GCP	244
Understanding Terraform configuration files	234	Writing Terraform configuration files for AWS	247
Terraform native configuration syntax	234	Writing Terraform configuration files for Azure	250
Terraform override file	236	Summary	254
JSON configuration syntax	239	Questions	255
Data types	241	Further reading	256
Terraform style conventions	242		

9

Understanding Terraform Stacks

Technical requirements	258	Writing Terraform stacks for AWS	269
Understanding Terraform stacks	258	Writing Terraform stacks for Azure	273
Writing Terraform stacks for GCP	259	Summary	276
		Questions	277
		Further reading	278

10

Terraform Cloud and Terraform Enterprise

Technical requirements	280	Overviewing Terraform	
Introducing Terraform Cloud	280	Sentinel	293
Terraform Cloud workflow	282	Comparing different Terraform features	300
Understanding Terraform Enterprise	292	Summary	301
		Questions	302
		Further reading	303

11

Terraform Glossary

Assessments

Other Books You May Enjoy

Index

Preface

Terraform is in huge demand in the IT market. There is a serious question about managing the IT infrastructure in code format. So, Terraform is an option for users that can help to build, configure, and manage the infrastructure.

This book provides you with a walkthrough from the very basics of Terraform to an industry expert level. It is a comprehensive guide that starts with an explanation of **Infrastructure as Code (IaC)**, mainly covering what Terraform is and what the advantages of using Terraform are. Moving further, you will be able to set up Terraform locally in your system. In the next phase, you will get a thorough understanding of the different blocks, such as providers, resources, variables, output, and data, used in Terraform configuration code.

This book provides details of the Terraform backend, provisioners, loops, and inbuilt functions, and how to perform debugging with Terraform. In this book, you will learn how to integrate Terraform with Azure, AWS, and GCP. Along with integration, you will also be able to write Terraform configuration files covering all the three major clouds, which make up the best part of this book, so that you get a proper understanding of different use cases in all the major clouds, that is, Azure, AWS, and GCP. We also explain the complete Terraform life cycle, covering `init`, `plan`, `apply`, and `destroy`, which will help you to get an understanding of the Terraform workflow with CI/CD tools.

In the expert section of this book, we will discuss Terraform stacks and modules. In this section, you will understand how effectively you can write modules and combine those modules to build a complex stack covering all three clouds – Azure, AWS, and GCP. This helps users to manage huge enterprise-level infrastructure easily and effectively.

This book also covers Terraform products such as Terraform Enterprise and Terraform Cloud. You will get an understanding of the different features of Terraform Enterprise and Terraform Cloud. You will also get a great understanding of Sentinel, which is policy as code that can be implemented to ensure infrastructure is getting provisioned as per the necessary compliance before infrastructure actually gets provisioned into cloud providers such as Azure.

Overall, this book provides great learning for you about Terraform and using Terraform and how effectively you can use it to build and manage your enterprise-level infrastructure. Not only this, but with this book you can also get yourself prepared for the Terraform Associate exam and easily crack the exam.

Who this book is for

This book is designed for those who are planning to take the Terraform Associate exam. It is good for developers, administrators, and architects who are keen to learn about IaC, that is, Terraform. This doesn't mean that it restricts others from learning. Anyone who wishes to explore and learn about the HashiCorp product Terraform can read this book and get great learning from it.

What this book covers

Chapter 1, Getting to Know IaC, looks at IaC, which is basically a way of writing infrastructure in a code format so that the whole deployment of the infrastructure, its updating, and its manageability can be performed easily. Terraform is an IaC product from HashiCorp.

This chapter will cover topics that include the definition of IaC and its benefits. This chapter will also include an introduction to Terraform and a comparison with other IaC options such as AWS CloudFormation and Azure ARM templates. We will also discuss Terraform architecture and the workings of Terraform, which will help you to get a thorough understanding of how Terraform is used.

Chapter 2, Terraform Installation Guide, covers Terraform installation; before we start working with Terraform and get our hands dirty by writing Terraform configuration code, we need to have `terraform.exe` files installed on our system.

This chapter is one of the foundational pillars for learning about Terraform. It will cover how a user can install `terraform.exe` on their local machine, whether it is a macOS, Linux, or Windows system. Not only this, but it will also provide information on how you can validate the presence of Terraform in your local system.

Chapter 3, Getting Started with Terraform, discusses how writing Terraform configuration code with proper syntax is very important. This chapter consists of the core concepts of Terraform. When we need to write Terraform configuration code, we need to define it with some building blocks such as resources, data, variables, outputs, and providers. In this chapter, we will be discussing the different blocks used in Terraform configuration code and how a user can define and use them in the real world.

Chapter 4, Deep Dive into Terraform, looks at how there are many things that need to be taken care of, such as the state file of Terraform. The state file may hold some sort of confidential data of your infrastructure, so it needs to be stored securely.

This chapter will cover topics including use cases of the Terraform backend and how it can be used for storing the state file. In this chapter, we are also going to talk about Terraform provisioners and their use cases.

Like other programming languages, Terraform also supports different types of loops that can be used while writing the Terraform configuration file. In this chapter, we will also talk about some important topics such as different loops supported by Terraform and Terraform inbuilt functions that help to convert respective values into the required format. Along with this, we will explain how you can perform debugging in Terraform.

Chapter 5, Terraform CLI, looks at how if you wish to deploy infrastructure in Azure, AWS, or GCP using Terraform, then how you can provision them. To do that, we need to get our Terraform CLI authenticated to respective clouds.

This chapter will talk about different authentication methods used by the Terraform CLI for cloud providers such as Azure, AWS, and GCP. Not only this, but it will also cover different Terraform CLI commands and their uses.

Chapter 6, Terraform Workflows, covers how it is important for us to understand how Terraform is used to perform workflows and how it is used to manage its life cycle. We also need to understand the importance of the Terraform life cycle.

This chapter will cover the core workflow of Terraform, which includes creating a Terraform configuration file (Write), previewing the changes (Plan), and then finally committing those changes to the target environment (Apply). Once we are done with the creation of the resources, we might be required to get rid of those infrastructures (Destroy). In a nutshell, we plan to cover Terraform core workflows that mainly consist of `init`, `plan`, `apply`, and `destroy` and the respective subcommands and their outputs. This chapter also covers how we can use CI/CD tools such as Azure DevOps with Terraform.

Chapter 7, Terraform Modules, discusses how managing a large infrastructure is a challenging task for the administrator. So, we need to have some solution so that it could be easy for the administrator to manage the whole infrastructure using Terraform. So, for better understanding, in this chapter, we will cover how you can create a module and reuse that module while drafting Terraform configuration files.

Chapter 8, Terraform Configuration Files, explains that when writing Terraform configuration files, following the correct syntax and industry best practices is very important. This chapter will cover different types of configuration files, that is, JSON files and HCL files. We are also going to talk about what industry best practices can be followed while writing a Terraform configuration file for major cloud providers such as AWS, Azure, and GCP. We will discuss how you can use different blocks such as resources, data sources, locals, variables, and modules in the Terraform configuration file.

Chapter 9, Understanding Terraform Stacks, looks at how sometimes it is required to deploy a very large enterprise-level infrastructure, and writing Terraform configuration code for it would be very lengthy, so we need to think about how we can shorten the length of the code and make it reusable.

In this chapter, we are going to cover Terraform stacks, which are nothing but a collection of modules. We are also going to discuss some best practices of preparing stacks and modules for cloud providers such as AWS, GCP, and Azure.

Chapter 10, Terraform Cloud and Terraform Enterprise, covers different products of Terraform, that is, Terraform Cloud and Terraform Enterprise. We will discuss different source control, such as GitHub, that can be integrated with Terraform Cloud to get Terraform configuration files. We will also discuss Terraform Sentinel, that is, policy as code. Sometimes it is essential to ensure that our infrastructure is provisioned as per compliance and Terraform Sentinel features that are available in most enterprise products of HashiCorp, such as Vault Enterprise, Nomad Enterprise, Consul Enterprise, Terraform Cloud, and Terraform Enterprise, to help us to set up a policy to check and validate before the actual deployment of infrastructure happens. Furthermore, we will see the different features that are present in Terraform Cloud and Enterprise as compared to the Terraform CLI.

Chapter 11, Terraform Glossary, is the most interesting chapter of the entire book. Almost everyone wants to have a quick revision of the keywords used in the book. So, this chapter will talk about different Terraform acronyms used in the book.

To get the most out of this book

It is always good for you to have a basic understanding of different cloud services, such as AWS, GCP, and Azure. Along with this, you should have some knowledge about DevOps tools such as Azure DevOps and GitHub. Additionally, some experience with writing scripting such as PowerShell or Bash would be beneficial. However, the code samples in this book have been written using the HashiCorp Configuration Language.

Software/hardware covered in the book	OS requirements
The Azure, GCP, and AWS clouds	Windows, macOS, or Linux (Ubuntu)
Azure DevOps	Windows, macOS, or Linux (Ubuntu)
GitHub	Windows, macOS, or Linux (Ubuntu)
Terraform v1.0	Windows, macOS, or Linux (Ubuntu)
VSCode (editor)	Windows, macOS, or Linux (Ubuntu)
The Git Bash CLI	Windows, macOS, or Linux (Ubuntu)

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

Irrespective of whether you are sitting the HashiCorp Infrastructure Automation Terraform Associate Certification exam, it is recommended to attempt the assessment questions after completing all the chapters. This will be a good way to assess the knowledge absorbed from the content of the book.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide>. 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!

Code in Action

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

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/9781800565975_ColorImages.pdf.

Conventions used

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

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "This step is skipped if `terraform init` is run with the `-plugin-dir=<PATH>` or `-get-plugins=false` options."

A block of code is set as follows:

```
# Configure the Microsoft Azure provider
provider "azurerm" {
  version           = "=2.20.0"
  features {}
}
```

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

```
subscription_id = "...."
client_id      = "...."
client_secret    = "...."
tenant_id       = "...."
}
```

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

```
mkdir terraform && cd terraform
```

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: "Now you need to set the environment variable path for `Terraform.exe`, and to do that, go to **This PC**, right-click on it, and go to **Properties | Advance system settings | Environment Variables**."

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Share Your Thoughts

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

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

Section 1: The Basics

Section 1 aims to introduce you to the basics of **Infrastructure as Code (IaC)**, comparing other IaC options such as ARM templates and AWS CloudFormation to Terraform and how Terraform can be used to provision infrastructure. You will also get an understanding of how Terraform can be set up on a local system.

The following chapters will be covered under this section:

- *Chapter 1, Getting to Know IaC*
- *Chapter 2, Terraform Installation Guide*

1

Getting to Know IaC

In this chapter, we are going to discuss what **Infrastructure as Code (IaC)** is in detail. We will be discussing the benefits of using IaC. Furthermore, we will start describing the basics of **Terraform**, and then undertake a comparison of Terraform with other available IaC options, including **AWS CloudFormation**, **Azure Resource Manager (ARM)**, and **Google Cloud Deployment Manager**. Moving on, we will then discuss **Terraform architecture**.

Throughout this chapter, we will be focusing on Terraform for major cloud providers, including **GCP**, **AWS**, and **Azure**. The whole chapter will help you in terms of achieving a fair understanding of Terraform (IaC) and how you can overcome the old lego system of executing manual changes to your environment. This will help you get an idea of how you can start using Terraform for infrastructure automation in your organization.

The following topics will be covered in this chapter:

- Introduction to IaC
- Advantages of IaC
- Introduction to Terraform
- A comparison with other IaC
- An understanding of Terraform architecture

Technical requirements

To follow along with this chapter, you need to have a basic knowledge of different IaC options, including ARM templates and **AWS CloudFormation**, while a basic knowledge of major cloud providers, such as GCP, AWS, and Azure, will be beneficial.

Introduction to IaC

Before learning about Terraform, let's try to understand what it basically means for us and how it helps make users' lives easier in the IT industry. The first thing that always comes to consumers' minds is that when they need an IT infrastructure, for example, if they want a virtual machine, they need to raise a ticket on some ticket portal such as ServiceNow, and someone from the backend would log in to that ticketing portal and take that ticket from the queue and deploy the virtual machine for the consumer, either in VMware or a HyperV environment through the management portal using some click jobs. That is the traditional approach for infrastructure deployment, which is somewhat fine if they need to manage infrastructure in their private data center and there is very little possibility of performing scaling of those deployed resources, which means once it gets provisioned, after that no one is requesting further changes to the created resource.

In all these cases, it is fine if they easily go ahead and perform all the operations manually but what about if they need to deploy a very large infrastructure consisting of more repeatable work in the cloud? Then it would be a really tedious job for the administrator to provision those resources manually and also it is a very time-consuming job for them. So, to overcome this challenging situation, most cloud vendors have come up with an approach of IaC, which is mostly an API-driven approach. Most cloud vendors have published APIs for all their resources. Using that API, we can easily get the resource deployed in the cloud.

Nowadays, as most customers are moving toward the cloud, and as we all know, cloud platforms provide us with more elasticity and scalability in terms of their infrastructure, this means you can easily utilize the resources and pay for what you use; you don't need to pay anything extra. Just think down the line of an administrator needing to perform the scaling up and down of resources and how difficult it would be for them. Let's suppose there are 1,000 resources that need to be scaled up during the day and scaled down at night.

In this case, consumers need to raise 1,000 tickets for performing the scale-up and again 1,000 more tickets for scaling down, which means by the end of the day, the system administrator who is managing the infrastructure will get flooded with so many requests and it would be really impossible for them to handle this. So, here we have something called IaC, which is a way of deploying or managing the infrastructure in an automated way. All the resources that need to be managed will be defined in code format and we can keep that code in any source control repository, such as GitHub or Bitbucket. Later, we can apply a DevOps approach to manage our infrastructure easily. There are many advantages of using IaC; we are going to discuss a few of them.

Advantages of IaC

Let's discuss a few of the advantages of using IaC.

Simple and speedy

Using IaC, you would be able to spin up a complete infrastructure architecture by simply running a script.

Suppose you need to deploy infrastructure in multiple environments, such as development, test, preproduction, and production. It would be very easy for you to provision it with just a single click. Not only this, but say you need to deploy the same sort of infrastructure environments in other regions where your cloud provider supports backup and disaster recovery.

You can do all this by writing simple lines of code.

Configuration consistency

The classical approach of infrastructure deployment is very ugly because the infrastructure is provisioned and managed using a manual approach that helps to maintain some consistency in the infrastructure deployment process. But it always introduces human error, which makes it difficult to perform any sort of debugging.

IaC standardizes the process of building and managing the infrastructure so that the possibility of any errors or deviations is reduced. Definitely, this will decrease any incompatibility issues with the infrastructure and it will help you to run your application smoothly.

Risk minimization

When we were managing infrastructure manually, it was observed that only a handful of **Subject Matter Experts (SMEs)** knew how to do it and the rest of the team members remained blank, which introduces dependency and security risks for the organization. Just think of a situation where the person who is responsible for managing the complete infrastructure leaves the organization; that means whatever they knew they might not have shared with others or they may not have updated the documents. At the end of the day, risk has been introduced to the organization because that employee is leaving. Many times, in such cases, the organization needs to undergo some reverse engineering to fix any issues.

This challenging situation can be controlled by using IaC for the infrastructure. IaC will not only automate the process, but it also serves as a form of documentation for your infrastructure and provides insurance to the company in cases where employees leave the company with institutional knowledge. As you know, we generally keep IaC to source control tools such as Azure Repos or GitHub. So, if anyone makes any configuration changes to the infrastructure, they will get recorded in the source control repository. So, if anyone leaves or goes on vacation, it won't impact the manageability of the infrastructure because the version control tool will have kept track of the changes that have been performed on the infrastructure and this would definitely reduce the risk of failure.

Increased efficiency in software development

Nowadays, with the involvement of IaC for infrastructure provisioning and managing, developers get more time to focus on productivity. Whenever they need to launch their sandbox environments to develop their code, they are easily able to do so. The **quality analyst (QA)** will be able to have a copy of the code and test it in separate environments. Similarly, security and user acceptance testing can also be done in different staging environments. With a single click, both the application code and infrastructure code can be done together following **Continuous Integration and Continuous Deployment (CI/CD)** techniques.

Even if we want to get rid of any infrastructure, we can include an IaC script that will spin down the environments when they're not in use. This will shut down all the resources that were created by the script. So, we won't end up performing a cleanup of the orphan resources that are not in use. All this would help to increase the efficiency of the engineering team.

Cost savings

IaC would definitely help to save costs for the company. As we mentioned earlier, IaC script can help us create and automate the infrastructure deployment process, which allows engineers to stop doing manual work and start spending more time in performing more value-added tasks for the company, and because of this, the company can save money in terms of the hiring process and engineers' salaries.

As we mentioned earlier, IaC script can automatically tear down environments when they're not in use, which will further save companies cloud computing costs.

After getting a fair understanding of IaC and the benefits of using it, let's move ahead and try to learn some details about one IaC technology – Terraform.

Introduction to Terraform

Welcome to this introductory guide to Terraform. For anyone who is new to Terraform and unaware of what it is, as well as for the purpose of comparison with other IaC tools that are currently associated with major cloud providers including AWS, Azure, and Google, we believe that this is the best guide to begin with. In this guide, we will be focusing on what Terraform is and what problems it can solve for you, undertaking a comparison with other software tools, including ARM templates, AWS CloudFormation, and Google Cloud Deployment Manager, and explaining how you can start using Terraform effectively in your day-to-day jobs related to the provisioning and maintenance of your IT infrastructure.

What is Terraform?

Terraform is one of the open source tools that was introduced to the market by HashiCorp in 2014 as IaC software (IaC means we can write code for our infrastructure) that is mainly used for building, changing, and managing infrastructure safely and efficiently. Terraform can help with multi-cloud environments by having a single workflow, in other words, `terraform init`, `terraform plan`, `terraform apply`, and so on, for all clouds. The infrastructure that Terraform manages can be hosted on public clouds such as AWS, Microsoft Azure, and GCP, or on-premises in private clouds such as VMware vSphere, OpenStack, or CloudStack. Terraform handles IaC, so you never have to worry about your infrastructure drifting away from its desired configuration.

Terraform mainly uses Terraform files ending with `.tf` or `.tf.json` that hold detailed information about what infrastructure components are required in order to run a single application or your entire data center. Terraform generates an execution plan, which describes what it is going to do to reach the desired state, and then executes it to build the infrastructure described. If there is any change in the configuration file, Terraform is able to determine what has been changed and create incremental execution plans that can be applied.

Terraform can not only manage *low-level* components, such as **compute instances**, **storage**, and **networking**; it can also support high-level components, such as **DNS** and **SaaS features**, provided that the resource API is available from the providers.

After learning about what Terraform is, you might have one more question in your mind: what exactly makes this Terraform so popular? To answer that question, first and foremost, Terraform is cloud-agnostic, which means you can provision or manage your infrastructure in any cloud platform. The second thing that makes Terraform very much in demand is its standard workflow. You don't need to remember *N* number of parts of the workflow; a simple `init`, `plan`, and `apply` from Terraform's point of view would be enough and it is the same across any platform. The third factor is the Terraform syntaxing. Terraform uses uniform code syntaxing whether you work on any cloud or on-premises. There are many more exceptional factors that could encourage enterprise customers to start using Terraform.

Features of Terraform

Let's now try to get an understanding of all of Terraform's features, which are pushing up market demand for the product.

Infrastructure as code

Infrastructure is defined in a code format based on proper syntax in a configuration file that can be shared and reused. The code defined in the configuration file will provide a blueprint of your data center or the resource that you are planning to deploy. You should be able to deploy a complete infrastructure from that configuration file following the Terraform workflow.

Execution plans

The Terraform workflow has three steps – `init`, `plan`, and `apply`. During the *planning* step, it generates an execution plan. The execution plan gives you information about what Terraform will do when you call `apply`. This means you do not get any sort of surprise when you perform `terraform apply`.

Note

We are going to cover in detail the Terraform workflow in the upcoming chapters. So, stay tuned.

By using Terraform, entire things that are required for a Heroku application setup could be codified in a configuration file, thereby ensuring that all the required add-ons are available, and the best part of this is that with the help of Terraform, all of this can be achieved in just 60 seconds. Any changes requested by the developer in the Heroku app can be actioned immediately using Terraform, whether it be a complex task related to configuring *DNS* to set a *CNAME* or setting up Cloudflare as a **content delivery network (CDN)** for the app, and so on and so forth.

Multi-tier applications

N-tier architecture deployment is quite common across the industry when thinking about the required infrastructure for an application. Generally, two-tier architecture is more in demand. This is a pool of web servers and a database tier. As per the application requirements, additional tiers can be added for API servers, caching servers, routing meshes, and so on. This pattern is used because each tier can be scaled independently and without disturbing other tiers:

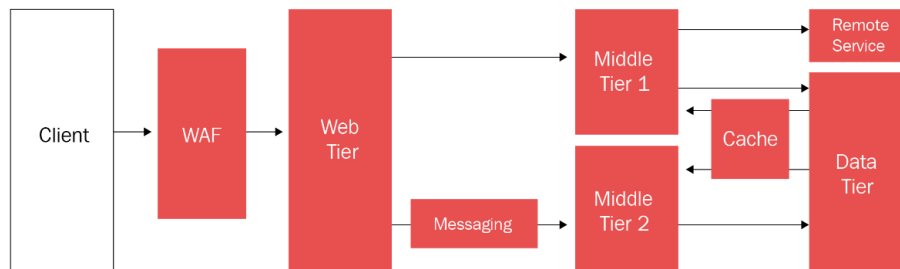


Figure 1.2 – N-tier application architecture

Now, let's try to understand how Terraform can support us in achieving *N-tier* application infrastructure deployment. In the Terraform configuration file, each tier can be described as a collection of resources, and the *dependencies* between the resources for each tier can either be implicit or we can define them explicitly so that we can easily control the sequence of the resource deployment. This helps us to manage each tier separately without disturbing the others.

Self-service clusters

In a large organization, it's quite challenging for the central operation team to provide infrastructure to the product team as and when needed. The product team should be able to create and maintain their infrastructure using tooling provided by the central operations team:

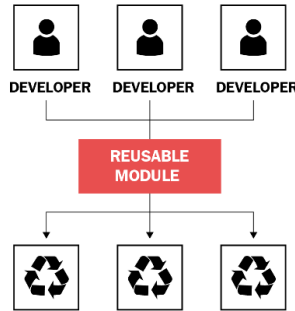


Figure 1.3 – Self-service cluster

In the preceding requirement, the entire infrastructure can be codified using Terraform, which will focus on building and scaling the infrastructure, and a Terraform configuration file can be shared within an organization, enabling product teams to use the configuration as a black box and use Terraform as a tool to manage their services. During deployment of the infrastructure, if the product team encounters any issues, they can reach out to the central operations team for help and support.

Software demos

Nowadays, software development is increasing by the day, and it is very difficult to get the infrastructure required to test that software. We have tools such as **Vagrant** at our disposal to help us build virtualized environments, and while you may be able to use that environment for demonstration purposes, it is really difficult to perform software demos directly on production infrastructure:

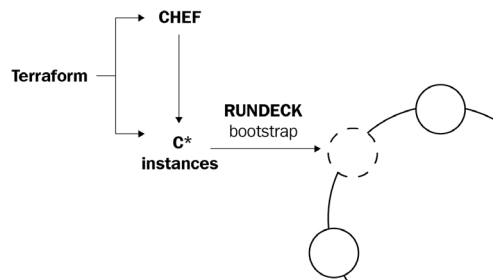


Figure 1.4 – Software demo example

A software developer can provide a Terraform configuration to create, provision, and bootstrap a demo on cloud providers such as Azure, GCP, and AWS. This allows end users to easily demo the software on their infrastructure, and it even allows them to perform scale-in or scale-out of the infrastructure.

Disposable environments

In the industry, it is quite common to have multiple landscapes, including production, staging, or development environments. These environments are generally designed as a subset of the production environment, so as and when any application needs to be deployed and tested, it can easily be done in the smaller environment; but the problem with the increase in complexity of the infrastructure is that it's very difficult to manage it:

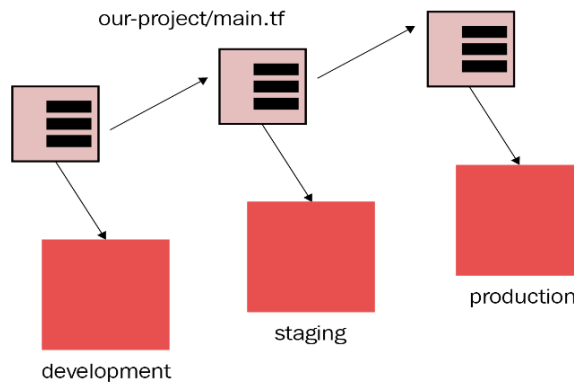


Figure 1.5 – Multiple environments

Using Terraform, the production environment that you constructed can be written in a code format, and then it can be shared with other environments, such as staging, QA, or dev. This configuration code can be used to spin up any new environments to perform testing, and can then be easily removed when you are done testing. Terraform can help to maintain a parallel environment and it can provide an option in terms of its scalability.

Software-defined networking

Software-Defined Networking (SDN) is quite famous in data centers, as it allows operators to operate a software-defined network very smoothly and developers are able to develop their applications, which can easily be run on top of the network infrastructure provided. The control layer and infrastructure layer are the two main components for defining a software-defined network:

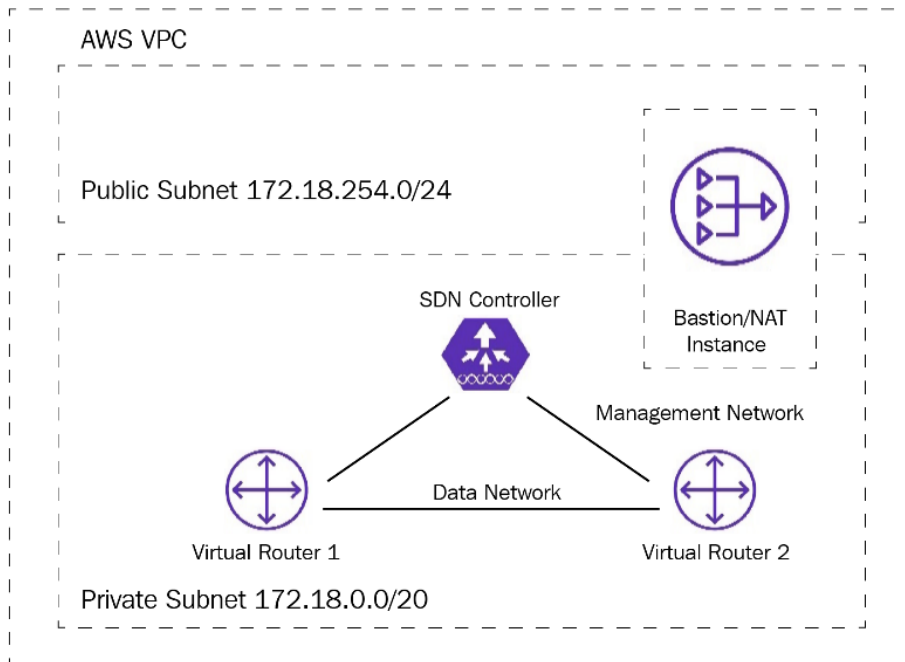


Figure 1.6 – Software-defined network

Software-defined networks can be transformed into code using Terraform. The configuration code written in Terraform can automatically set up and modify settings by interfacing with the control layer. This allows the configuration to be versioned and changes to be automated. As an example, Azure Virtual Network is one of the most commonly used SDN implementations and can be configured by Terraform.

Resource schedulers

In large-scale infrastructures, the static assignment of applications to machines is very challenging. In terms of Terraform, there are many schedulers available, such as **Borg**, **Mesos**, **YARN**, and **Kubernetes**, that can be used to overcome this problem. These can be used to dynamically schedule **Docker containers**, **Hadoop**, **Spark**, and many other software tools:

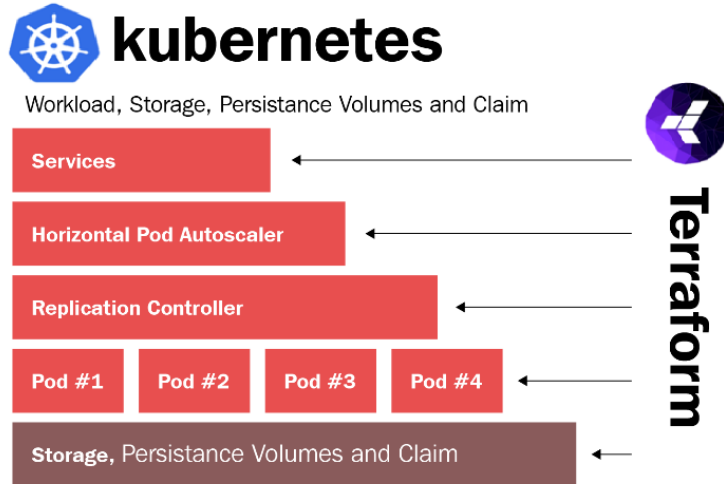


Figure 1.7 – Kubernetes with Terraform

Terraform is not just limited to cloud providers such as Azure, GCP, and AWS. Resource schedulers can also behave as providers, enabling Terraform to request resources from them. This allows Terraform to be used in layers, to set up the physical infrastructure running the schedulers, as well as provisioning them on the scheduled grid. There is a Kubernetes provider that can be configured using Terraform to schedule any Pod deployment. You can read about Kubernetes with Terraform at <https://learn.hashicorp.com/collections/terraform/kubernetes>.

Multi-cloud deployment

Nowadays, every organization is moving toward *multi-cloud*, and one of the challenging tasks is to deploy the entire infrastructure in a different cloud. Every cloud provider has its own defined manner of deployment, such as ARM templates for Azure or AWS CloudFormation. Hence, it is very difficult for an administrator to learn about all of these while maintaining the complexity of the environment deployment:

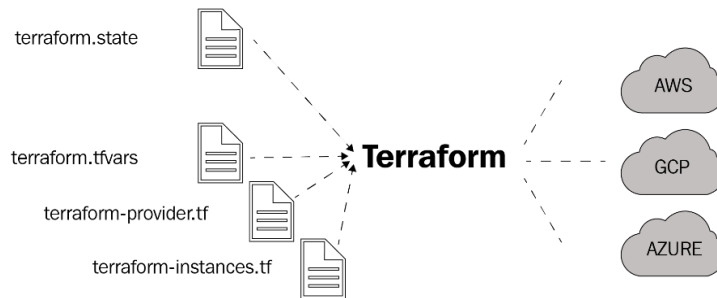


Figure 1.8 – Multi-cloud deployment

Realizing the complexity of multi-cloud infrastructure deployments using already-existing tools that are very specific to each cloud provider, HashiCorp came up with an approach known as Terraform. Terraform is cloud-agnostic. A single configuration can be used to manage multiple providers, and it can even handle cross-cloud dependencies. This simplifies management and orchestration, helping administrators to handle large-scale, multi-cloud infrastructures.

So far, we have covered IaC, namely, Terraform, its features, and the different use case scenarios where we can apply Terraform. Furthermore, we have covered how Terraform differs from other IaCs mainly used in the major cloud providers, including AWS, Azure, and Google.

A comparison with other IaC

In the preceding section, you got to know about Terraform, and similar to Terraform, there are many other IaC options that are more specific to individual cloud providers, such as **ARM templates** for the Azure cloud, **CloudFormation** for AWS, and **Cloud Deployment Manager** for GCP. Likewise, each provider has come up with their own IaC that is used for their infrastructure provisioning. Now, the challenge for an administrator or developer is to learn and remember a vast number of different template syntaxes. To overcome this challenge, Terraform came up with a solution involving common workflows and syntax that allows operators to operate complex infrastructure. Let's try to understand how Terraform is different in terms of being cross-platform, as well as its modularity, language of code, validation, readability, maintainability, workflow, error management, state management, and so on for major cloud providers including AWS, Azure, and GCP.

CloudFormation versus Terraform

In this section, we will see how Terraform differs from CloudFormation based on various parameters:



Figure 1.9 – CloudFormation versus Terraform

Let's get started.

Cross-platform

A major benefit of Terraform is that you can use it with a variety of cloud providers, including AWS, GCP, and Azure. CloudFormation templates are more centric to AWS. So here, Terraform would be more preferred than AWS CloudFormation.

Language

Terraform is written in **HashiCorp Configuration Language (HCL)**. HCL's syntax is very powerful and allows you to reference variables, attributes, and so on, whereas CloudFormation uses either JSON or YAML, which are bare notation languages and are not as powerful as HCL. With CloudFormation, you can also reference parameters, such as other stacks, but overall, we think HCL makes Terraform more productive.

On the other hand, CloudFormation has good support for various conditional functionalities. Terraform has the `count`, `for_each`, and `for` loops, which make certain things easy (such as creating N identical resources) and certain things a bit harder (such as the `if-else` type conditional structure; for example, `count = var.create_eip == true ? 1 : 0`). CloudFormation has also a `wait` condition and `creation policy`, which can be important in certain deployment situations where you may have to wait before a certain condition is satisfied.

In terms of language, Terraform is simple and easy to draft, which would encourage developers and administrators to start exploring Terraform, but it practically depends upon the use case scenario, which might in some cases require you to use CloudFormation for AWS resource deployment and manageability.

Modularity

Modularity is generally defined as how much easier it is for you to implement your infrastructure code using Terraform or CloudFormation. Terraform's modules can be easily written and reused as infrastructure code in multiple projects by multiple teams. Hence, it is quite easy to modularize Terraform code. You have the option to modularize CloudFormation code using nested stacks. You cannot modularize your CloudFormation stack itself in the same way you can prepare stacks of stacks in Terraform. You can use cross-stack references in CloudFormation and in the same way, it can be referenced in Terraform using the `terraform_remote_state` data source, which is used for deriving root module output from the other Terraform configuration.

Comparing both AWS CloudFormation and Terraform in terms of modularity, we can see that Terraform seems to be more friendly and easy to use.

Validation

Both tools allow you to validate infrastructure, but there is a difference. Using CloudFormation, you can validate AWS CloudFormation stacks, which means it will check only the syntax of your stack. If you want to update some resources, you can use CloudFormation change sets, which let you review the changes, meaning whether that specific resource defined in AWS CloudFormation will get replaced or updated, before you actually execute AWS CloudFormation stacks.

In the Terraform workflow, there is a `plan` phase that provides complete information about what resources are getting modified, deleted, or created before you deploy the infrastructure code.

In AWS CloudFormation, you have CloudFormation Designer, which can help you to know all the resources you have defined in AWS CloudFormation. This will provide you with insight before you actually go ahead and perform the deployment.

In terms of validation, both Terraform and AWS CloudFormation have certain options to perform it. The only major difference that can be noticed is that in AWS CloudFormation, you would be required to use multiple AWS services whereas in Terraform, we are easily able to validate with just its workflow.

Readability

There is no doubt that when you write a Terraform configuration file you will find it easier compared to AWS CloudFormation. Terraform configuration files are written in HCL, meaning they are more presentable when you understand the syntax. When you think about CloudFormation, which is generally written in JSON or YAML, it is quite complex in terms of its readability, especially JSON; YAML is somewhat fine to read and understand but the problem with YAML is regarding its whitespace and linting. In Terraform also, there is the linting concern but Terraform has a smart way of performing linting by just running `terraform fmt -recursive`, which will perform linting to all the Terraform configuration files present in the directory, whereas in YAML, you have to perform something manually or use some YAML validator tool that can help you with the validations.

So, in a nutshell, Terraform has more preferred than to AWS CloudFormation.

License and support

AWS CloudFormation is a managed service from AWS offered for free whereas Terraform is an open source infrastructure automation tool provided by HashiCorp. There is the Terraform Cloud Enterprise product from HashiCorp that is a licensed version and Enterprise customers need to purchase it from HashiCorp.

In terms of support plans, AWS has their own support plan that can be taken from AWS, in the same way that Terraform has their support option that can be used by customers.

AWS console support

As AWS CloudFormation is a native AWS IaC tool, it naturally provides excellent support in the AWS console. In the AWS console, you can monitor the progress of deployments, see all deployment events, check various errors, integrate CloudFormation with CloudWatch, and so on. In the Terraform CLI, we don't have any console option, but in the Terraform Cloud Enterprise version, there is a console where you can see all the progress of the resource deployments.

Workflow

You must be thinking that AWS CloudFormation will be more mature from an operational perspective, and this is true to a certain extent, but here you need to understand the pain area. Let's take an example where you have a complex infrastructure of around 50 resources and you are creating it using AWS CloudFormation. After running for around 20-30 minutes, your AWS CloudFormation stack receives an error in one of the resources and because of that, the stack fails. In that case, it may roll back and destroy whatever it

had created in AWS or sometimes, it may be able to deploy some of the resources even though it failed provided you have disabled *rollback on failure* in AWS CloudFormation. For further reading on the rollback on failure option in AWS CloudFormation, you can refer to <https://aws.amazon.com/premiumsupport/knowledge-center/cloudformation-prevent-rollback-failure/>. But to be very honest, if it did roll back and destroyed what it had created, then as an operator, you won't be looking for something like this because this adds more time and effort. Now, when you provision your AWS infrastructure using the Terraform workflow, this will provision all the required infrastructure in AWS. Even though, let's suppose, the Terraform workflow failed, then it will maintain the existing resources that had already been created prior to the failure. You can re-run the Terraform configuration, and this will simply provision the remaining resources rather than creating all the resources from scratch, which will save a significant amount of time and effort from the point of view of reworking.

So, based on the preceding paragraph, it appears that both AWS CloudFormation and Terraform workflows can be used. It's totally depends upon the consumer and which one they prefer.

Error message understandability

Errors thrown by Terraform can be tough for a novice to understand because sometimes they provide a strange error that is generated from the respective provider's APIs. In order to have an understanding of all different sorts of errors, you need to have an understanding of the arguments that are supported by the providers in Terraform.

So, comparing the understandability of error messages between Terraform and AWS CloudFormation, on occasion you may find that AWS CloudFormation is straightforward as you will have experience of working with the latter, and then, with your knowledge of Terraform, you will easily be able to figure out most of the errors.

Infrastructure state management

Both AWS CloudFormation and Terraform maintain the state of the infrastructure in their own way. In Terraform, state files get stored in the local disk or you can provide a remote backend location such as AWS S3 where you wish to store your Terraform state file. This state file would be written in JSON and would hold the information of the infrastructure that is defined in the Terraform configuration file. So, it is quite easy for Terraform to validate and check any sort of configuration drift.

In contrast, AWS CloudFormation is native to AWS, so we don't need to worry about how AWS CloudFormation would maintain its state. AWS CloudFormation used to get a signal from the resources that got provisioned or managed through AWS CloudFormation. If there is any configuration drift, then AWS CloudFormation may not be able to detect it.

Validation

Both tools allow you to have validation of infrastructure, but there is a difference. Using the Azure CLI or PowerShell, you can validate a template by running `az group deployment validate`, which means it will only check the syntax of your template. To validate what all the resources are that get provisioned, you have to deploy the complete ARM template.

Terraform provides a validation with the `plan` phase that checks the current deployment in the Terraform state file, refreshes the status of the actual cloud configuration, and then calculates the delta between the current configuration and the target configuration.

In terms of validation, if you were to compare both, then here too, Terraform wins out over ARM templates.

Azure ARM templates versus Terraform

In this section, we will see how Terraform differs from ARM based on various parameters:



Figure 1.10 – ARM templates versus Terraform

Let's get started.

Cross-platform

The major benefit of Terraform is that you can use it with many cloud providers, including AWS, GCP, and Azure. The ARM template is more central to Azure whereas for Terraform, you just need to learn the HCL language syntax and Terraform workflow and start using any cloud provider, such as GCP, Azure, or AWS. So from this, we can understand that Terraform is cloud-agnostic and it would be easy for the user to implement infrastructure on any of these platforms using Terraform.

Language

Terraform is written in **HCL**. HCL's syntax is very powerful and allows you to reference variables, attributes, and so on, whereas ARM templates use JSON and is a little bit complex but very powerful, because you can use conditional statements, nested templates, and so on.

Keeping all of this in mind, here too, we would encourage end users and developers to use Terraform.

Modularity

Modularity generally means how much easier for you it is to implement your infrastructure code using Terraform or ARM templates. Terraform's modules can be easily written and reused as infrastructure code in multiple projects by multiple teams. Hence, it is quite easy to modularize the Terraform code.

You have the option to modularize ARM templates using a nested ARM template, although you also have the option of forming a nested template, the first one in line in your main template, and the second one using separate JSON files called from your main template.

Here, comparing both ARM templates and Terraform, Terraform scores more points than ARM templates, although in this case, ARM templates are still a viable option for end users if they are planning to create resources in Azure.

Readability

There is no doubt that when you are intending to write a Terraform configuration file, you will find it more readable compared to ARM templates. Terraform configuration files written in HCL are more presentable when you understand their syntax, and the best part is that if you have learned the syntax, you would be able to write and use it anywhere in any cloud provider, including Azure, GCP, and AWS. When you consider ARM templates, which are generally written in JSON, they are quite complex in terms of their readability.

License and support

ARM templates use an Azure-native IaC approach and are available for free, whereas Terraform is an open source tool available from HashiCorp. There is the Terraform Cloud Enterprise version of Terraform that enterprise customers can purchase. Regarding support, both Azure and Terraform have their own support plans that can be utilized by the customer.

Azure portal support

As an ARM template is a native Azure IaC tool, it naturally provides excellent support in the Azure portal. You can monitor deployment progress in the Azure portal.

There is no Azure portal support for Terraform but if a customer is willing to use the Terraform Cloud Enterprise product, then they can utilize a portal that shows enough information about the infrastructure that they are planning to manage through Terraform.

Workflow

In terms of workflow, both ARM templates and Terraform have very good features. If the deployment of resources is interrupted halfway through, both Terraform and ARM templates have the capability to deploy the remaining resources during the second run, ignoring existing resources.

There is one minor difference, however. ARM template deployment requires an Azure resource group in place before you run the ARM template code, whereas Terraform configuration code is able to create an Azure resource group during the runtime itself, just as you would be required to define a resource group code block in the configuration file. Hence, it seems that both ARM templates and Terraform are quite satisfactory in terms of workflow.

Error message understandability

Terraform code can generate some weird errors. There is no doubt that in the backend, it would be using ARM provider APIs, which would show those errors, and it might be difficult for those end users who are new to Terraform to understand them, whereas in ARM templates, errors can be easily identified and fixed as end users may have experience of working with ARM templates.

So, in terms of comparing error message understandability between Terraform and ARM templates, this process may be easier in ARM templates.

Infrastructure state management

Both ARM templates and Terraform provide good state management. Terraform provides **state file** management using the backend state mechanism. In the state file, Terraform maps the resource that's been defined in the configuration file with what has actually been deployed so that it can easily trace any configuration drift. ARM is able to roll back to a previous successful deployment. You may have read that ARM is stateless as it does not store any state files.

Google Cloud Deployment Manager versus Terraform

In this section, we will see how Terraform differs from Google Cloud Deployment Manager based on various parameters:



Figure 1.11 – Cloud Deployment Manager versus Terraform

Let's get started.

Cross-platform

As you have already seen a comparison of AWS and Azure, you should now have an understanding of Terraform, which can be used on any platform, including GCP too. You just need to know how you can write Terraform configuration files using HCL and their workflow.

Cloud Deployment Manager is an infrastructure deployment service that automates the creation and management of Google Cloud resources. So here, Terraform wins out over Cloud Deployment Manager.

Language

Terraform is written in **HCL**. HCL's syntax is very easy and powerful and it allows you to reference variables, attributes, and so on, whereas Cloud Deployment Manager uses YAML to create a configuration file and you can define multiple templates in the configuration file, which is written using Jinja or Python.

Keeping all of this in mind, here too we would encourage end users or developers to use Terraform because in order to use cloud deployment, you would be required to have command of Python or Jinja and YAML.

Modularity

Modularity means how easily you can define configuration files and reuse them for infrastructure provisioning and management using Terraform or Cloud Deployment Manager. Terraform's modules can be written and published easily, and these modules can easily be consumed by other product teams in their projects. So, modularizing Terraform code is quite straightforward.

You have the option to modularize Cloud Deployment Manager using a nested template inside a configuration file. These templates can be written in either *Python* or *Jinja*. A single configuration can import both Jinja and Python templates. These template files should be present locally or placed on a third-party URL so that Cloud Deployment Manager can access them.

Here, comparing both Cloud Deployment Manager and Terraform, Terraform scores more points than Google Cloud Deployment Manager, although Google Cloud Deployment Manager remains a viable option for end users here, too, if they are planning to create resources in Google.

Validation

Both tools allow you to validate infrastructure, but there is a difference. To figure out any syntax errors in the configuration file, you would be required to run it from Google Cloud Deployment Manager. This will throw any possible errors, as well as validating what resources get provisioned. In this case, you have to deploy a complete Google Cloud Deployment Manager configuration file in Google Cloud.

Terraform provides a validation with the plan phase that tells you what resources it is going to create, destroy, or update. It is used for comparison purposes with the existing Terraform state file and accordingly provides detailed insights to the administrator as to when they will run `terraform plan` before being deployed to Google Cloud.

In terms of validation, if you were to compare both, then here, too, Terraform wins out over Google Cloud Deployment Manager.

Readability

If you are intending to write a Terraform configuration file, you will find it more readable compared to a Google Cloud Deployment Manager configuration file. Terraform configuration files written in HCL are more presentable, and the best part is that if you have learned Terraform syntaxes, you would be able to write and use them anywhere in any cloud, including Google, GCP, and AWS, or even on-premises.

When you think about Cloud Deployment Manager, which is generally written in YAML, it is also easy to understand, but when you need to introduce a template in the configuration file, which is written in Jinja or Python, then it becomes complex to read and understand.

Therefore, I am definitely of the opinion that Terraform is easier to use compared with Cloud Deployment Manager.

Maintainability

At a certain level, I think Terraform is more maintainable because of its superior modularity and how you can create different resources in Google Cloud and store its state file in Google Cloud Storage.

A Cloud Deployment Manager configuration file that is written in YAML and whose template is written in Python or Jinja is not that easy to read, which means the consumer needs to learn multiple coding languages. The nested template appears very complex and you need to store the template file in localhost or a third-party URL. Defining its template code locally in the configuration file and consuming it in the configuration file is not possible.

Cloud Deployment Manager doesn't have any kind of state file whereas Terraform has a state file that is stored in the remote backend.

The Terraform code configuration can be stored in any version control tool, such as Git, Azure Repos, or Bitbucket, if we are planning to use CI/CD DevOps pipelines.

Maintaining both Terraform code and Google Cloud Deployment Manager templates would be a little bit complex because both need to be stored either locally or in the source control tools.

Google console support

As Google Cloud Deployment Manager is a native Google IaC tool used for the provisioning and maintenance of infrastructure, it provides excellent support in the Google Cloud console. You can see the progress in terms of deployment in the Google Cloud console.

There is no Google Cloud console support for Terraform, though there is a separate portal if you are using the Terraform Cloud Enterprise version.

Workflow

In terms of workflow, both Cloud Deployment Manager and Terraform have very good features. If the deployment of resources is interrupted halfway through, both Terraform and Cloud Deployment Manager have the capability to deploy the remaining resources during the second run, ignoring existing resources.

There is one minor difference, however. Cloud Deployment Manager deployment requires a project in place before you run the configuration file using Cloud Deployment Manager, whereas Terraform configuration code is able to create a Google project during the runtime itself, just as you would be required to define a Google project resource block code in the configuration file. Therefore, I feel both of them are quite satisfactory when it comes to the workflow.

Error message understandability

In the previous AWS and Azure comparisons, we mentioned that errors used to emanate from the respective providers' APIs, and learning how to handle those errors comes with experience. If you were already equipped with that knowledge, then you would easily be able to rectify those issues. Again, compared to Google Cloud Deployment Manager, Terraform code exhibits a number of difficult errors that would be difficult for you to understand if you were new to Terraform. It doesn't mean that you can easily understand Google Cloud Deployment Manager errors simply because they involve multiple coding languages, such as YAML, JSON, or Jinja.

So, in terms of comparing the understandability of error messages between Terraform and Google Cloud Deployment Manager, if you had experience with Google Cloud, then it would be instinctive for you and this will help you to understand the errors easily. Regarding handling errors with Terraform, it may take some time for you to figure it out as Google Cloud resource APIs release errors specific to that.

Infrastructure state management

Both Cloud Deployment Manager and Terraform provide good state management. Terraform provides state file management using a backend state mechanism and state files can be stored in Google Cloud Storage. Cloud Deployment Manager is capable of rolling back to a previous successful deployment, since deployment used to happen incrementally by default, and it is quite easy to implement any changes you wish to make in terms of Google resources using Cloud Deployment Manager.

We covered how Terraform differs from other IaC options, mainly in terms of being cross-platform, as well as its modularity, readability, infrastructure state management, language, maintainability, workflow, cloud console support, and so on, and how it can be used by major cloud providers such as AWS, Azure, and GCP. All these comparisons will help you to understand how Terraform is a more powerful IaC and how you can effectively choose Terraform IaC for your infrastructure provisioning rather than selecting any specific IaC. Moving on, let's try to gain an understanding of Terraform architecture.

An understanding of Terraform architecture

With the help of the preceding section, we learned and became familiar with Terraform, which is just a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform is entirely built on a plugin-based architecture. Terraform plugins enable all developers to extend Terraform usage by writing new plugins or compiling modified versions of existing plugins:

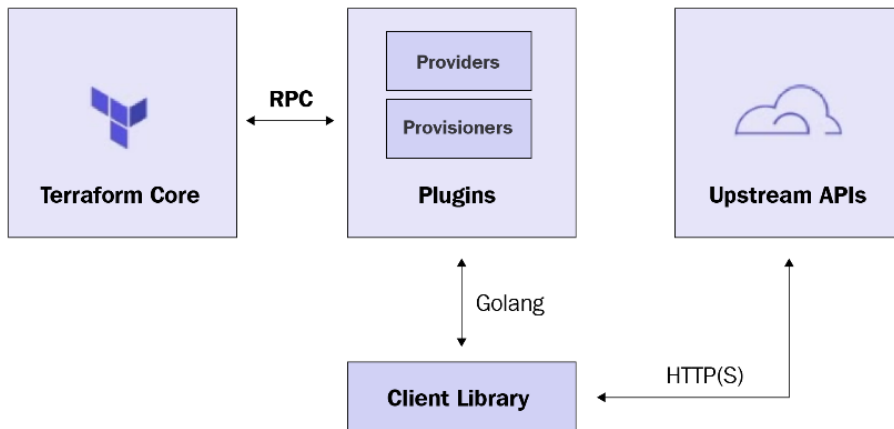


Figure 1.12 – Terraform architecture

As you can see in the preceding Terraform architecture, there are two key components on which Terraform's workings depend: Terraform Core and Terraform plugins. Terraform Core uses **Remote Procedure Calls (RPCs)** to communicate with Terraform plugins and offers multiple ways to discover and load plugins to use. Terraform plugins expose an implementation for a specific service, such as AWS, or a provisioner, and so on.

Terraform Core

Terraform Core is a statically compiled binary written in the Go programming language. It uses RPCs to communicate with Terraform plugins and offers multiple ways to discover and load plugins for use. The compiled binary is the Terraform CLI. If you're interested in learning more about this, you should start your journey from the Terraform CLI, which is the only entry point. The code is open source and hosted at github.com/hashicorp/Terraform.

The responsibilities of Terraform Core are as follows:

- IaC: Reading and interpolating configuration files and modules
- Resource state management
- Resource graph construction
- Plan execution
- Communication with plugins via RPC

Terraform plugins

Terraform plugins are written in the Go programming language and are executable binaries that get invoked by Terraform Core via RPCs. Each plugin exposes an implementation for a specific service, such as AWS, or a provisioner, such as Bash. All providers and provisioners are plugins that are defined in the Terraform configuration file. Both are executed as separate processes and communicate with the main Terraform binary via an RPC interface. Terraform has many built-in provisioners, while providers are added dynamically as and when required. Terraform Core provides a high-level framework that abstracts away the details of plugin discovery and RPC communication, so that developers do not need to manage either.

Terraform plugins are responsible for the domain-specific implementation of their type.

The responsibilities of provider plugins are as follows:

- Initialization of any included libraries used to make API calls
- Authentication with the infrastructure provider
- The definition of resources that map to specific services

The responsibilities of provisioner plugins are as follows:

- Executing commands or scripts on the designated resource following creation or destruction

Plugin locations

By default, whenever you run the `terraform init` command, it will be looking for the plugins in the directories listed in the following table. Some of these directories are static, while some are relative to the current working directory:

Directory	Purpose
.	For convenience during plugin development
Location of the Terraform binary (<code>/usr/local/bin</code> , for example)	For air-gapped installations; refer to the Terraform bundle
<code>terraform.d/plugins/<OS>_<ARCH></code>	For checking custom providers into a configuration's Version Control Systems (VCS) repository; not usually desirable, but sometimes necessary in Terraform Enterprise
<code>.terraform/plugins/<OS>_<ARCH></code>	Automatically downloaded providers
<code>~/.terraform.d/plugins</code> or <code>%APPDATA%</code> <code>\terraform.d\plugins</code>	The user plugins directory
<code>~/.terraform.d/plugins/<OS>_<ARCH></code> or <code>%APPDATA%</code> <code>\terraform.d\plugins\<OS>_<ARCH></code>	The user plugins directory, with explicit OS and architecture

You can visit the following link for more information on plugin locations:

<https://www.terraform.io/docs/extend/how-terraform-works.html#plugin-locations>

Important note

`<OS>` and `<ARCH>` use the Go language's standard OS and architecture names, for example, `darwin_amd64`.

Third-party plugins should usually be installed in the user plugins directory, which is located at `~/.terraform.d/plugins` on most OSes and `%APPDATA%\terraform.d\plugins` on Windows.

If you are running `terraform init` with the `-plugin-dir=<PATH>` option (with a non-empty `<PATH>`), this will override the default plugin locations and search only the path that you had specified.

Provider and provisioner plugins can be installed in the same directories. Provider plugin binaries are named with the scheme `terraform-provider-<NAME>_vX.Y.Z`, while provisioner plugins use the scheme `terraform-provisioner-<NAME>_vX.Y.Z`. Terraform relies on filenames to determine plugin types, names, and versions.

Selecting plugins

After finding any installed plugins, `terraform init` compares them to the configuration's version constraints and chooses a version for each plugin as defined here:

- If there are any acceptable versions of the plugin that have already been installed, Terraform uses the newest *installed* version that meets the constraint (even if `releases.hashicorp.com` has a newer acceptable version).
- If no acceptable versions of plugins have been installed and the plugin is one of the providers distributed by HashiCorp, Terraform downloads the newest acceptable version from `releases.hashicorp.com` and saves it in `.terraform/plugins/<OS>_<ARCH>`.

This step is skipped if `terraform init` is run with the `-plugin-dir=<PATH>` or `-get-plugins=false` options.

- If no acceptable versions of plugins have been installed and the plugin is not distributed by HashiCorp, then the initialization fails and the user must manually install an appropriate version.

Upgrading plugins

When you run `terraform init` with the `-upgrade` option, it rechecks `releases.hashicorp.com` for newer acceptable provider versions and downloads the latest version if available.

This will only work in the case of providers whose *only* acceptable versions are in `.terraform/plugins/<OS>_<ARCH>` (the automatic downloads directory); if any acceptable version of a given provider is installed elsewhere, `terraform init -upgrade` will not download a newer version of the plugin.

We have now covered *Terraform architecture* and learned about its core components, in other words, Terraform Core and Terraform plugins, and how Terraform Core communicates with Terraform plugins (provisioners and providers) using RPCs. In conjunction with this, you now have an understanding of the different sources from where Terraform will attempt to download plugins, because without plugins, you will not be able to use Terraform.

Summary

With the help of this chapter, you will now have a fair understanding of what Terraform is. Terraform is an IaC orchestration tool introduced by HashiCorp that is mainly used for the provisioning of infrastructure for your applications in code format. It is written in HCL, which is easily readable and understood, and we have also examined a number of different use cases, including multi-tier applications, SDNs, multi-cloud deployment, and resource schedulers, as well as examples of where you can use Terraform.

Moving on, you got to understand how Terraform differs from other IaCs, such as with ARM templates and CloudFormation, in terms of language, readability, modularity, error handling, and suchlike. Later, we explained Terraform architecture, and from there we got to know about how Terraform Core is used to communicate with Terraform plugins using RPCs, and by using Terraform Core, we would be able to run different Terraform CLI commands, including `init`, `plan`, and `apply`.

In the next chapter, we are going to discuss how you can install `terraform.exe` on a different machine, such as Windows, macOS, and Linux, which will help you in getting started with Terraform.

Questions

The answers to these questions can be found in the *Assessments* section at the end of this book:

1. What do you understand the definition of Terraform to be?
 - a) It is a virtual box.
 - b) It is an orchestration tool that is used for the provisioning of infrastructure.
 - c) It is a cloud.
 - d) It is a Google Chrome extension.

2. Which of the following are Terraform plugins? Select one or more:
 - a) Terraform providers
 - b) Terraform provisioners
 - c) Terraform resources
 - d) Terraform plan
3. A Terraform configuration file is written in which language?
 - a) Python
 - b) HCL
 - c) YAML
 - d) Go
4. Which of the following is not a Terraform provider?
 - a) Azure
 - b) AWS
 - c) GCP
 - d) SAP
5. Terraform is an orchestration tool developed by which company?
 - a) Microsoft
 - b) HashiCorp
 - c) Amazon
 - d) Google

Further reading

You can check out the following links for more information about the topics covered in this chapter:

- What is Terraform? <https://www.Terraform.io/intro/index.html>
- Terraform use cases: <https://www.Terraform.io/intro/use-cases.html>
- How Terraform works: <https://www.Terraform.io/docs/extend/how-Terraform-works.html>
- Terraform releases: <https://github.com/hashicorp/Terraform/releases>

2

Terraform Installation Guide

In the previous chapter, we discussed **Infrastructure as Code (IaC)**, mainly focusing on what Terraform is, and then we covered a comparison of Terraform with other available IaC options, such as *AWS CloudFormation templates*, *Azure ARM templates*, and *Google Cloud Deployment Manager*. Moving on, we had a detailed discussion about Terraform architecture and further learned about different versions of Terraform and their respective available features.

In this chapter, we will be focusing on how you can install **Terraform** on your local machine, whether it is Windows, Linux, or macOS. Once you are done with the installation of Terraform, you should be able to start drafting your configuration code in Terraform and run it locally from your system.

The following topics will be covered in this chapter:

- Installing Terraform on Windows
- Installing Terraform on Linux
- Installing Terraform on macOS

Technical requirements

To follow along with this chapter, you need to have an understanding of what Terraform is and in what scenarios you should use it, and some basic knowledge of major cloud providers, such as GCP, AWS, and Azure, would add more benefits.

Installing Terraform on Windows

Welcome to the Terraform installation guide. In this section, we are planning to install Terraform on a Windows machine. Then, after the installation of Terraform, we will try to define it in an environment variable so that you will be able to run the Terraform configuration file from any command line, such as *Windows CMD*, *PowerShell*, or *Bash*.

Downloading Terraform

Before you start using Terraform, you need to download the appropriate Terraform package for your operating system and architecture. The latest version v1.0.0 of Terraform is available and can be downloaded. After downloading the latest Terraform binary, extract it and update `terraform.exe` in your environment path so that you can run it from any command line.

Follow these steps to install Terraform on a Windows machine:

1. Visit the Terraform download URL, <https://www.terraform.io/downloads.html>, and download the Terraform package by selecting the Windows **32-bit** or **64-bit** operating system depending on your Windows operating system, as shown in the following screenshot:

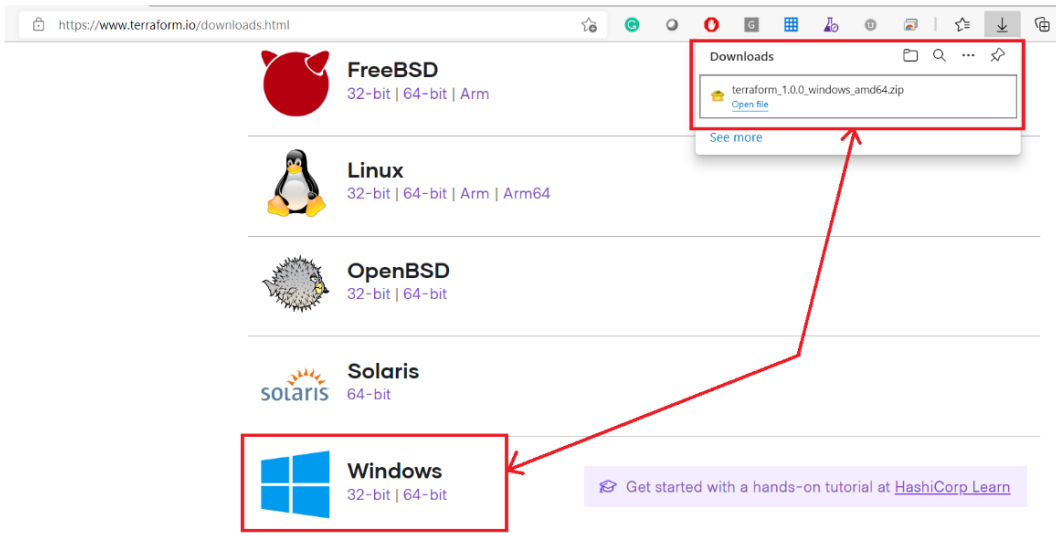


Figure 2.1 – Download Terraform (Windows)

2. Extract the downloaded file to any location on your computer, as shown in the following screenshot:

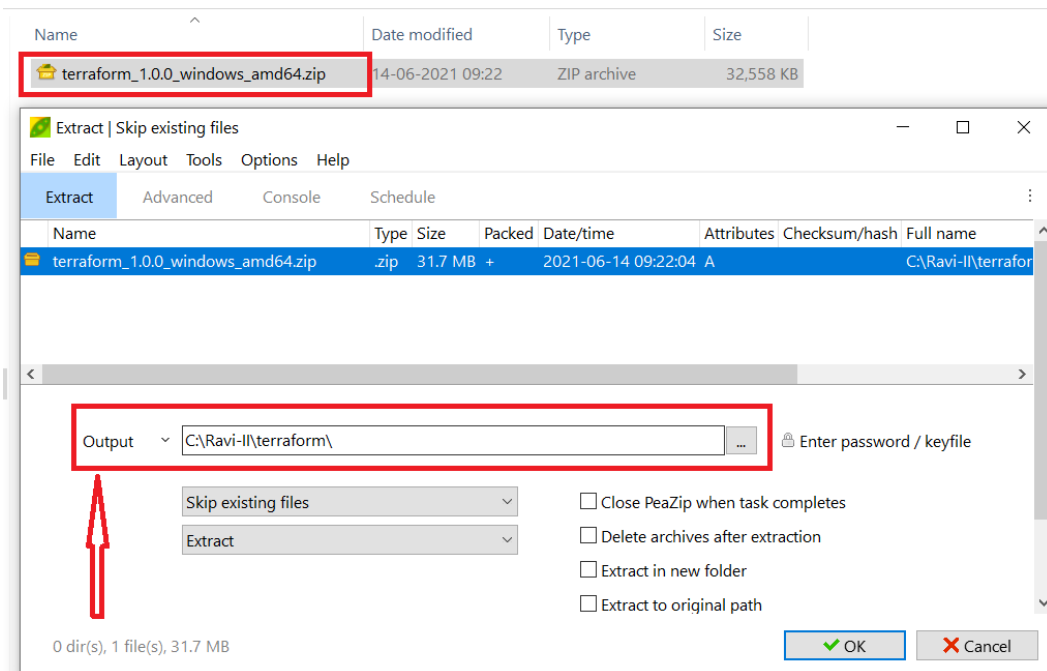


Figure 2.2 – Extract the terraform file

- Now, you need to set the environment variable path for `terraform.exe`. To do that, go to **This PC**, right-click on it, and go to **Properties** | **Advanced system settings** | **Environment Variables...**, as shown in *Figure 2.3*:

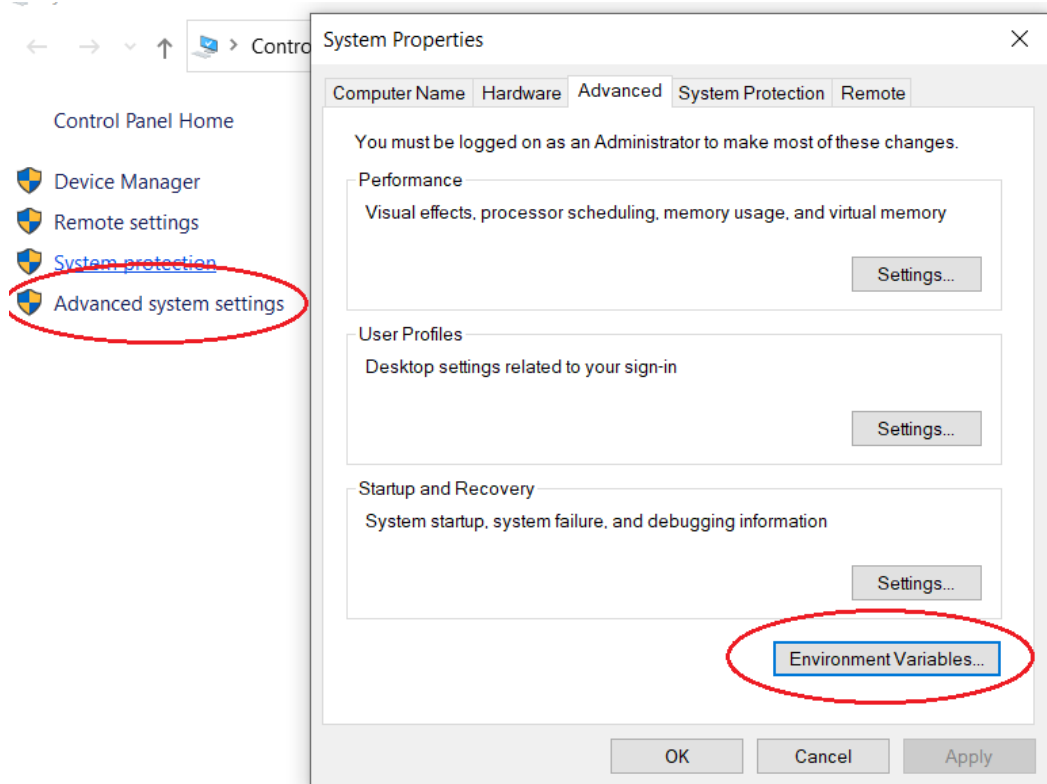


Figure 2.3 – Environment variable

- From the **Environment Variables** screen, go to the **System variables** section, and under **Path**, just add a path by locating the `terraform.exe` file where you saved it after doing the extraction. Then, click **OK | OK**, as shown in *Figure 2.4*:

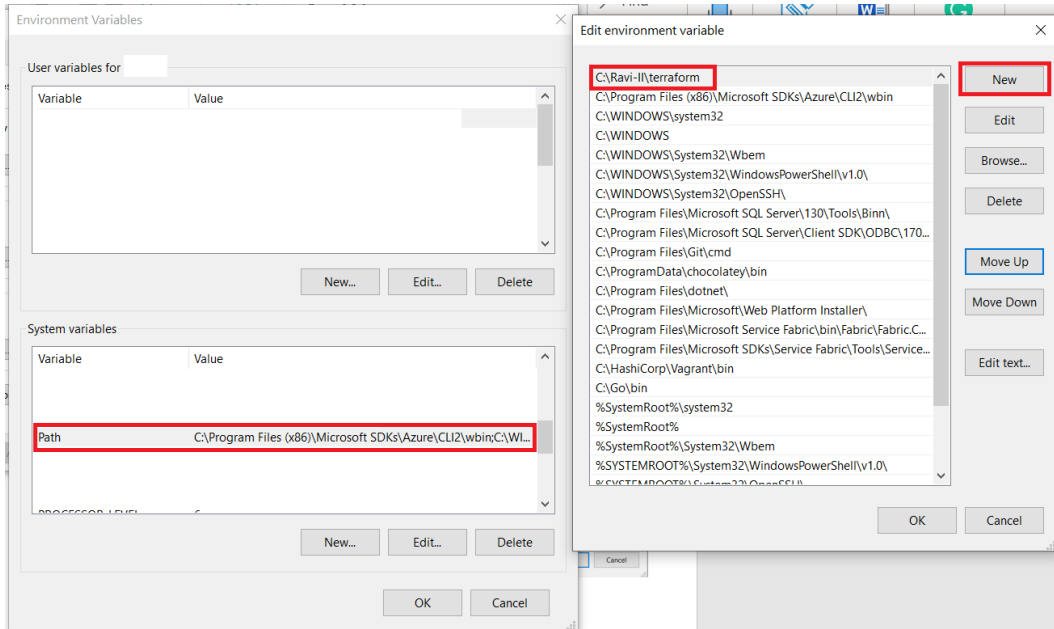


Figure 2.4 – Environment path variable

Now, let's try to verify whether we have Terraform on our system or not. Just open any CLI, such as Windows CMD or Windows PowerShell, and then type `terraform -v` or `terraform -h`; you should see the output shown in *Figure 2.5*:

```
Administrator: Command Prompt
C:\>terraform -v
Terraform v1.0.0
on windows_amd64

C:\>terraform -h
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init           Prepare your working directory for other commands
  validate      Check whether the configuration is valid
  plan          Show changes required by the current configuration
  apply         Create or update infrastructure
  destroy       Destroy previously-created infrastructure

All other commands:
  console       Try Terraform expressions at an interactive command prompt
  fmt          Reformat your configuration in the standard style
  force-unlock Release a stuck lock on the current workspace
  get          Install or upgrade remote Terraform modules
  graph        Generate a Graphviz graph of the steps in an operation
  import       Associate existing infrastructure with a Terraform resource
  login        Obtain and save credentials for a remote host
  logout       Remove locally-stored credentials for a remote host
  output       Show output values from your root module
  providers    Show the providers required for this configuration
  refresh      Update the state to match remote systems
  show         Show the current state or a saved plan
  state        Advanced state management
  taint        Mark a resource instance as not fully functional
  test         Experimental support for module integration testing
  untaint      Remove the 'tainted' state from a resource instance
  version      Show the current Terraform version
  workspace    Workspace management

Global options (use these before the subcommand, if any):
  -chdir=DIR   Switch to a different working directory before executing the
               given subcommand.
  -help       Show this help output, or the help for a specified subcommand.
  -version    An alias for the "version" subcommand.

C:\>
```

Figure 2.5 – Validating Terraform (Windows)

Important note

You can put `terraform.exe` in the `C:\Windows\system32` location, provided you don't want to define a separate path in the environment variables.

We have covered the installation of Terraform on Windows and learned how we can download it from the Terraform website, <https://www.terraform.io/downloads.html>, and then saw how we can set up Terraform locally on our system. With all this, you are ready to start drafting your configuration code. So now, let's learn how to install Terraform on a Linux machine.

Installing Terraform on Linux

As we have already learned how to install Terraform on Windows, now let's learn how to install Terraform on a *Linux machine*. Then, once we're done with the installation of Terraform, we will learn how we can verify whether a Linux machine has Terraform installed or not. For this section, we have considered an *Ubuntu Linux machine*; considering multiple Linux platforms such as CentOS, Red Hat, and SUSE is beyond the scope of this book, but in a nutshell, if you know how to install Terraform on one of the Linux platforms, you should easily be able to do so on other Linux platforms too.

Downloading Terraform

Before you start using Terraform, you need to download the appropriate Terraform package for your Linux operating system and architecture. The latest version v1.0 of Terraform is available and can be downloaded from <https://www.terraform.io/downloads.html> or https://releases.hashicorp.com/terraform/1.0.0/terraform_1.0.0_linux_amd64.zip. We have used v1.0.0 of Terraform.

Follow these steps to install Terraform on a Linux machine:

1. SSH to your Linux, that is, *Ubuntu*, machine using `putty.exe` or any client, then update its library using the `sudo apt update -y` command:

```
inmishrar@terraform-vm:~$ sudo apt update -y
Hit:1 http://azure.archive.ubuntu.com/ubuntu bionic InRelease
Get:2 http://azure.archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]
Get:3 http://azure.archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
1
Get:4 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Get:5 http://azure.archive.ubuntu.com/ubuntu bionic/universe amd64 Packages [857
0 kB]
```

Figure 2.6 – Linux VM update

2. Create a directory with any name and get inside it using the `cd` command. We are creating a directory named `terraform` using the `mkdir terraform && cd terraform` command.
3. Then, download *Terraform v1.0.0* using the `wget` command (https://releases.hashicorp.com/terraform/1.0.0/terraform_1.0.0_linux_amd64.zip):

```
inmishrar@terraform-vm: ~/terraform
inmishrar@terraform-vm:~/terraform$ wget https://releases.hashicorp.com/terraform/1.0.0/terraform_1.0.0_linux_amd64.zip
--2021-06-14 05:08:36-- https://releases.hashicorp.com/terraform/1.0.0/terraform_1.0.0_linux_amd64.zip
Resolving releases.hashicorp.com (releases.hashicorp.com)... 151.101.209.183, 2a04:4e42:3b::439
Connecting to releases.hashicorp.com (releases.hashicorp.com)|151.101.209.183|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 33043000 (32M) [application/zip]
Saving to: 'terraform_1.0.0_linux_amd64.zip'

terraform_1.0.0_linux_amd64.zip  100%[=====] 31.51M  59.8MB/s  in 0.5s
2021-06-14 05:08:37 (59.8 MB/s) - 'terraform_1.0.0_linux_amd64.zip' saved [33043000/33043000]

inmishrar@terraform-vm:~/terraform$ ls
terraform_1.0.0_linux_amd64.zip
inmishrar@terraform-vm:~/terraform$
```

Figure 2.7 – Download Terraform (Linux)

4. Install a software package called **unzip** that will help us to extract files from the existing ZIP file:

```
sudo apt install unzip -y
```

5. Once installed, you can unpack the already-downloaded Terraform ZIP file using the following command:

```
sudo unzip terraform_1.0.0_linux_amd64.zip
```

After running the preceding command, you will see the following:

```

inmishrar@terraform-vm:~/terraform$ sudo apt install unzip -y
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  linux-headers-4.15.0-144
Use 'sudo apt autoremove' to remove it.
Suggested packages:
  zip
The following NEW packages will be installed:
  unzip
0 upgraded, 1 newly installed, 0 to remove and 9 not upgraded.
Need to get 168 kB of archives.
After this operation, 567 kB of additional disk space will be used.
Get:1 http://azure.archive.ubuntu.com/ubuntu bionic-updates/main amd64 unzip amd64 6.0-21ubuntu1.1 [168 kB]
Fetched 168 kB in 0s (7014 kB/s)
Selecting previously unselected package unzip.
(Reading database ... 76932 files and directories currently installed.)
Preparing to unpack .../unzip_6.0-21ubuntu1.1_amd64.deb ...
Unpacking unzip (6.0-21ubuntu1.1) ...
Setting up unzip (6.0-21ubuntu1.1) ...
Processing triggers for mime-support (3.60ubuntu1) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
inmishrar@terraform-vm:~/terraform$ sudo unzip terraform_1.0.0_linux_amd64.zip
Archive:  terraform_1.0.0_linux_amd64.zip
  inflating: terraform
inmishrar@terraform-vm:~/terraform$ ls
terraform  terraform_1.0.0_linux_amd64.zip
inmishrar@terraform-vm:~/terraform$ █

```

Figure 2.8 – Unzipping Terraform

- Set the Linux path to point to Terraform with the following command:

```
export PATH=$PATH:$HOME/terraform
```

You can check whether Terraform installed or not by typing the `terraform --version` or `terraform --help` command. We will validate the presence of Terraform by using `terraform console` commands. You should be able to perform some of the operations from `terraform console` as shown:

```

inmishrar@terraform-vm:~/terraform$ echo "1+7" | terraform
console
8
inmishrar@terraform-vm:~/terraform$

```

The previously defined `terraform console` command gives us an output by calculating $1 + 7 = 8$, which tells us that Terraform is working on our Linux machine.

We have covered the installation of Terraform on Linux on an Ubuntu machine, and learned how we can download it from the Terraform website, <https://www.terraform.io/downloads.html> or <https://releases.hashicorp.com/terraform/>, and then saw how we can set up Terraform locally on our Linux system. After covering the initial setup of Terraform, you should be ready to start drafting the configuration file. So now, let's learn how to install Terraform on a macOS machine.

Installing Terraform on macOS

As you already know how to install Terraform on Linux, mainly on an Ubuntu machine, now let's learn how to install Terraform on a *Mac machine*. Once we're done with the installation of Terraform, we will learn how to verify whether a Mac machine has Terraform installed or not.

Downloading Terraform

Before you start using Terraform, you need to download the appropriate Terraform package for your Mac operating system and architecture. The latest version, such as v1.0.0, of Terraform is available and can be downloaded from <https://www.terraform.io/downloads.html> for a Mac system. For our case, we have used v1.0.0 of Terraform.

Follow these steps to install Terraform on a Mac machine:

1. To install Terraform on a MacBook, you simply have to download the Terraform binary from <https://www.terraform.io/downloads.html>, as you can see in *Figure 2.9*:

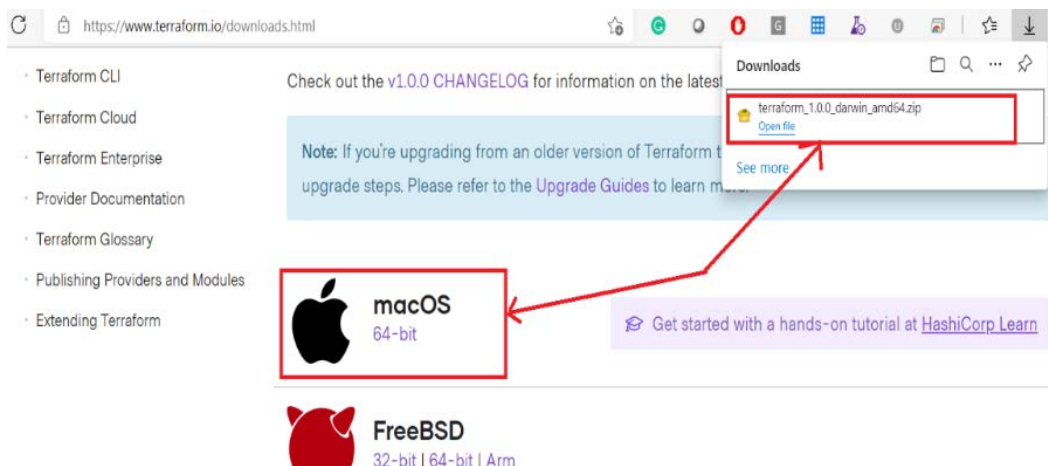


Figure 2.9 – Download Terraform (Mac)

2. After downloading the Terraform binary, unzip it and place it in any folder path; you can either update the `$PATH` variable to point to the location where the Terraform binary is or you can just move it to the local `bin` directory. If you choose the latter, then on MacBook, you will need to execute the following command:

```
sudo mv ./terraform /usr/local/bin:
```

```
MacBook-Pro:n komminenisrinandini$ sudo mv terraform /usr/local/bin
Password:
MacBook-Pro:n komminenisrinandini$ ls
MacBook-Pro:n komminenisrinandini$ cd /usr/local/bin
MacBook-Pro:bin komminenisrinandini$ ls
2to3                dumpcap              python3.8
2to3-3.8            easy_install-3.8    python3.8-config
VBoxAutostart       editcap              randpkt
VBoxBalloonCtrl     idle3                rawshark
VBoxBugReport       idle3.8              reordercap
VBoxDTrace          mergecap             terraform
VBoxHeadless        mmdbresolve         text2pcap
VBoxManage          pip3                 tshark
VBoxVRDP            pip3.8              vbox-img
VirtualBox          pydoc3              vboxwebsrv
capinfos            pydoc3.8            wireshark
capytype            python3
dftest              python3-config
MacBook-Pro:bin komminenisrinandini$
```

Figure 2.10 – Terraform path

3. You can validate whether Terraform was installed successfully on your MacBook by using either the `terraform -h` or `terraform -v` commands. You can see the output in *Figure 2.11*:

```
Hamids-MacBook-Pro:~ hamidraza$ terraform -v
Terraform v1.0.0
on darwin_amd64
Hamids-MacBook-Pro:~ hamidraza$ terraform -h
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init           Prepare your working directory for other commands
  validate       Check whether the configuration is valid
  plan           Show changes required by the current configuration
  apply          Create or update infrastructure
  destroy        Destroy previously-created infrastructure

All other commands:
  console        Try Terraform expressions at an interactive command prompt
  fmt           Reformat your configuration in the standard style
  force-unlock  Release a stuck lock on the current workspace
  get           Install or upgrade remote Terraform modules
  graph         Generate a Graphviz graph of the steps in an operation
  import        Associate existing infrastructure with a Terraform resource
  login         Obtain and save credentials for a remote host
  logout        Remove locally-stored credentials for a remote host
  output        Show output values from your root module
  providers     Show the providers required for this configuration
  refresh       Update the state to match remote systems
  show          Show the current state or a saved plan
  state         Advanced state management
  taint         Mark a resource instance as not fully functional
  test          Experimental support for module integration testing
  untaint       Remove the 'tainted' state from a resource instance
  version       Show the current Terraform version
  workspace     Workspace management

Global options (use these before the subcommand, if any):
  -chdir=DIR    Switch to a different working directory before executing the
                given subcommand.
  -help        Show this help output, or the help for a specified subcommand.
  -version     An alias for the "version" subcommand.
```

Figure 2.11 – Validate Terraform (Mac)

We have covered the installation of Terraform on macOS and learned how to download it from the Terraform website, <https://www.terraform.io/downloads.html>, and then saw how to set up Terraform locally on our *Mac* system. After covering the initial setup of Terraform, you should be ready to start drafting your first configuration file.

Summary

In this chapter, you learned how to install `terraform.exe` on different machines, such as Windows, Mac, and Linux, which will help you to get started with Terraform.

In the next chapter, we will discuss all the Terraform core components, such as resources, data, variables, output, inbuilt functions, backend, locals, iterations, providers, and provisioners. These core components that will be discussed will cover multiple providers, such as Azure, GCP, and AWS.

Questions

The answers to these questions can be found in the *Assessment* section at the end of this book:

1. How can you check which version of Terraform is installed on your system?
 - A. `terraform -psversion`
 - B. `terraform --version`
 - C. `terraform --help`
 - D. `terraform fmt`
2. You are using a Linux machine where Terraform is already installed. As you are very new to Terraform, you are looking for the "help" option in Terraform. Which command will you be executing? Select all the possible answers:
 - A. `terraform -h`
 - B. `terraform -help`
 - C. `terraform fmt`
 - D. `terraform plan`
3. Do you need to install the Go language library on your local system if you want to install Terraform?
 - A. Yes
 - B. No

Further reading

You can check out the following links for more information about the topics that were covered in this chapter:

- Terraform download: <https://www.terraform.io/downloads.html>
- Terraform releases: <https://github.com/hashicorp/Terraform/releases>
- Terraform installation guide: <https://learn.hashicorp.com/tutorials/terraform/install-cli>

Section 2: Core Concepts

From this section, you will be able to get full insight into the industry's best practices that are generally followed while provisioning enterprise-level infrastructure. You will get an in-depth understanding of the complete Terraform life cycle and how effectively Terraform code can be drafted and run using the Terraform CLI.

The following chapters will be covered under this section:

- *Chapter 3, Getting Started with Terraform*
- *Chapter 4, Deep Dive into Terraform*
- *Chapter 5, Terraform CLI*
- *Chapter 6, Terraform Workflows*
- *Chapter 7, Terraform Modules*

3

Getting Started with Terraform

In the previous chapter, we discussed the installation of Terraform on your local machine, whether it is Windows, Linux, or macOS. Once you are done with the installation of Terraform, you should be ready to start drafting your configuration code in Terraform and running it locally from your system.

In this chapter, we are going to discuss **Terraform plugins**, which include providers and provisioners. Here, we will discuss **Terraform providers**, while in a later chapter, we will discuss Terraform provisioners. Furthermore, we will see how you can take input from users by defining **Terraform variables** and then, once you have provided variables and your configuration file is ready, you can validate the output using output values. We will even discuss how you can use already existing resources by calling the **data block** in your configuration code.

The following topics will be covered in this chapter:

- Introducing Terraform providers
- Knowing about Terraform resources
- Understanding Terraform variables
- Understanding Terraform output
- Understanding Terraform data

Technical requirements

To follow along with this chapter, you need to have an understanding of what Terraform is and how you can install Terraform on your local machine. Some basic knowledge of major cloud providers such as GCP, AWS, and Azure would add extra benefits during the chapter.

Introducing Terraform providers

In this section, we will learn what **Terraform providers** are. Going further, we will try to understand Terraform providers for the major clouds, such as GCP, AWS, and Azure. Once you have an understanding of Terraform providers, we will see how you can define a Terraform providers block in your configuration code and how your Terraform configuration code downloads specific providers when you execute `terraform init`.

Terraform providers

You may be wondering how Terraform knows where to go and create resources in, let's say, for example, a situation where you want to deploy a virtual network resource in Azure. How will Terraform understand that it needs to go and create the resources in Azure and not in other clouds? Terraform manages to identify the Terraform provider. So, let's try to understand what the definition of a Terraform provider is:

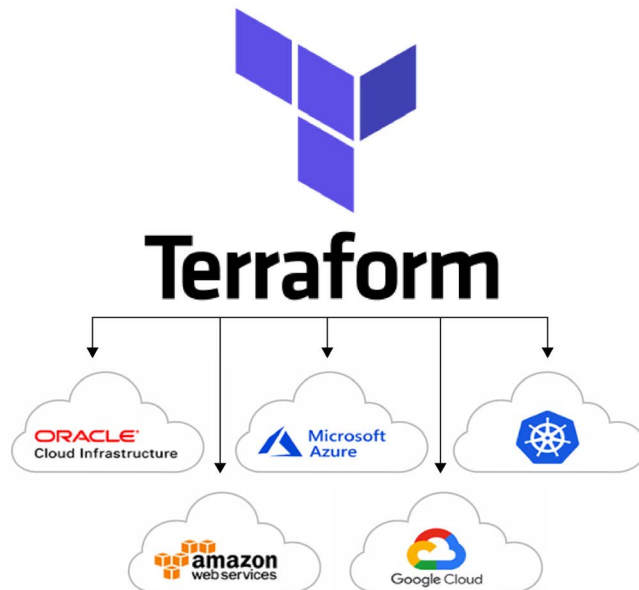


Figure 3.1 – Terraform providers

A **provider** is an executable plugin that is downloaded when you run the `terraform init` command. The Terraform provider block defines the version and passes initialization arguments to the provider, such as authentication or project details. Terraform providers are the component that makes all the calls to HTTP APIs for specific cloud services, that is, AzureRM, GCP, or AWS.

A set of resource types is offered by each provider plugin that helps in defining what arguments a resource can accept and which attributes can be exported in the output values of that resource.

Terraform Registry is the main directory of publicly available Terraform providers and hosts providers for most major infrastructure platforms. You can also write and distribute your own Terraform providers, for public or private use. For more understanding about Terraform Registry, you can follow <https://registry.terraform.io/>.

Providers can be defined within any file ending with `.tf` or `.tf.json` but, as per best practices, it's better to create a file with the name `providers.tf` or `required_providers.tf`, so that it would be easy for anyone to follow and, within that file, you can define your provider's code. The actual arguments in a provider block may vary depending on the provider, but all providers support the meta-arguments of `version` and `alias`.

In Terraform, there is a list of community providers that are contributed to and shared by many users and vendors. These providers are not all tested and are not officially supported by HashiCorp. You can see a list of the Terraform community providers at <https://www.terraform.io/docs/providers/type/community-index.html>.

If you are interested in writing Terraform providers, then you can fill in this form: https://docs.google.com/forms/d/e/1FAIpQLSeenG02tGEmz7pntIqMKlp5kY53f8AV5u88wJ_H1pJc2CmvKA/viewform#responses or visit <https://registry.terraform.io/publish/provider>.

We will not be discussing how you can write your own Terraform providers, but if you are interested in exploring it further, you can visit <https://learn.hashicorp.com/tutorials/terraform/provider-setup> and <https://github.com/hashicorp/terraform-plugin-docs>.

Let's try to understand what a provider block looks like for AzureRM, GCP, and AWS.

AzureRM Terraform provider

As we discussed regarding providers in Terraform, HashiCorp has introduced the **AzureRM provider**. Now, let's try to understand the code for the Azure provider:

```
# We strongly recommend using the required_
providers block to set the
# Azure Provider source and version being used
terraform {
  required_version = ">= 1.0"
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "2.54.0"
    }
  }
}

# Configure the Microsoft Azure Provider
provider "azurerm" {
  features {}
  subscription_id = "...."
  client_id       = "...."
  client_secret   = "...."
  tenant_id       = "...."
}
```

In the previous AzureRM provider code block, we had considered authentication of Terraform to Azure using the service principle. Let's try to understand what are the different arguments supported:

- `features` (required): Some of the Azure provider resource behaviors can be customized by defining them in the `features` block.
- `client_id` (optional): The client ID can be taken from the service principle. It can source from the `ARM_CLIENT_ID` environment variable.
- `client_secret` (optional): This is the client secret that you can generate for the service principle that you have created. This can also be sourced from the `ARM_CLIENT_SECRET` environment variable.
- `subscription_id` (optional): This provides your Azure subscription ID. It can be sourced from the `ARM_SUBSCRIPTION_ID` environment variable.

- `tenant_id` (optional): This provides your Azure tenant ID. It can be sourced from the `ARM_TENANT_ID` environment variable.

In order to authenticate to your Azure subscription, you are required to provide values for `subscription_id`, `client_id`, `client_secret`, and `tenant_id`. Now, it is recommended that either you pass these values through an environment variable or use cached credentials from the Azure CLI. As per the recommended best practice, avoid hardcoding secret information, such as credentials, into the Terraform configuration. For more details about how to authenticate Terraform to an Azure provider, you can refer to <https://www.terraform.io/docs/providers/azurerm/index.html>.

Let's now try to get a detailed understanding of the `version` and `features` arguments defined in the provider code.

The `version` argument defined in the code block is mainly used to constrain the provider to a specific version or a range of versions. This would prevent downloading a new provider that may contain some major breaking changes. If you don't define the `version` argument in your provider code block, Terraform will understand that you want to download the most recent provider during `terraform init`. If you wish to define versions in the provider block, HashiCorp recommends that you create a special `required_providers` block for Terraform configuration, as follows:

```
terraform {  
  required_version = ">= 1.0"  
  required_providers {  
    azurerm = {  
      source = "hashicorp/azurerm"  
      version = "2.54.0"  
    }  
  }  
}
```

Rather than setting the version of a provider for each instance of that provider, the `required_providers` block sets it for all instances of the provider, including child modules. Using the `required_providers` block makes it simpler to update the version on a complex configuration.

Let's see what different possible options you have when passing version values in the provider code block:

- `>= 2.54.0`: Greater than or equal to the version.
- `= 2.54.0`: Equal to the version.

- `!= 2.54.0`: Not equal to the version.
- `<= 2.54.0`: Less than or equal to the version.
- `~> 2.54.0`: This one is funky. It means any version in the `2.54.X` range. It will always look for the rightmost version increment.
- `>= 2.46, <= 2.54`: Any version between `2.46` and `2.54`, inclusive.

One of the most common arguments is `~>`, which means you want the same major version while still allowing minor version updates. For instance, let's say there's a major change coming to the Azure provider in version 2.0. By setting the version to `~>1.0`, you would allow all version 1 updates that come down while still blocking the big 2.0 release.

Let's try to understand what the `features` argument is doing within this Azure provider code block.

As per the latest update, you can control the behavior of some of the Azure resources using this `features` block. More details about what resources you can control are defined as follows.

The `features` block supports the following Azure resources or services:

- `key_vault`
- `template_deployment`
- `virtual_machine`
- `virtual_machine_scale_set`
- `log_analytics_workspace`

The `key_vault` block supports the following arguments:

- `recover_soft_deleted_key_vaults` (optional): The default value is set to `true`. It will try to recover a key vault that has previously been soft deleted.
- `purge_soft_delete_on_destroy` (optional): The default value is set to `true`. This will help to permanently delete the key vault resource when we run the `terraform destroy` command.

The `template_deployment` block supports the following argument:

- `delete_nested_items_during_deletion` (optional): The default value is set to `true`. This will help to delete those resources that have been provisioned using the ARM template when the ARM template got deleted.

The `virtual_machine` block supports the following argument:

- `delete_os_disk_on_deletion` (optional): The default value is set to `true`. This will help to delete the OS disk when the virtual machine got deleted.

The `virtual_machine_scale_set` block supports the following argument:

- `roll_instances_when_required` (optional): The default value is set to `true`. This will help to roll the number of the VMSS instance when you update SKU/images.

The `log_analytics_workspace` block supports the following arguments:

- `permanently_delete_on_destroy` (optional). The default value is set to `true`. This will help to permanently delete the log analytics when we perform `terraform destroy`.

The previously defined `features` block arguments have been taken from <https://www.terraform.io/docs/providers/azurerm/index.html>. For more information, you can check out that URL.

In the following example, you can see how you can define a `features` argument in the Azure provider code block and can control specific properties of the Azure key vault, You can use the same approach and customize all the previously defined arguments of the Azure resource or services:

```
provider "azurerm" {  
  version = "~> 2.54.0"  
  features {  
    key_vault {  
      recover_soft_deleted_key_vaults = false  
    }  
  }  
}
```

Now, let's try to understand a use case where you want to deploy multiple Azure resources in different subscriptions. Terraform provides an argument called `alias` in your provider block. Using that `alias` argument, you can reference same provider multiple times with a different configuration in your configuration block.

Here is an example of the code:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    azurearm = {
      source = "hashicorp/azurearm"
      version = "2.54.0"
    }
  }
}

# Configure the Microsoft Azure Provider
provider "azurearm" {
  features {}
}

provider "azurearm" {
  features {}
  alias = "nonprod_01_subscription"
}

# To Create Resource Group in specific subscription
resource "azurearm_resource_group" "example" {
  provider = azurearm.nonprod_01_subscription
  name      = "example-resources"
  location = "West Europe"
}
```

Moving on, let's try to understand how you can define multiple different providers in the Terraform configuration file. Here is one of the code snippets:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    azurearm = {
      source = "hashicorp/azurearm"
      version = "2.54.0"
    }
    random = {
      source = "hashicorp/random"
    }
  }
}
```

```
    version = "3.1.0"
  }
}
# Configure the Microsoft Azure Provider
provider "azurerm" {
  features {}
}
# Configure the Random Provider
provider "random" {}
resource "random_integer" "rand" {
  min = 1
  max = 50
}
resource "azurerm_resource_group" "examples" {
  name      = "example1-resources-${random_integer.rand.result}"
  location = "West Europe"
}
```

As described in the previous code, we have defined two different providers: `random` and `azurerm`. The `random` provider will help us to generate a random integer between 1 and 50 and the result of that integer will get appended to the Azure resource group name. Using this approach, we can define multiple different providers in the same configuration file.

Let's try to understand how we can define a provider for our *AWS*. This will also be using the same approach as we observed with *AzureRM*.

AWS Terraform provider

We've already discussed the Terraform *AzureRM* provider. Similarly, HashiCorp has introduced an *AWS* provider. Let's try to understand how the *AWS* provider can be defined in the Terraform configuration file.

The following is a code snippet:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    aws = {
```

```
    source = "hashicorp/aws"
    version = "~> 3.35.0"
  }
}
}
provider "aws" {
  region      = "us-east-1"
  access_key = "..."
  secret_key = "..."
}
```

There are many arguments supported by the AWS provider code block. A few of them are described here, but for more information regarding all the arguments, you can visit <https://registry.terraform.io/providers/hashicorp/aws/latest/docs>:

- `access_key` (optional): This is the AWS access key. It must be provided, but it can also be sourced from the `AWS_ACCESS_KEY_ID` environment variable, or via a shared credentials file if a profile is specified.
- `secret_key` (optional): This is the AWS secret key. It must be provided, but it can also be sourced from the `AWS_SECRET_ACCESS_KEY` environment variable, or via a shared credentials file if a profile is specified.
- `region` (optional): This is the AWS region where you want to deploy AWS resources. This can be sourced from the `AWS_DEFAULT_REGION` environment variable, or via a shared credentials file if a profile is specified.

For authentication to your AWS account, you can define `access_key` and `secret_key` in the environment variable because, as you know, hardcoding of a secret in the provider code block is not recommended, so it is better to pass it during the runtime itself or define it in the environment variable. For more details about this authentication option, you can visit <https://registry.terraform.io/providers/hashicorp/aws/latest/docs>.

The rest of the common arguments, such as `alias` and `version`, which we discussed in the Azure provider, can be used in the AWS provider block as well, so we will skip them here.

Google Terraform provider

Like the AWS and Azure providers, HashiCorp has introduced a Google cloud provider. So far, you will have a fair understanding of the Terraform providers, so let's try to see how we can define the Terraform provider code block for Google Cloud:

```
terraform {  
  required_version = ">= 1.0"  
  required_providers {  
    google = {  
      source = "hashicorp/google"  
      version = "3.63.0"  
    }  
  }  
}  
  
provider "google" {  
  credentials = file("account.json")  
  project     = "my-google-project-id"  
  region     = "europe-west2"  
  zone       = "europe-west2-b"  
}
```

Like the AWS and Azure providers, the Google provider also supports many arguments. We have highlighted a few of them here:

- `credentials` (optional): This is a JSON file that holds login information to the Google cloud. You can provide the file path and filename.
- `project` (optional): This is the name of the Google project where resources are to be managed.
- `region` (optional): This is the region where we want to deploy our Google resources.
- `zone` (optional): This is a specific data center within the region that can be defined.

To have an understanding of `alias` and `version` arguments, you can go back and read the Azure provider section because the working principle in all the providers is the same; you just need to understand the concept behind it.

We have discussed the Terraform providers, specifically covering Azure, AWS, and GCP. You must have understood how to define a provider code block in your Terraform configuration file. Along with that, you should have learned how you can use multiple same providers in your configuration file by defining `alias`. After learning about the Terraform providers, let's try to understand how you can write a resource code block in the Terraform configuration file, which is the only code actually helping you to provision or make changes to your existing resources. We will cover resource code blocks in the major clouds: Azure, AWS, and Google.

Knowing about Terraform resources

Having acquired a good understanding of Terraform providers, now we are going to discuss resources in Terraform. Let's try to understand how a resource code block is defined in the Terraform configuration file and why it is so important. First of all, let's see what a Terraform resource is.

Terraform resources

Resources are the most important code blocks in the Terraform language. By defining a resource code block in the configuration file, you are letting Terraform know which infrastructure objects you are planning to create, delete, or update, such as *compute*, *virtual network*, or higher-level PaaS components, such as *web apps* and *databases*. When you define a resource code block in the configuration file, it starts with the provider name at the very beginning, for example, `aws_instance`, `azurerm_subnet`, and `google_app_engine_application`.

Azure Terraform resource

It is very important for you to understand how you should write your Terraform resource code in the configuration file. We will take a very simple example of the *Azure public IP address*, which you can see in the following screenshot:

```

resource "azurem_resource_group" "example" {
  name     = "test-resources"
  location = "West US 2"
}
resource "azurem_public_ip" "azure-pip" {
  name                = "test-pip"
  location             = azurem_resource_group.example.location
  resource_group_name = azurem_resource_group.example.name
  allocation_method   = "Dynamic"
  idle_timeout_in_minutes = 30

  tags = {
    environment = "test"
  }
}

```

Figure 3.2 – Azure Terraform resource

As you can see in *Figure 3.2*, the red highlighted text is an actual resource that you are planning to update, create, or destroy. In the same way, Azure has many such resources and you can find out detailed information about it from <https://www.terraform.io/docs/providers/azurem/>:

The screenshot shows the Terraform Registry page for the `azurem` provider. The left-hand navigation menu is highlighted with a red box, and a red arrow points to the `Azure Resources` section. The main content area shows the `Authenticating to Azure` section, which lists four methods for authenticating to Azure:

- Authenticating to Azure using the Azure CLI
- Authenticating to Azure using Managed Service Identity
- Authenticating to Azure using a Service Principal and a Client Certificate
- Authenticating to Azure using a Service Principal and a Client Secret

Figure 3.3 – Azure resources

There, on the left-hand side of the website given previously, you will be able to see all the Azure resources as shown in *Figure 3.3*. You will be able to write your configuration file using those Azure resources provided.

The **blue highlighted text** is just a *local name* for the Terraform to the particular resource code that we are writing. In *Figure 3.2*, we are creating an Azure resource group and a public IP address. So, we have defined a local name for the resource group as `example` and `azure-pip` for the Azure public IP address.

The **green highlighted text** in *Figure 3.2* provides you with information on how you can reference certain arguments from other resource blocks. In the preceding example, we want to create an Azure public IP address. To provision the Azure public IP address, you will be required to provide the resource group name and location, which can be obtained from the earlier defined resource group code block.

With you now having an idea of how to define the Azure resource code block in the Terraform configuration file, let's see how we can do it in AWS.

AWS Terraform resource

You may be thinking that Terraform would behave separately and that there would be a change in the syntax when defining the resource code block in the Terraform configuration file for AWS, but this is not true. It follows the same approach in terms of how we can define it for the Azure resources. Let's try to discuss this using an example of AWS:

```
resource "aws_instance" "web" {  
  ami           = "ami-a1b2c3d4"  
  instance_type = "t2.micro"  
}
```

As you can see in the preceding code snippet, we are trying to deploy an EC2 instance in AWS. So for that, we have defined a resource block declaring a resource of a given type (`"aws_instance"`), which has a local name of `"web"`. The local name is mainly used to refer to this resource in any other resource, data, or module code block. For more information about all the available AWS resources, you can visit <https://registry.terraform.io/providers/hashicorp/aws/latest/docs>.

So far, we have looked at resource code blocks for Azure and AWS and haven't seen any difference in the syntax. You may be thinking that it will definitely get changed in the Google Cloud. In our next section, we are going to define how you can write a Google resource code block in the Terraform configuration file.

Google Terraform resource

We have already seen how we can define a resource block code for AWS and Azure. In terms of Terraform resource syntax, there are no differences; the only thing that will be different in all the providers is the resource block arguments because every provider has its defined arguments that can be passed into the resource code block. Let's see an example of Google Cloud Terraform resource code for creating **Google App Engine**:

```
resource "google_project" "my_project" {
  name          = "My Project"
  project_id    = "your-project-id"
  org_id        = "1234567"
}

resource "google_app_engine_application" "app" {
  project      = google_project.my_project.project_id
  location_id = "us-central"
}
```

Here, we are using the `google_project` resource block having a local name `my_project` to create a Google Cloud project and `google_app_engine_application` with a local name `app` to create a Google App Engine resource.

We have covered the Terraform resource code block and learned about how to write a resource block for cloud providers such as *AWS*, *Azure*, and *GCP*. We have even learned that there is no difference in the syntax of the resource code blocks; there may only be differences in terms of the argument supported by each provider. Moving forward, we are going to understand how you can take input from users, such as the name of the EC2 instance, by defining Terraform variables.

Understanding Terraform variables

In an earlier section, you learned about how you can write a resource code block in the Terraform configuration file. In that resource code block, we either hardcoded argument values or we referenced them from another resource code block. Now, we will try to understand how we can define those hardcoded values in a variable and define them in a separate file that can be used again and again.

Terraform variables

If you have basic experience of writing any *scripting* or *programming language*, you must have noticed that we can define some variables and use those defined variables again and again in the whole script. Likewise, in other programming languages, Terraform also supports variables that can be defined in the Terraform configuration code. The only difference between other programming language variables and **Terraform variables** is that, in Terraform variables, you are supposed to define input values when you want to execute your Terraform configuration code. We will be explaining the Terraform input variables approach for all three clouds – Azure, AWS, and GCP.

Azure Terraform input variables

Let's try to take the same Azure public IP address example that we discussed in the *Knowing about Terraform resources* section. We will try to define variables for most of the arguments so that we can use this resource code again and again by just providing values for the defined *input variables*. Here is the code snippet that will provide you with an idea of how you can define variables in the Terraform configuration code. You can simply take this code into any file ending with `.tf`. We generally try to put them into `main.tf`:

```
# To Create Resource Group
resource "azurerm_resource_group" "example" {
  name      = var.rgname
  location  = var.rglocation
}

# To Create Azure Public IP Address
resource "azurerm_public_ip" "azure-pip" {
  name                = var.public_ip_name
  location             = azurerm_resource_group.example.
location
  resource_group_name = azurerm_resource_group.example.name
  allocation_method   = var.allocation_method
  idle_timeout_in_minutes = var.idle_timeout_in_minutes
  tags                 = var.tags
}
```

In the preceding code, you may have observed how we defined the variables against all the resource arguments. The syntax is `<argument name> = var.<variable name>`. We have highlighted name and location in the resource group so that you get an understanding of how generally we are supposed to define Terraform input variables. After defining the Terraform input variable, don't forget to declare it. We can declare it in any file ending with `.tf`. Generally, we prefer to declare it in a separate file with the name `variables.tf`. Variables need to be defined within the variable block, in other words, `variables <variable name>`. Here is a code snippet for your reference:

```
# variables for Resource Group
variable "rgname" {
  description = "(Required)Name of the Resource Group"
  type        = string
  default     = "example-rg"
}
variable "rglocation" {
  description = "Resource Group location like West Europe etc."
  type        = string
  default     = "West Europe"
}
# variables for Azure Public IP Address
variable "public_ip_name" {
  description = "Name of the Public IP Address"
  type        = string
  default     = "Azure-pip"
}
variable "allocation_method" {
  description = "Defines the allocation method for this IP
address. Possible values are `Static` or `Dynamic`"
  type        = string
  default     = "Dynamic"
}
variable "idle_timeout_in_minutes" {
  description = "Provide Idle timeout in minutes"
  type        = number
  default     = 30
}
variable "tags" {
```

```
description = "A map of tags to assign to the resource.  
Allowed values are `key = value` pairs"  
type        = map(any)  
default = {  
  environment = "Test"  
  Owner       = "Azure-Terraform"  
}  
}
```

Only declaring Terraform variables won't be helpful. In order to complete the Terraform workflows, these variables need to get values, so there are four ways of having Terraform variable values. The first approach that we can follow is by defining variable values in the Terraform environment variables. Terraform will look for the values in the environment variable, starting with `TF_VAR_`, followed by the name of the declared variable, as you can see here:

```
$ export TF_VAR_rgname=example-rg
```

Secondly, we can store variable values in either a default supported file named `terraform.tfvars` or `terraform.tfvars.json` or in a file name ending with `.auto.tfvars` or `.auto.tfvars.json`.

We are showing here how you can define them in the `terraform.tfvars` file:

```
rgname          = "Terraform-rg"  
rglocation      = "West Europe"  
idle_timeout_in_minutes = 10  
tags = {  
  environment = "Preprod"  
  Owner       = "Azure-Terraform"  
}  
allocation_method = "Dynamic"  
public_ip_name    = "azure-example-pip"
```

If you are planning to define input variable values in any other file, such as `testing.tfvars`, then you will be required to explicitly mention the filename in the given Terraform cmdlet: `terraform apply -var-file="testing.tfvars"`. This means that Terraform will be able to read the values from that `.tfvars` file.

The third way of having Terraform variable values is during runtime. If you defined the variable in `main.tf` or `variables.tf` and its input values are missing, then it will prompt you to provide respective variable values during the runtime itself, as can be seen in the following snippet:

```
$ terraform plan
var.rglocation
  Resource Group location like West Europe etc.
Enter a value:
```

The fourth way could be to define a variable value directly as a `default` value while declaring the variables, as can be seen in the following code snippet:

```
variable "rgname" {
  description = "(Required)Name of the Resource Group"
  type        = string
  default     = "example-rg"
}
```

Using all these methods, we can take the values of variables and Terraform will be able to complete its workflow.

In the previously defined `variables.tf` code, you can see many constraint types, including `number`, `string`, and `map`. We will discuss these in our upcoming *Chapter 7, Terraform Modules*, so for now, just understand how you can define variables and how you can define input variable values. There is one more thing you must have noticed, in our variable code, we have mentioned default values. Here, the question is which one should be preferred; if you are defining variable values in a Terraform environment variables, defining in `terraform.tfvars`, defining default values, or providing values during the runtime. Hence, things happen in the following sequence: *Environment variable values | values during runtime | terraform.tfvars | default values*.

AWS Terraform input variables

In the previous section, we learned about Azure Terraform input variables. You learned about what exactly variables are and how you can define Terraform input variables. Let's try to consider one of the examples for AWS:

```
# You can define this code in main.tf or in any file named like
aws-ec2.tf
# To create ec2 instance in AWS
resource "aws_instance" "web" {
```



```
ami          = var.ami
instance_type = var.instance_type
}
```

The previously defined AWS resource code block helps us to provision an EC2 instance in AWS. Let's try to define the `variables.tf` file for it:

```
# variables for Resource Group
variable "ami" {
  description = "Name of the AMI"
  type        = string
}
variable "instance_type" {
  description = "Name of the instance type"
  type        = string
}
```

We have to define the `variables.tf` file for the `ec2_instance` resource code. Next, we need to have the values of those defined variables. So, let's define the values in the `terraform.tfvars` file:

```
ami          = "ami-a1b2c5d6"
instance_type = "t1.micro"
```

GCP Terraform input variables

We have learned about defining input variables in AWS and Azure, and there is no difference in terms of defining it for GCP as well. Let's try to take some sample Terraform configuration code for *GCP* and see how we can define input variables' code for it. We are going to discuss this with the same resource code as earlier that is, for App Engine in Google Cloud:

```
resource "google_project" "my_project" {
  name          = var.myproject_name
  project_id    = var.project_id
  org_id        = var.org_id
}
resource "google_app_engine_application" "app" {
```

```
project      = google_project.my_project.project_id
location_id  = var.location_id
}
```

Let's try to define our `variables.tf` file for the previous code, which is going to deploy Google App Engine for us:

```
# variables for Google Project
variable "myproject_name" {
  description = "Name of the google project"
  type        = string
}
variable "project_id" {
  description = "Name of the project ID"
  type        = string
}
variable "org_id" {
  description = "Define org id"
  type        = string
}
variable "location_id" {
  description = "Provide location"
  type        = string
}
```

We're done with defining all the variables in a `variable.tf` file. Now, let's try to pass values of them in `terraform.tfvars`:

```
myproject_name = "My Project"
project_id     = "your-project-id"
org_id         = "1234567"
location_id    = "us-central"
```

We have covered Terraform variables and acquired an understanding of what the best practices of defining Terraform variables are and how we can take input from the users by passing variable values in Terraform environment variables, `terraform.tfvars`, default values, and runtime from the CLI. The take-away from this entire topic is how effectively you can write a Terraform configuration file with the help of Terraform variables. In the next topic, we will be discussing *Terraform output*. Terraform output will help you in validating what you can expect as output when you create or update any resources.

Understanding Terraform output

In this section, we are going to see how you can define the **Terraform output** file as well as what the best practices are for referencing the output of one resource as input for other dependent resources. We will be discussing Terraform output for AWS, GCP, and Azure.

Terraform output

Let's try to understand what this Terraform output is and ideally, what we can achieve from it, as well as why we need to define Terraform output for any of the Terraform configuration files. Output values are the return values of a Terraform resource/module/data, and they have many use cases:

- The output from one resource/module/data can be called into other resources/modules/data if there is a dependency on the first resource. For example, if you want to create an Azure subnet and you have already created an Azure virtual network, then, in order to provide the reference of your virtual network in the subnet, you can use the output of the virtual network and consume it in subnet resources.
- You can print certain output of the resources/modules/data on the CLI by running `terraform apply`.
- If you are using a remote state, other configurations via a `terraform_remote_state` data source can help you to access root module outputs.

Terraform manages all of your resource instances. Each resource instance helps you with an export output attribute, which can be used in the other configuration code blocks. Output values help you to expose some of the information that you might be looking for. Once you have provisioned a particular resource instance using Terraform, or even existing resources, output values can also be referred to using the data source's code block.

Let's try to understand how we can define Terraform output in Azure, AWS, and GCP.

Azure Terraform output

In the Terraform resource topic, we considered the Azure public IP address. Let's try to take the same resource code block and see how we can extract output from that resource:

```
resource "azurerm_resource_group" "example" {
  name      = "resourceGroup1"
  location = "West US"
}

resource "azurerm_public_ip" "example" {
  name                = "acceptanceTestPublicIp1"
  resource_group_name = azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  allocation_method  = "Static"
  tags = {
    environment = "Production"
  }
}
```

From the `azurerm_public_ip` resource code block, we can export the following attributes:

- `id`: The public IP ID.
- `name`: The public IP address name.
- `resource_group_name`: The resource group name of the public IP address.
- `ip_address`: The IP address value that was allocated.
- `fqdn`: The **fully qualified domain name (FQDN)** of the DNS record associated with the public IP. `domain_name_label` must be specified to get the FQDN. This is the concatenation of `domain_name_label` and the regionalized DNS zone.

Let's see how you can define the output code block in your `main.tf` file, or how you can create a separate `output.tf` file and define everything there itself. We recommend that you define all your output code in a separate file, in other words, `output.tf`:

```
output "id" {
  value = azurerm_public_ip.example.id
}
output "name" {
  value = azurerm_public_ip.example.name
}
output "resource_group_name" {
  value = azurerm_public_ip.example.resource_group_name
}
output "ip_address" {
  value = azurerm_public_ip.example.ip_address
}
output "fqdn" {
  value = azurerm_public_ip.example.fqdn
}
```

From the previous code, we manage to get the possible output after creating an Azure public IP address.

Important note

Dynamic public IP addresses aren't allocated until they're attached to a device (for example, a virtual machine/load balancer). Instead, you can obtain the IP address once the public IP has been assigned via the `azurerm_public_ip` data source.

We have discussed Terraform output and seen how we can validate it. Let's try to see how we can use a Terraform attribute reference from a resource/module/data code block while creating any new resource. For a better understanding of the Terraform attribute reference, we have taken an Azure load balancer resource:

```

# To Create Azure Resource Group
resource "azurerms_resource_group" "example" {
  name      = "Terraform-rg"
  location  = "West Europe"
}

# To Create Azure Public IP Address
resource "azurerms_public_ip" "example" {
  name                = "Terraform-pip"
  location             = azurerms_resource_group.example.location
  resource_group_name = azurerms_resource_group.example.name
  allocation_method   = "Static"
  idle_timeout_in_minutes = 10
}

# To Create Azure Load Balancer
resource "azurerms_lb" "example" {
  name                = "Terraform-LoadBalancer"
  location             = azurerms_resource_group.example.location
  resource_group_name = azurerms_resource_group.example.name

  frontend_ip_configuration {
    name                = azurerms_public_ip.example.name
    public_ip_address_id = azurerms_public_ip.example.id
  }
}

```

Figure 3.4 – Azure Terraform attribute reference

As shown in *Figure 3.4*, while creating an Azure load balancer, we are taking the reference values from *Azure public IP address*, in other words, the public IP address name and public IP address resource ID. Actually, this is not a Terraform output, but it is an attribute reference, meaning that one block value can be called into another.

AWS Terraform output

As we have learned how we can extract output when we are creating any Azure resource, let's now see how we can define the same in AWS. If you already understand the syntax of defining output values, this will remain the same for any Terraform providers. We are taking an example of a VPC resource to demonstrate AWS Terraform output further:

```

# To Create AWS VPC
resource "aws_vpc" "terraform-vpc" {
  cidr_block      = "10.0.0.0/16"
  instance_tenancy = "default"
  tags = {
    Environment = "Terraform-lab"
  }
}

```

```
}  
}
```

Let's try to see what output values we can expect from the VPC resource block. Many arguments can be exported, as you can see in *Figure 3.5*, or visit <https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/vpc#attributes-reference>. We are going to discuss a few of these, such as `id` and `cidr_block`. From there, you can get an idea of how output can be extracted for the other attributes mentioned:

- `arn` - Amazon Resource Name (ARN) of VPC
- `id` - The ID of the VPC
- `cidr_block` - The CIDR block of the VPC
- `instance_tenancy` - Tenancy of instances spin up within VPC.
- `enable_dns_support` - Whether or not the VPC has DNS support
- `enable_dns_hostnames` - Whether or not the VPC has DNS hostname support
- `enable_classiclink` - Whether or not the VPC has Classiclink enabled
- `main_route_table_id` - The ID of the main route table associated with this VPC. Note that you can change a VPC's main route table by using an `aws_main_route_table_association`.
- `default_network_acl_id` - The ID of the network ACL created by default on VPC creation
- `default_security_group_id` - The ID of the security group created by default on VPC creation
- `default_route_table_id` - The ID of the route table created by default on VPC creation
- `ipv6_association_id` - The association ID for the IPv6 CIDR block.
- `ipv6_cidr_block` - The IPv6 CIDR block.
- `owner_id` - The ID of the AWS account that owns the VPC.

Figure 3.5 – AWS VPC output attributes

The following code snippet will give you an insight as to how you can validate the respective output following creation of the *AWS VPC*:

```
output "id" {
  value = aws_vpc.terraform-vpc.id
}
output "cidr_block" {
  value = aws_vpc.terraform-vpc.cidr_block
}
```

You may be wondering whether you need to consume the output of this AWS VPC in any other resource, such as an *AWS subnet*, and how you can do this. Let's try to understand this, which is known as Terraform attribute referencing. Already, in the *Azure Terraform output* section, we explained that by using an implicit reference, you would be able to consume the one resource/module/data code block in another. Here is the code snippet to create a subnet within a specific AWS VPC:

```
resource "aws_subnet" "terraform-subnet" {
  vpc_id      = aws_vpc.terraform-vpc.id
  cidr_block  = "10.0.0.0/24"

  tags = {
    Environment = "Terraform-lab"
  }
}
```

You can see in the previous code how we manage to perform Terraform attribute referencing from one resource code block and consume it in another resource code block. In the same way, you can do this with data sources and modules.

GCP Terraform output

In this section, we will try to understand how you can define output values for GCP resources. Let's start by understanding the output with a simple Google Cloud resource. In the following code, we have defined the resource code block for creating a Google App Engine resource:

```
resource "google_project" "my_project" {
  name      = "My Project"
  project_id = "your-project-id"
```



```
org_id      = "1234567"
}
resource "google_app_engine_application" "app" {
  project      = google_project.my_project.project_id
  location_id = "us-central"
}
```

Here is a list of the attributes you can export as output from the Google App Engine resource:

- `id` - an identifier for the resource with format `{{project}}`
- `name` - Unique name of the app, usually `apps/{PROJECT_ID}`
- `app_id` - Identifier of the app, usually `{PROJECT_ID}`
- `url_dispatch_rule` - A list of dispatch rule blocks. Each block has a `domain`, `path`, and `service` field.
- `code_bucket` - The GCS bucket code is being stored in for this app.
- `default_hostname` - The default hostname for this app.
- `default_bucket` - The GCS bucket content is being stored in for this app.
- `gcr_domain` - The GCR domain used for storing managed Docker images for this app.
- `iap` - Settings for enabling Cloud Identity Aware Proxy
 - `oauth2_client_secret_sha256` - Hex-encoded SHA-256 hash of the client secret.

Figure 3.6 - GCP App Engine output attributes

From the list of arguments defined in *Figure 3.6*, we will consider `id` and `name`, which, as you can see in the following code snippet, you can define within `main.tf` or in a separate file such as `output.tf`:

```
output "id" {
  value = google_app_engine_application.app.id
}
output "name" {
  value = google_app_engine_application.app.name
}
```

Terraform output optional arguments

Terraform output supports some arguments such as `description`, `sensitive`, and `depends_on`, which are described as follows:

- `description`: In the output values for your reference, you can define its description so that you can understand what you are getting as an output value. You can refer to the following code snippet to get an understanding of how you can define `description` in the output code block:

```
output "instance_ip_addr" {
  value      = aws_instance.server.private_ip
  description = "The private IP address of the main
server instance."
}
```

- `sensitive`: If you want to set output values that have sensitive information, then you can define the `sensitive` argument. In the following code snippet, we have declared the `sensitive` argument in the output value code block:

```
output "db_password" {
  value      = aws_db_instance.db.password
  description = "The password for logging in to the
database."
  sensitive  = true
}
```

When you define output values as `sensitive`, this prevents Terraform from showing its values on the Terraform CLI after running `terraform apply`. Still, there is some chance that it may be visible in the CLI output for some other reasons, such as if the value is referenced in a resource argument. This has been taken care of in Terraform version 0.14 and above, which prevents the displaying of sensitive output in the CLI output. You can refer to the following blog post regarding it: <https://www.hashicorp.com/blog/terraform-0-14-adds-the-ability-to-redact-sensitive-values-in-console-output>.

Even if you have defined output values as `sensitive`, they will still be recorded in the state file, and they will be visible in plain text to anyone who has access to the state file.

- `depends_on`: Output values are just used to extract information of the resource that got provisioned or that already exists. You may be thinking, why do we need to define any kind of dependency for the output values? Just as you generally define `depends_on` while using the resource/module/data code level so that Terraform will understand and maintain the resource dependency graph while creating or reading the existing resources, in the same way, if you wish to define an explicit `depends_on` argument in your output values, then you can define it as shown in the following code snippet:

```
output "instance_ip_addr" {
  value      = aws_instance.server.private_ip
  description = "The private IP address of the main
server instance."

  depends_on = [
    # Security group rule must be created before this IP
address could
    # actually be used, otherwise the services will be
unreachable.
    aws_security_group_rule.local_access,
  ]
}
```

Important note

The `depends_on` argument should be used only as a last resort. When using it, always include a comment explaining why it is being used, so as to help future maintainers understand the purpose of the additional dependency.

We have discussed the Terraform output. You should now have a fair understanding of how you can use Terraform output values for the validation of the resources that you have provisioned, and you also got to know how you perform Terraform attribute referencing from one resource/module/data code block to another resource/module/data code block. In the upcoming section, we are going to discuss Terraform data. From there, you will learn how you can read already existing resources in your infrastructure. For the explanations, we will again be considering our three major cloud services: Azure, AWS, and GCP.

Understanding Terraform data

In this section, we are going to discuss how you can define Terraform data sources. You can also refer to <https://www.terraform.io/docs/configuration/data-sources.html> to see under which circumstances you would be able to use Terraform data sources. Just as with Terraform output, resources, providers, and variables, we will be concentrating on Terraform data sources for AWS, GCP, and Azure.

Terraform data sources

Let's try to understand **Terraform data sources** with the help of an example. Matt and Bob are two colleagues working for a company called **Obs** based out in the US. Obs is a shipping company that does business all over the world and recently, they started using multi-cloud AWS and Azure. A heated conversation is ongoing between Bob and Matt. Bob is saying that he will be doing all the IT infrastructure deployment using Azure-provided ARM templates, while Matt is saying that he would prefer to use Terraform. Now, the problem is that Bob has already provisioned some of the production infrastructures in Azure using the ARM template. Let's say, for example, he provisioned *one virtual network, five subnets, and five virtual machines*. Matt has been asked to create five more virtual machines in those existing virtual networks and subnets. He wants to use Terraform to perform this deployment. He has many questions in mind, such as how will he be able to read the existing infrastructure and how will he be able to write a Terraform configuration file for the new deployment? After some searching, he learns about Terraform data sources, which allow you to extract output or information from already existing resources that got provisioned by any other Terraform configuration, or manually or by any other means.

Azure Terraform data sources

In this section, we will explain how you can define Terraform data sources specific to Azure. Let's try to understand this with the help of an example. Suppose you already have an Azure virtual network created and now you are trying to create a new subnet in that existing virtual network. In this instance, how can you define your Terraform data source's code block? In the following code snippet, we demonstrate how you can get an output value from the existing virtual network:

```
data "azurerm_virtual_network" "example" {
  name                = "production-vnet"
  resource_group_name = "Terraform-rg"
}

output "virtual_network_id" {
```

```
value = data.azurerm_virtual_network.example.id
}
```

If you want to create a subnet in the existing virtual network, then the following defined code snippet can help you out by extracting the existing resource group and virtual network, and then allowing you to provision a new subnet inside that existing virtual network:

```
data "azurerm_resource_group" "example" {
  name = "Terraform-rg"
}
data "azurerm_virtual_network" "example" {
  name                       = "production-vnet"
  resource_group_name = data.azurerm_resource_group.example.name
}
resource "azurerm_subnet" "example" {
  name                       = "terraform-subnet"
  resource_group_name = data.azurerm_resource_group.example.name
  virtual_network_name = data.azurerm_virtual_network.example.name
  address_prefixes      = ["10.0.1.0/24"]
}
```

As with Azure virtual networks, there are Terraform data sources for each Azure service. If you wish to understand the syntax of writing an Azure Terraform data source code block in your Terraform configuration file, you can refer to <https://www.terraform.io/docs/providers/azurerm/>.

AWS Terraform data sources

Similar to the way in which we defined our Azure Terraform data sources code block, let's try to understand how we can define Terraform data sources for AWS. We want to create a subnet in an existing AWS VPC. You can refer to the following code snippet where we have defined `vpc_id` as an *input variable*:

```
variable "vpc_id" {}
data "aws_vpc" "example" {
  id = var.vpc_id
}
```

```
}  
resource "aws_subnet" "example" {  
  vpc_id          = data.aws_vpc.example.id  
  availability_zone = "us-west-2a"  
  cidr_block      = cidrsubnet(data.aws_vpc.example.cidr_  
block, 4, 1)  
}
```

For detailed information about each AWS service that can be defined in AWS Terraform data sources, you can refer to the AWS providers website at <https://registry.terraform.io/providers/hashicorp/aws/>.

GCP Terraform data sources

In this section, we are going to discuss how you can define the GCP Terraform data sources code in your configuration. To aid understanding, let's take a simple example of extracting existing GCP compute instance details. The following code snippet will give you an idea of how you can draft GCP Terraform data sources:

```
data "google_compute_instance" "example" {  
  name = "Terraform-server"  
  zone = "us-central1-a"  
}
```

If you want more information, you can refer to the Terraform Google provider website at <https://www.terraform.io/docs/providers/google/>, where each Google service is explained along with its data sources.

We have covered data sources' code blocks in the Terraform configuration files. We have learned how we can draft our Terraform configuration file using data sources for different providers such as Azure, AWS, and GCP. Data sources help to read already existing configurations and use that output in new or updated infrastructures.

Summary

In this chapter, you gained an understanding of Terraform's core components, including providers, resources, variables, output, and data sources. In a nutshell, a provider is an API plugin that you need if you want to deploy/update services for your infrastructure. Resources are the actual services that you are planning to update/deploy for your respective providers. Variables are input from the users that makes your configuration code reusable. The output is what you are expecting when you are creating/updating your resources. Data sources help you out with extracting existing resource configurations. All of these help you to draft your Terraform configuration file.

In the next chapter, we will get into a detailed discussion regarding backend configuration, provisioners, and inbuilt functions, how to perform debugging in Terraform, and how you can perform different kinds of iteration using `for` and other loops in Terraform.

Questions

The answers to these questions can be found in the *Assessments* section at the end of this book:

1. Terraform is written using the **HashiCorp Configuration Language (HCL)**. What other syntax can Terraform be written in?
 - A. JSON
 - B. YAML
 - C. TypeScript
 - D. XML
2. The following is a Terraform code snippet from your Terraform configuration file:

```
provider "aws" {  
  region = "us-east-1"  
}  
provider "aws" {  
  region = "us-east-2"  
}
```

When validated, it results in the following error:

```
Error: Duplicate provider configuration  
on main.tf line 5:  
provider "aws" {
```

```
A default provider configuration for "aws" was already
given at
main.tf:1,1-15. If multiple configurations are required,
set the "_____"
argument for alternative configurations.
```

Fill in the blank in the error message with the correct string from the following list:

- A. version
 - B. multi
 - C. label
 - D. alias
3. Referring to the following Terraform code, what is the local name for the resource that is defined?

```
resource "aws_instance" "example" {
ami = "ami-082b5a644766e6e6f"
instance_type = "t2.micro"
count = 2
}
```

- A. aws_instance
 - B. example
 - C. ami-082b5a644766e0e6f
 - D. t2.micro
4. Matt is implementing Terraform in his environment. He is looking to deploy some virtual machines with the virtual network in Azure. He has ascertained that one of his colleagues has already created a virtual network in Azure, and now he needs to create a virtual machine within that already existing virtual network. Suggest what he should use in his Terraform configuration code block:
- A. Make use of Terraform variables for the virtual network.
 - B. Make use of a Terraform resource block for the virtual network.
 - C. Make use of a data source for the virtual network.
 - D. None of the above.

5. You have been given a Terraform configuration file and have been asked to make it dynamic and reusable. What exactly will you be using to convert static parameters?
 - A. Output values
 - B. Terraform input variables
 - C. Data sources
 - D. Regular expressions

Further reading

You can check out the following links for more information about the topics that have been covered in this chapter:

- Terraform environment variables: <https://www.terraform.io/docs/commands/environment-variables.html>
- Terraform variables: <https://upcloud.com/community/tutorials/terraform-variables/>
- Terraform data sources: <https://stackoverflow.com/questions/47721602/how-are-data-sources-used-in-terraform>
- Terraform best practices: <https://www.terraform-best-practices.com/key-concepts>
- Terraform providers: <https://registry.terraform.io/browse/providers>
- Terraform resources: <https://www.cloudreach.com/en/resources/blog/guide-terraform-resource-dev/>

4

Deep Dive into Terraform

In the previous chapter, we discussed Terraform providers, which mainly help Terraform to understand which API to use for deployment. We also covered Terraform resources, which help you to consume the provider's service API to provision the respective services. Moving on, we discussed taking input from users by defining Terraform variables, which make Terraform code reusable. We also saw how you can validate the output of the resources that you have provisioned or that already exist using Terraform output, and finally, we discussed how you can use already-existing resources by calling the Terraform data block in your configuration code.

In this chapter, we are going to discuss the **Terraform backend**, which helps you to store your Terraform `tfstate` file. Furthermore, we will be covering **Terraform provisioners**, which help you to execute script within the Terraform configuration code. Later, we will be discussing different **Terraform loops (iterations)** that can be used within the Terraform configuration code. We will also see what different **Terraform built-in functions** are and how effectively you can use these functions in your configuration code. Finally, we will look at some options that Terraform provides in terms of debugging.

The following topics will be covered in this chapter:

- Introducing the Terraform backend
- Understanding Terraform provisioners

- Understanding Terraform loops
- Understanding Terraform functions
- Understanding Terraform debugging

Technical requirements

To follow along with this chapter, you need to have an understanding of writing Terraform configuration code by defining providers, resources, variables, data, and output. Along with this, some basic knowledge about major cloud providers such as GCP, AWS, and Azure would add benefit during the entire course. You can find all the code used in this chapter at the following link: <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter4>.

Check out the following link to see the Code in Action video:

<https://bit.ly/2SY35WS>

Introducing the Terraform backend

In this section, we are going to talk about the Terraform state file and the **Terraform backend**. As you know, Terraform follows a desired state configuration model where you describe the environment you would like to build using declarative code and Terraform attempts to make that desired state a reality. A critical component of the desired state model is mapping what currently exists in the environment and what is expressed in the declarative code. Terraform tracks this mapping through a JSON formatted data structure called a **state file**. We are going to look at where the Terraform state file can be stored, how it can be configured and accessed, and what the best practices for keeping a Terraform `tfstate` file are.

Terraform state

You are already aware of when you write the Terraform configuration file and how while executing `terraform init`, `plan`, and `apply`, it is used to generate a state file that stores information about your complete infrastructure or the services that you are trying to deploy using Terraform. The state file is used by Terraform to map real-world resources with your Terraform configuration file. By default, Terraform stores `terraform.tfstate` in the current working directory; you can store this state file in a remote storage location as well, such as Amazon S3 or Azure Blob storage. Terraform combines the configuration with the state file, as well as refreshing the current state of elements in the state file to create plans.

It performs a second round of refreshing when the `apply` phase is executed.

Terraform state files are just JSON files; it is not recommended to edit or make any changes to the state file. Generally, state files are used to hold a one-to-one binding of the configured resources with remote objects. Terraform creates each object and records its identity in the state file. If you wish to destroy any resources, then remove the configuration of the resource from the configuration file and run `terraform apply`, which will remove that specific resource from the state file. Remember, it is not recommended to remove resource bindings from the state file.

If you are adding or removing resources by some other means – let's suppose you created a resource manually and want to use that resource in Terraform – then you would be required to create a configuration matching the resource and then import that resource in the state file using the `terraform import` cmdlet. In the same way, if you want Terraform to forget one of the objects, then you can use the `terraform state rm` cmdlet, which would remove that object from the state file.

The purpose of the Terraform state file

As you know, Terraform generates a state file. You might be thinking, why does Terraform **Infrastructure as Code (IaC)** need a state file, and what will happen if Terraform doesn't have a state file? To answer that question, a state file in Terraform stores current knowledge of the state of the configured infrastructure, which reduces the complexity of the resource deployment when you are handling large Enterprise infrastructures. We are going to discuss what the main purpose of having a Terraform state file is and how it benefits us:

- **Mapping to the real world:** Terraform needs to have a database to map the Terraform configuration to the real world. When you define a resource such as `resource "azurerm_resource_group" "example"` in your configuration, Terraform defines this code block in JSON format in the state file mapping to a specific object. This tells you that Terraform has created an Azure resource group and in the future when you make any changes in the configuration code block, it will try to check the existing resource in the state file and will let you know accordingly whether that resource is being created, amended, or destroyed, whenever you run `terraform plan`. This helps to understand any kind of infrastructure deviation from the existing resources.
- **Metadata:** Along with the mapping between resources and objects that are there in the state file, Terraform needs to maintain metadata such as resource dependencies. Terraform is intelligent enough to understand which resource is to be created or destroyed and in what sequence. To perform these activities, Terraform stores resource dependencies in the state file.

- **Performance:** Terraform has basic mapping. It also stores a cache of all the attributes in the state file, which is one of the optional features of Terraform and is used only for performance improvements. When you run the `terraform plan` command, Terraform needs to know the current state of the resources in order to know the changes it needs to make, to reach out to the desired configuration.

For small infrastructures, Terraform can easily get the latest attributes from all your resources, which is the default behavior of Terraform, to sync all resources in the state before performing any operations such as `plan` and `apply`.

For large enterprise infrastructures, Terraform can take many hours to query all the attributes of the resources, totally depending on the size of the infrastructure, and it may surprise you in terms of the total time it takes. Many cloud providers don't provide an API that can support querying multiples resources at once. So, in this scenario, many users set `-refresh=false` as well as the `-target` flag, and because of this, the cached state is treated as the record of truth, which helps Terraform to cut down the plan phase time by almost half.

- **Syncing:** By default, Terraform stores the state file in the current working directory where you have kept your configuration file. This is acceptable if you are the only user who is using the Terraform configuration code. But what about if you have multiple team members working together on the same Terraform configuration code? Then, in that scenario, you just have to keep your Terraform state file in a **remote state**. Terraform uses remote locking so that only one person is able to run Terraform at a time. If at the same time other users try to run the Terraform code, then Terraform will throw an error saying that the state file is in a **locked state**. This feature of Terraform ensures that every time Terraform code is run, it should refer to the latest state file.

We have learned about Terraform state files and their purpose. Now, let's try to understand the Terraform backend types.

Terraform backend types

The Terraform state file needs to be stored at either the local or remote backend. We are going to discuss both the local and remote backends.

Local backend

Terraform generates a state file and it should be stored somewhere. In the absence of a remote backend configuration, it would be stored in the directory where you have kept the Terraform configuration file, which is the default behavior of Terraform. Let's see the folder structure for an example configuration:

```

.
├── .terraform
├── main.tf
└── terraform.tfstate

```

This Terraform configuration has been initialized and run through `plan` and `apply`. As a result, the `.terraform` directory holds the plugin information that you would have defined in the configuration file. In our case, we have taken the Azure provider, so in the `.terraform` directory, Terraform downloads the `azurerm` provider. The `terraform.tfstate` file holds the state of the configuration.

In the following code snippet, we have shown you how you can change the location of the local state file by using the `-state=statefile` command-line flag for `terraform plan` and `terraform apply`:

```

root@terraform-vm:~# terraform apply -state=statefile
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
inmishrar@terraform-vm:~#
inmishrar@terraform-vm:~# ls
main.tf  statefile  terraform-lab
inmishrar@terraform-vm:~# cat statefile
{
  "version": 4,
  "terraform_version": "1.0.0",
  "serial": 1,
  "lineage": "03e0af5c-c3a0-c2ae-9d40-d067358067dd",
  "outputs": {},
  "resources": []
}

```

Let's try to understand what will happen to the state file if you use `terraform workspace`. In our case, we have created a new Terraform workspace with the name `development`. After creating a new workspace and running `terraform plan` and `terraform apply`, we can see that Terraform creates a `terraform.tfstate.d` directory and a subdirectory for each workspace. In each workspace directory, a new `terraform.tfstate` file got created:

```
inmishrar@terraform-vm:~# terraform workspace new development
Created and switched to workspace "development"!
You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
inmishrar@terraform-vm:~# terraform workspace list
  default
* development
inmishrar@terraform-vm:~# terraform apply
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
inmishrar@terraform-vm:~# tree
.
├── main.tf
├── .terraform.lock.hcl
├── terraform.tfstate
└── terraform.tfstate.d
    ├── development
    └── terraform.tfstate
```

From the previously defined code block, we can understand that the same configuration file can be used under a different Terraform workspace (that is, a landscape such as `development` or `test`).

Remote backend

As we mentioned earlier, Terraform uses a local file backend by default. There are some advantages to using remote backends. A few of them have been explained as follows:

- **Working in a team:** Remote backends help you to store state files in a remote storage location and protect them with a state lock feature so that the state file will never get corrupted. The state file goes into a locked state whenever any of the team members are running the Terraform configuration file so that there won't be any conflict, which means no two team members will be able to run the Terraform configuration code simultaneously. Some Terraform backends, such as Terraform Cloud, hold the history of all the state revisions.
- **Keeping sensitive information off disk:** The Terraform state file usually has sensitive data such as passwords that is very critical. So, if you're using remote backends, then this information would be stored in remote storage and you can retrieve it from backends on demand and ensure it is only stored in memory. Most remote backends provide data encryption at rest. So, the state file inside remote backends will be secure and confidential.
- **Remote operations:** Suppose you are dealing with very large infrastructures or performing certain configuration changes. Because of that, the `terraform apply` command may take longer than expected to be executed. Some remote backends support remote operations that enable them to perform operations remotely. In this case, you can turn off your computer and you will find that the Terraform operation will keep on running remotely until it is completed. This remote operation is only supported by the *SaaS* product of HashiCorp, that is, *Terraform Cloud*.

Remote backends store your state data in a remote location based on the backend configuration you've defined in your configuration. Not all backends are the same, and HashiCorp defines two classes of backend:

- **Standard:** Includes state management and possibly locking
- **Enhanced:** Includes remote operations on top of standard features

The enhanced remote backends are either Terraform Cloud or Terraform Enterprise, which will be covered in *Chapter 10, Terraform Cloud and Terraform Enterprise*. Both of these services allow you to run your Terraform operations, such as `plan` and `apply`, on the remote service as well as locally.

Going through all the standard backends is beyond scope of this book. Still, we will be discussing a couple of them, such as **AWS S3**, **Azure Storage**, and **Google Cloud Storage (GCS)**.

Let's look at the Azure Storage backend as an example:

```
terraform {  
  backend "azurerm" {  
    storage_account_name = "terraform-stg"  
    container_name       = "tfstate"  
    key                   = "terraform.tfstate"  
    access_key           = "tyutyutyutryuvsd68564568..."  
  }  
}
```

From the previously defined code block, you will be required to define an access key to get full access to Azure Blob storage. Generally speaking, it is not recommended to define `access_key` in the backend code configuration. Defining credentials in the configuration file has a couple of issues; for example, you might be required to change the credentials at regular intervals, which means you would be required to keep on changing the backend configuration file. It's also not good practice to define credentials in plain text and store them on a source control or local machine.

Now, the problem is how you would define credentials in the backend configuration file. You don't have an option to define them as a variable because the backend configuration file runs during the initialization itself and until then, Terraform will not be able to understand the variables. In order to overcome this challenge, you can define partial backend configuration in the root module and the rest of the information at the runtime itself.

In the previously defined code block, we saw how to define backend configuration code for Azure Blob storage, but we were defining `access_key` in plain text in the backend configuration, which is not at all recommended. So, we will define the backend configuration code using the partial method. The following code snippet will give you an idea of how you are supposed to define a partial backend configuration:

```
terraform {  
  backend "azurerm" {  
    container_name = "tfstate"  
    key            = "terraform.tfstate"  
  }  
}
```

The values for `storage_account_name` and `access_key` can be supplied in one of three ways:

- When you run `terraform init`, you can pass values at the interactive terminal of the CLI.
- Through the `-backend-config` flag with a set of key-value pairs.
- Through the `-backend-config` flag with a path to a file with key-value pairs.

You even have the option to take the values from the environment variables; for example, you can define `ARM_ACCESS_KEY` as an environment variable in your local machine and during the Terraform initialization, Terraform will be able to take values from those environment variables. You can check the Azure Storage backend configuration to find out what arguments are supported in the environment variables: <https://www.terraform.io/docs/backends/types/azurerm.html>.

After doing the Terraform initialization for the first time with all the backend values, you won't be required to supply the values again for other commands. The backend information and other initialization data are stored in a local file in the `.terraform` subdirectory of your configuration. When you are pushing your configuration code to source control, make sure not to push this `.terraform` directory because it may have sensitive data, such as the authentication credentials of the backend configuration.

Important note

When you create a new repository on GitHub, in the repository, when you create a `.gitignore` file, you can select Terraform as one of the options. It will automatically exclude the `.terraform` directory, as well as files such as `terraform.tfvars` and `terraform.tfstate`. We recommend using `.gitignore` on any new project with those exemptions. You can further learn about `.gitignore` for Terraform at <https://github.com/github/gitignore/blob/master/Terraform.gitignore>.

Let's try to understand how the Terraform remote backend will help with collaboration within a team using an example.

Suppose you are collaborating with *John*, the network administrator, on a network configuration. Both of you will be updating your local copy of the configuration and then pushing it to version control. When it is time to update the actual environment, you run the `plan` and `apply` phases. During that period, the remote state file will be in a locked state, so John won't be able to make any changes simultaneously. When John runs the next plan, it will be using the updated remote state data from your most recent `apply` phase. Azure Blob storage has a native capability to support state locking and data consistency.

So, using the following code block, you can extract output from the existing remote backend using the Terraform data block. For more information about Azure Blob storage backend configuration and authentication, you can see <https://www.terraform.io/docs/backends/types/azurerm.html>:

```
data "terraform_remote_state" "example" {
  backend = "azurerm"
  config = {
    storage_account_name = "terraform-stg"
    container_name       = "tfstate"
    key                  = "terraform.tfstate"
  }
}
```

Now, let's try to see the remote backend configuration for the AWS S3 bucket. As with Azure Blob storage, AWS S3 also supports remote backend configuration for storing the Terraform state file, but if you want to have state locking and consistency, then you would be required to use *Amazon DynamoDB*:

```
terraform {
  backend "s3" {
    bucket = "terraform-state-dev"
    key    = "network/terraform.tfstate"
    region = "us-east-1"
  }
}
```

Let's try to read output from the AWS S3 backend configuration using data sources. The following code block will help you to understand how you can extract the respective output:

```
data "terraform_remote_state" "example" {
  backend = "s3"
  config = {
    bucket = "terraform-state-dev"
    key    = "network/terraform.tfstate"
    region = "us-east-1"
  }
}
```

The `terraform_remote_state` data source will return the following values:

```
data.terraform_remote_state.example:
id = 2016-10-29 01:57:59.780010914 +0000 UTC
addresses.# = 2
addresses.0 = 52.207.220.222
addresses.1 = 54.196.78.166
backend = s3
config.% = 3
config.bucket = terraform-state-dev
config.key = network/terraform.tfstate
config.region = us-east-1
elb_address = web-elb-790251200.us-east-1.elb.amazonaws.com
public_subnet_id = subnet-1e05dd33
```

Let's see how we can define the Terraform backend using *GCS*. For your knowledge, *GCS* stores the state file and supports state locking by default. The following is a code snippet:

```
terraform {
  backend "gcs" {
    bucket = "tf-state-prod"
    prefix = "terraform/state"
  }
}
```

Now, the question is, how can you read output from the already-configured Terraform backend in GCS? You can use the following code block:

```
data "terraform_remote_state" "example" {
  backend = "gcs"
  config = {
    bucket = "terraform-state"
    prefix = "prod"
  }
}

resource "template_file" "terraform" {
  template = "${greeting}"
  vars {
    greeting = "${data.terraform_remote_state.example.greeting}"
  }
}
```

In this section, we learned about the Terraform backend, both the local backend and the remote backend. By default, Terraform stores the state file in the local backend, but if you are looking for collaboration and security, then Terraform provides you with the option to store the state file in the remote backend. We have discussed how you can configure the remote backend using *AWS S3*, *Azure Blob storage*, and *GCS*. Along with this, we also discussed how you can read the content of the configured backend using Terraform data sources. With all this, you will have gained a strong understanding of the Terraform backend and its importance. In the upcoming section, we are going to discuss Terraform provisioners and will try to discuss some real-time examples of Terraform provisioners.

Understanding Terraform provisioners

In this section, we are going to talk about **Terraform provisioners**. Let's try to understand them with an example. Suppose you are working with one of your colleagues, *Mark*. You have both been given a task to deploy a complete enterprise infrastructure in Microsoft Azure that contains almost 10 Ubuntu servers, 10 virtual networks, 1 subnet per virtual network, and 5 load balancers, and all these virtual machines should have Apache installed by default before handing over to the application team. Mark suggests that in order to have Apache installed on all those servers, you can use Terraform provisioners. Now you're thinking, what is this Terraform provisioner and how can we use it?

Let's try to understand Terraform provisioners. Suppose you are creating some resources and you need to run some sort of script or operations that you want to perform locally or on the remote resource. You can fulfill this expectation using Terraform provisioners. The execution of Terraform provisioners does not need to be idempotent or atomic, since it is executing an arbitrary script or instruction. Terraform will not be able to track the results and status of provisioners in the same way it is used to doing for other resources. Because of this, HashiCorp recommends the use of Terraform provisioners as a last resort when you don't have any other option to complete your goal.

Important note

Don't use Terraform provisioners unless there is absolutely no other way to accomplish your goal because Terraform doesn't store provisioners' details in the state file, which makes it difficult to perform troubleshooting. So, in this case, you can use some third-party tools such as CircleCI or Jenkins to perform some operations or run any script. This can also help you to get more detailed debug logs.

Terraform provisioner use cases

Let's try to understand some of the use cases of Terraform provisioners. There could be many use cases, totally depending on different scenarios. A few of them are as follows:

- Loading data into a virtual machine
- Bootstrapping a virtual machine for a config manager
- Saving data locally on your system

The `remote-exec` provisioner connects to a remote machine via WinRM or SSH and helps you to run a script remotely. The remote machine should allow remote connection; otherwise, the Terraform `remote-exec` provisioner will not be able to run the provided script. Instead of using `remote-exec` to pass data to a virtual machine, most cloud providers provide built-in tools to pass data, such as the `user_data` argument in AWS or `custom_data` in Azure. All of the public clouds support some sort of data exchange that doesn't require remote access to the machine; for further reading about built-in tools to pass data in different clouds, you can refer to <https://www.terraform.io/docs/language/resources/provisioners/syntax.html>. A few of them are shown here:

- Alibaba Cloud: `user_data` on `alicloud_instance` or `alicloud_launch_template`.
- Amazon EC2: `user_data` or `user_data_base64` on `aws_instance`, `aws_launch_template`, and `aws_launch_configuration`.
- Amazon Lightsail: `user_data` on `aws_lightsail_instance`.
- Microsoft Azure: `custom_data` on `azurerm_virtual_machine` or `azurerm_virtual_machine_scale_set`.
- Google Cloud Platform: `metadata` on `google_compute_instance` or `google_compute_instance_group`.
- Oracle Cloud Infrastructure: `metadata` or `extended_metadata` on `oci_core_instance` or `oci_core_instance_configuration`.
- VMware vSphere: Attach a virtual CDROM to `vsphere_virtual_machine` using the `cdrom` block, containing a file called `user-data.txt`.

Figure 4.1 – Cloud data passing options

As you can see in the figure, different data exchange arguments are supported by different cloud providers. This would help you to find data that you are looking for on those resources during the creation or update. There are many Linux OS images, such as Red Hat, Ubuntu, and SUSE, that have an built-in software called `cloud-init` that allows you to run arbitrary scripts and perform some basic system configuration during the initial deployment of the server itself, and because of this, you won't be required to take SSH of the server explicitly.

In addition to the `remote-exec` provisioner, there are also configuration management provisioners for *Chef*, *Puppet*, *Salt*, and so on. They allow you to bootstrap the virtual machine to use your config manager of choice. One of the best alternatives is to create a custom image with the config manager software already installed and get it to register with your config management server at bootup using one of the data loading options mentioned in the previous paragraph. We are now going to have a detailed discussion about the types of Terraform provisioners.

Terraform provisioner types

Now that we have some understanding about Terraform provisioners, let's see what different types of provisioners are available to us.

The local-exec provisioner

Using the `local-exec` provisioner, you will run the Terraform configuration code locally to extract some information. In some cases, there is a provider that already has the functionality you're looking for. For instance, a local provider can interact with files on your local system. Still, there could be a situation where you would be required to run a local script using the `local-exec` provisioner. HashiCorp recommends using `local-exec` as a temporary workaround if the feature that you are looking for is not available in the provider.

The following is a code snippet that would help you to define the `local-exec` code block in your Terraform configuration code:

```
resource "aws_instance" "example" {  
  # ...  
  
  provisioner "local-exec" {  
    command = "echo The EC2 server IP address is ${self.  
private_ip}"  
  }  
}
```

In the previously defined code block, if you notice, we have defined a `local-exec` provisioner code block inside the resource code block of the AWS instance. This would help you to extract the server IP address when it gets created. We have defined an object named `self` that represents the provisioner's parent resource and has the ability to extract all of that resource's attributes. In our case, we have defined `self.public_ip`, which is referencing `aws_instance's public_ip` attribute.

Suppose you are trying to run the `local-exec` provisioner and it is failing. Then, let's see how you can define the code block handling the failure behavior of the provisioner. The `on_failure` argument can be used with all the provisioners:

```
resource "aws_instance" "example" {
  # ...

  provisioner "local-exec" {
    command    = "echo The EC2 server IP address is ${self.
private_ip}"
    on_failure = continue
  }
}
```

In the preceding code, we have defined `on_failure = continue`. You have two supported values for `on_failure` arguments: either `continue` or `fail`:

- `continue`: Continuing with creation or destruction by ignoring the error, if any.
- `fail`: If there is any error, stop applying the default behavior. If this happens with the creation provisioner, it will taint the resource.

You might be wondering, what will happen to the provisioner when you're looking to destroy your resources? The following is a code sample of how you can define when, an argument, with the value `destroy` (that is, `when = destroy`):

```
resource "google_compute_instance" "example" {
  # ...

  provisioner "local-exec" {
    when      = destroy
    command = "echo We are discussing Destroy-time provisioner"
  }
}
```

Destroy provisioners are run before the resource is destroyed. If it fails, Terraform will throw an error and rerun the provisioners again when you perform `terraform apply` the next time. Due to this behavior, you need to be careful when running destroy provisioners multiple times.

Destroy-time provisioners can only run if they exist in the configuration file at the time when a resource is being destroyed. If you remove the resource code block with a destroy-time provisioner, its provisioner configurations will also be removed along with it and because of that, you won't be able to run the destroy provisioner.

Let's see how we can define multiple provisioners. The following code block will give you an idea of how to define multiple provisioners:

```
resource "google_compute_instance" "example" {
  # ...
  provisioner "local-exec" {
    command = "echo We have executed first command
successfully"
  }
  provisioner "local-exec" {
    command = "echo We have executed second command
successfully"
  }
}
```

From the previously defined code block, we can understand that we can define as many provisioner code blocks inside the resource code block as we wish.

Let's take one more example of `local-exec`, where we will show you how you can define a PowerShell script that is in your local directory and how you can run that script by defining it in a `null_resource` code block:

```
resource "null_resource" "script" {
  triggers = {
    always_run = "${timestamp()}"
  }
  provisioner "local-exec" {
    command      = "${path.module}/script.ps1"
    interpreter  = ["powershell", "-File"]
  }
}
```

Suppose you are willing to export some output of the resource to the local file. Then, you can write the following code block. As you can see from this code block, we would be able to get the AWS instance's private IP address to our local `private_ips.txt` filename and that will be created in the local directory:

```
resource "aws_instance" "example" {
  # ...
  provisioner "local-exec" {
    command = "echo ${aws_instance.example.private_ip} >>
private_ips.txt"
  }
}
```

For more information about the `local-exec` provisioner, you can read <https://www.terraform.io/docs/provisioners/local-exec.html>.

The file provisioner

Now that you understand Terraform provisioners, there is one other type of provisioner, named the file provisioner, which helps us to copy files or directories from the machine where Terraform is being executed to the newly created resource. The file provisioner supports both types of connections, that is, `ssh` and `winrm`. The following code block will help you to understand how you can use file provisioners:

```
resource "aws_instance" "example" {
  # ...

  # Copies the app.conf file to /etc/app.conf
  provisioner "file" {
    source      = "conf/app.conf"
    destination = "/etc/app.conf"
  }

  # Copies the string in content into /tmp/amifile.log
  provisioner "file" {
    content      = "ami used: ${self.ami}"
    destination = "/tmp/amifile.log"
  }

  # Copies the configs.d folder to /etc/configs.d
  provisioner "file" {
```

```

    source      = "conf/configs.d"
    destination = "/etc"
  }
  # Copies all files and folders in apps/apptest to D:/IIS/
  webapp
  provisioner "file" {
    source      = "apps/apptest/"
    destination = "D:/IIS/webapp"
  }
}

```

In the previously defined code block, you can see that some of the arguments are supported by file provisioners:

- **source:** This is the source folder or file that you want to be copied. You can define a source relative to the current working directory or as an absolute path. You won't be able to specify a source attribute with the content attribute. Either source or content can be defined in file provisioners.
- **content:** This attribute will help you to copy content to its destination. If the destination is a file, the content will get written on that file. In the case of a directory, a new file named `tf-file-content` is created. It is recommended to use a file as the destination so that the content is copied into that defined file. You will not be able to use this attribute with the source attribute.
- **destination (required):** An absolute path that you want to copy to.

A *file provisioner* can be used with the defined nested connection code block. The following provisioner code will give you an insight into how you can define connection details such as `ssh` or `winrm` in the code itself:

```

# Copies the file as the root user using SSH
provisioner "file" {
  source      = "conf/app.conf"
  destination = "/etc/app.conf"
  connection {
    type      = "ssh"
    user      = "root"
    password  = var.root_ssh_password
    host      = var.host_name
  }
}

```

```
}  
# Copies the file as the Administrator user using WinRM  
provisioner "file" {  
  source      = "conf/app.conf"  
  destination = "C:/App/app.conf"  
  connection {  
    type      = "winrm"  
    user      = "TerraformAdmin"  
    password  = var.windows_admin_password  
    host      = var.host_name  
  }  
}
```

This whole code can be defined in the *resource* or the *provisioner*. If you want more information about connections within the *provisioner*, then you can refer to <https://www.terraform.io/docs/provisioners/connection.html>.

The remote-exec provisioner

The `remote-exec` provisioner helps you to run a script on the remote resource once it is created. You can use the `remote-exec` provisioner to run a configuration management tool, bootstrap into a cluster, and so on. It also supports both `ssh` and `winrm` type connections. The following code block will give you an idea of how you can define the `remote-exec` provisioner:

```
resource "aws_instance" "example" {  
  # ...  
  provisioner "remote-exec" {  
    inline = [  
      "puppet apply",  
      "consul join ${aws_instance.example.private_ip}",  
    ]  
  }  
}
```

The previously defined code block has the following arguments:

- `inline`: This is a list of commands that are executed in sequence the way you would define them in the code block. This cannot be written with `script` or `scripts`.

- `script`: This is the relative path of a local script that will be copied to the remote resource and then executed. This cannot be combined with `inline` or `scripts`.
- `scripts`: This is a list of relative paths to local scripts that are copied to the remote resource and then executed. It is used to execute in the order you have defined in the code block. You will not be able to define this with `inline` or `script`.

For further reading about `remote-exec` of provisioners, you can visit <https://www.terraform.io/docs/language/resources/provisioners/remote-exec.html>.

In this section, we discussed Terraform provisioners. You will now understand different types of provisioners, such as file provisioners, local provisioners, and remote provisioners. We also mentioned that you should choose to use a provisioner only when you don't have any other option. In a nutshell, a Terraform provisioner helps you to run some sort of script either locally or remotely in order to achieve some goal. In the upcoming section, we are going to discuss different types of loops supported by Terraform so that you will be able to define a code block for meeting a certain N number of requirements.

Understanding Terraform loops

In this section, we will be discussing different methods of **Terraform loops**. Like other programming languages, Terraform also supports some sorts of loops and this will help you to perform N number of Terraform operations very smoothly. Let's try to understand this with an example. Suppose you are working with your colleague, *Mark*, in one of the **multinational companies (MNCs)**. You are both discussing one of the requirements – the need to deploy an Azure virtual network with 10 different subnets along with a **Network Security Group (NSG)** associated with all these subnets. You tell Mark that this can be easily done using Terraform loops rather than writing the same code block for the subnet again and again.

We will explain how effectively we can write our Terraform configuration code block using Terraform loops. The following loops are supported by Terraform:

- `count`: Looping over resources
- `for_each`: Looping over resources and inline blocks within a resource
- `for`: Looping over defined lists and maps

Let's look at each of them in detail.

The count expression

Let's understand how we can use count parameters. For a better understanding, we have the following Terraform code block, which will help us to create a resource group in Azure:

```
# To Create Resource Group
resource "azurerm_resource_group" "example" {
  name      = "Terraform-rg"
  location = "West Europe"
}
```

Now, let's suppose that you want to create three resource groups in Azure. Then, how would you achieve this requirement? You might be planning to use a traditional loop approach that you might have seen in other programming languages, but Terraform doesn't support that approach:

```
# This pseudo code will not work in Terraform.
for (i = 0; i < 3; i++) {
  resource "azurerm_resource_group" "example" {
    name      = "Terraform-rg"
    location = "West Europe"
  }
}
```

As you know, this previously defined code block will not work in Terraform. Now, the question is, how do we define this loop in Terraform? You can use the count parameter to create three resource groups in Azure. The following code snippet will be able to perform that job for us:

```
# To Create Resource Group
resource "azurerm_resource_group" "example" {
  count = 3
  name   = "Terraform-rg"
  location = "West Europe"
}
```

There is one problem in the previously defined code: the name of the resource group in Azure should be unique. You can use `count.index` to make the name of the resource group unique. The following code snippet will give you an idea of how to define `count` and `count.index`:

```
# To Create Resource Group
resource "azure_rm_resource_group" "example" {
  count = 3
  name   = "Terraform-rg${count.index}"
  location = "West Europe"
}
```

If you are running `terraform plan`, you can see the following code snippet, which clearly shows that it will be provisioning three different resource groups in Azure when you run the `terraform apply` command:

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but
will not be
persisted to local or remote state storage.
-----
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create
Terraform will perform the following actions:
# azure_rm_resource_group.example[0] will be created
+ resource "azure_rm_resource_group" "example" {
  + id          = (known after apply)
  + location    = "westeurope"
  + name        = "Terraform-rg0"
}
# azure_rm_resource_group.example[1] will be created
+ resource "azure_rm_resource_group" "example" {
  + id          = (known after apply)
  + location    = "westeurope"
  + name        = "Terraform-rg1"
```



```

    }
    # azurerm_resource_group.example[2] will be created
    + resource "azurerm_resource_group" "example" {
      + id          = (known after apply)
      + location    = "westeurope"
      + name        = "Terraform-rg2"
    }
  }
}
Plan: 3 to add, 0 to change, 0 to destroy.

```

The Terraform-rg0, Terraform-rg1, and Terraform-rg3 resource group names don't look nice. We can give them our own defined names, rather than take them through `count.index`. We can perform this customization by defining a variable code block:

```

variable "rg_names" {
  description = "list of the resource group names"
  type        = list(string)
  default     = ["Azure-rg", "AWS-rg", "Google-rg"]
}

```

Now, as we have defined the names of the resource groups in the variable code block, our actual resource code block will look as follows:

```

# To Create Resource Group
resource "azurerm_resource_group" "example" {
  count = length(var.rg_names)
  name   = var.rg_names[count.index]
  location = "West Europe"
}

```

As you can see in the previously defined code block, we used a `length` function that returns the number of items in the given list. When you run the `terraform plan` command, you can see that Terraform wants to create three resource groups with the names `Azure-rg`, `AWS-rg`, and `Google-rg` in Azure:

```

$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but
will not be persisted to local or remote state storage.
-----
-----

```

```

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create
Terraform will perform the following actions:
# azurerm_resource_group.example[0] will be created
+ resource "azurerm_resource_group" "example" {
  + id          = (known after apply)
  + location    = "westeurope"
  + name        = "Azure-rg"
}
# azurerm_resource_group.example[1] will be created
+ resource "azurerm_resource_group" "example" {
  + id          = (known after apply)
  + location    = "westeurope"
  + name        = "AWS-rg"
}
# azurerm_resource_group.example[2] will be created
+ resource "azurerm_resource_group" "example" {
  + id          = (known after apply)
  + location    = "westeurope"
  + name        = "Google-rg"
}
Plan: 3 to add, 0 to change, 0 to destroy.

```

Note that as you have used `count` on the resource, it is no longer a single resource, but it became a list of resources. Since it is a list of resources, if you want to read an attribute from a specific resource, then you would be required to define it in the following way:

```
<PROVIDER>_<TYPE>.<NAME>[INDEX].ATTRIBUTE
```

Let's try to get the `id` attribute from the `Azure-rg` resource group. The following code block will provide us with the `id` attribute:

```

output "rg_id" {
  value          = azurerm_resource_group.example[0].id
  description    = "The Id of the resource group"
}

```

If you want the IDs of all the resource groups, you will be required to use a splat expression, `*`, instead of the index:

```
output "All_rg_id" {
  value      = azurerm_resource_group.example[*].id
  description = "The Id of all the resource group"
}
```

When you run the `terraform apply` command, `rg_id` will give you the output of the Azure-rg resource group and `All_rg_id` will provide you with the output of all the resource groups:

```
$ terraform apply
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
Terraform will perform the following actions:
Plan: 0 to add, 0 to change, 0 to destroy.
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Outputs:
All_rg_id = [
  "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/
resourceGroups/Azure-rg",
  "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/
resourceGroups/AWS-rg",
  "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/
resourceGroups/Google-rg",
]
rg_id = /subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/
resourceGroups/Azure-rg
```

We have just seen how we can define the `count` parameter, but there are some limitations to it and because of that, the use of the `count` parameter is reducing.

The first limit is when using `count` over an entire resource, you won't be able to use `count` within a resource to loop an inline code block. For example, say you are planning to create a virtual network in Azure and multiple subnets inside that virtual network. This subnet code block contains the properties and has its configuration. So, if we want to iterate over those properties, then we may not be able to use the `count` expression.

The second limitation with `count` is what will happen if you try to make any changes to your defined list. Let's continue with the same variable code that we defined for the resource groups:

```
variable "rg_names" {
  description = "list of the resource group names"
  type        = list(string)
  default     = ["Azure-rg", "AWS-rg", "Google-rg"]
}
```

Consider how you have removed `AWS-rg` from the list. So, our new variable code would look as follows:

```
variable "rg_names" {
  description = "list of the resource group names"
  type        = list(string)
  default     = ["Azure-rg", "Google-rg"]
}
```

Now, let's try to understand how exactly Terraform will behave when we run `terraform plan`:

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but
will not be
persisted to local or remote state storage.

-----
-----

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  - destroy
  -/+ destroy and then create replacement

Terraform will perform the following actions:

# azurerm_resource_group.example[1] must be replaced
-/+ resource "azurerm_resource_group" "example" {
  ~ id          = "/subscriptions/97c3799f-2753-40b7-a4bd-
157ab464d8fe/resourceGroups/AWS-rg" -> (known after apply)
    location = "westeurope"
```

```

    ~ name      = "AWS-rg" -> "Google-rg" # forces replacement
  - tags      = {} -> null
}

# azurerm_resource_group.example[2] will be destroyed
- resource "azurerm_resource_group" "example" {
  - id        = "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/Google-rg" -> null
  - location = "westeurope" -> null
  - name     = "Google-rg" -> null
  - tags     = {} -> null
}

Plan: 1 to add, 0 to change, 2 to destroy.

```

With `terraform plan`, you can see that it is going to destroy `Google-rg` and recreate it. But this isn't something that we are looking to do; we don't want that `Google-rg` to be destroyed and recreated if we are removing `AWS-rg` from our variables list.

When you define the `count` parameter on a resource, Terraform treats that resource as a list or array of resources and because of that, it is used to identify each resource defined within the array by its positional index in that array. The internal representation of these resource groups looks something like this when you run `terraform apply`:

```

azurerm_resource_group.example[0]: Azure-rg
azurerm_resource_group.example[1]: AWS-rg
azurerm_resource_group.example[2]: Google-rg

```

When you remove an item from the middle of an array, all the items after it shift back by one, so after running `plan` with just two items, Terraform's internal representation will be something like this:

```

azurerm_resource_group.example[0]: Azure-rg
azurerm_resource_group.example[1]: Google-rg

```

You may have noticed how `Google-rg` has moved from index 2 to index 1 and because of this, Terraform understands to "rename the item at index 1 to `Google-rg` and delete the item at index 2." In other words, whenever, you use `count` in your code to provision a list of resources, and suppose you remove any of the middle items from the list, Terraform will recreate all those resources that come after that middle item again from scratch, which is something we are definitely not looking for. To solve these limitations of the `count` parameter, Terraform provides `for_each` expressions.

The `for_each` expression

Terraform has come up with the `for_each` expression, which helps you to loop over a set of strings or maps. By using `for_each`, you can create multiple copies of the entire resource or multiple copies of the inline code block, which is defined inside the resource code block.

Let's try to understand how we can write a `for_each` code block just for the resource code block. For our reference, we are going to consider the same example of creating three resource groups in Azure:

```
# To Create Resource Group
resource "azurerm_resource_group" "example" {
  for_each = toset(var.rg_names)
  name     = each.value
  location = "West Europe"
}
```

We have used the `toset` Terraform function to convert the `var.rg_names` list into a set of strings. `for_each` only supports a set of strings and maps in the resource code block. If we use a `for_each` loop over this set, then `each.value` or `each.key` will provide the names of the resource groups. Generally, `each.key` is used for maps where you have key-value pairs.

Once we have defined `for_each` on a resource code block, it becomes a map of the resource rather than a single resource. In order to see what exactly we are talking about, let's define an output variable named `all_rg` and an input variable named `rg_names` with default values:

```
output "all_rg" {
  value = azurerm_resource_group.example
}
variable "rg_names" {
  description = "list of the resource group names"
  type        = list(string)
  default     = ["Azure-rg", "AWS-rg", "Google-rg"]
}
```

Here is what you can expect when you run the `terraform apply` command:

```
$ terraform apply
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
Terraform will perform the following actions:
  # azurerm_resource_group.example["AWS-rg"] will be created
  + resource "azurerm_resource_group" "example" {
    + id          = (known after apply)
    + location   = "westeurope"
    + name       = "AWS-rg"
  }
  # azurerm_resource_group.example["Azure-rg"] will be created
  + resource "azurerm_resource_group" "example" {
    + id          = (known after apply)
    + location   = "westeurope"
    + name       = "Azure-rg"
  }
  # azurerm_resource_group.example["Google-rg"] will be created
  + resource "azurerm_resource_group" "example" {
    + id          = (known after apply)
    + location   = "westeurope"
    + name       = "Google-rg"
  }
Plan: 3 to add, 0 to change, 0 to destroy.
Outputs:
all_rg = {
  "AWS-rg" = {
    "id" = "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/AWS-rg"
    "location" = "westeurope"
    "name" = "AWS-rg"
    "tags" = {}
  }
  "Azure-rg" = {
```

```

    "id" = "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/Azure-rg"
    "location" = "westeurope"
    "name" = "Azure-rg"
    "tags" = {}
  }
  "Google-rg" = {
    "id" = "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/Google-rg"
    "location" = "westeurope"
    "name" = "Google-rg"
    "tags" = {}
  }
}

```

As you can see, Terraform created three resource groups in Azure and the `all_rg` output variable has outputted a map where the keys represent the resource group name. Let's suppose you are looking for the ID with the name `all_id` in all the respective resource groups. Then, you would be required to do something extra; that is, you would be required to define `values`, which is a Terraform built-in function, to extract exact values. The Terraform output code can be defined in this way:

```

output "all_id" {
  value = values(azurerm_resource_group.example) [*].id
}

```

Now, when you run the `terraform apply` command again, you can expect the following output:

```

$ terraform apply
(...)
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:
all_id = [
  "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/AWS-rg",
  "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/Azure-rg",

```



```

"/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/
resourceGroups/Google-rg",
]

```

The best thing about the `for_each` expression is you are getting a map of the resource rather than a list of resources that you were getting with `count`. Let's try to remove the same middle resource group name, that is, `AWS_rg`, from the input variable and then try to run the `terraform plan` command. You can expect the following code snippet:

```

$ terraform plan
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  - destroy
Terraform will perform the following actions:

# azurerm_resource_group.example["AWS-rg"] will be destroyed
- resource "azurerm_resource_group" "example" {
  - id          = "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/AWS-rg" -> null
  - location   = "westeurope" -> null
  - name       = "AWS-rg" -> null
  - tags       = {} -> null
}

Plan: 0 to add, 0 to change, 1 to destroy.

Changes to Outputs:
~ all_id = [
  - "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/AWS-rg",
    "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/Azure-rg",
    "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/Google-rg",
  ]

```

As you can see from `terraform plan`, it is only deleting the `AWS_rg` resource group name without touching any other resource groups. That means the challenge that we had in the `count` expression can be overcome by using a `for_each` expression.

Let's try to understand how we can use a `for_each` expression for the inline code block inside the resource code block. Dynamically, we will try to have multiple subnets within the virtual network in Azure. The following code block will give you insight into how you can define multiple subnets within a virtual network using a `for_each` expression:

```
resource "azurerm_resource_group" "example" {
  name      = "Terraform-rg"
  location = "West Europe"
}

resource "azurerm_virtual_network" "vnet" {
  name                = var.vnet_name
  location             = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  address_space       = var.address_space

  dynamic "subnet" {
    for_each = var.subnet_names
    content {
      name                = subnet.value.name
      address_prefix     = subnet.value.address_prefix
    }
  }
}

variable "subnet_names" {
  default = {
    subnet1 = {
      name                = "subnet1"
      address_prefix     = "10.0.1.0/24"
    }
    subnet2 = {
      name                = "subnet2"
      address_prefix     = "10.0.2.0/24"
    }
  }
}

variable "vnet_name" {
  default = "terraform-vnet"
}
```

```
variable "address_space" {  
  default = ["10.0.0.0/16"]  
}
```

If you run the `terraform apply` command, you will find that it will create a new virtual network with two subnets. Similarly, if you want to have multiple subnets inside virtual network resources, you can use a `dynamic for_each` expression:

```
$ terraform apply  
(...)  
# azurerm_virtual_network.vnet will be created  
+ resource "azurerm_virtual_network" "vnet" {  
  + address_space      = [  
    + "10.0.0.0/16",  
  ]  
  + guid               = (known after apply)  
  + id                 = (known after apply)  
  + location           = "westeurope"  
  + name               = "terraform-vnet"  
  + resource_group_name = "Terraform-rg"  
  + subnet             = [  
    + {  
      + address_prefix = "10.0.1.0/24"  
      + id              = (known after apply)  
      + name            = "subnet1"  
      + security_group = ""  
    },  
    + {  
      + address_prefix = "10.0.2.0/24"  
      + id              = (known after apply)  
      + name            = "subnet2"  
      + security_group = ""  
    },  
  ]  
}
```

In the previous code block, we showed how you can use a dynamic block expression, which is used to produce one or more nested blocks. The subnet label name lets Terraform know that this will be a set of nested blocks of each type of subnet. Within the block, we have to provide some data to use for the creation of the nested blocks. We have stored our subnet configuration in the variable named `subnet_names`. So, we have seen the `for_each` loop. Now, let's try to understand the `for` expression.

The for expression

We have discussed the `count` and `for_each` loop expressions in Terraform. Now, let's try to understand the `for` expression. You can use the `for` expression for both `list` and `map`. The following is the syntax for defining the `for` expression in a code block:

```
[for <ITEM> in <LIST> : <OUTPUT>]
[for <KEY>, <VALUE> in <MAP> : <OUTPUT>]
```

Here, `ITEM` is the local variable name assigned to each item in `LIST` and `LIST` is the list through which you will be iterating. `OUTPUT` is a transformed `ITEM`.

Similarly, for a map, `KEY` and `VALUE` are the local variable names assigned to each key-value pair in `MAP`. `MAP` is the map through which you will be iterating, and `OUTPUT` is a transformed expression from `KEY` and `VALUE`.

The following code block helps to give you an idea of how you can define a `for` expression in a list:

```
variable "cloud" {
  description = "A list of cloud"
  type        = list(string)
  default     = ["azure", "aws", "gcp"]
}

output "cloud_names" {
  value = [for cloud_name in var.cloud : upper(cloud_name)]
}
```

If you run the `terraform apply` command, you can expect the following:

```
$ terraform apply
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
Changes to Outputs:
+ cloud_names = [
```

```
+ "AZURE",  
+ "AWS",  
+ "GCP",  
]
```

Let's see how to define the `for` expression in the code block for a map. In this code block, we have defined a variable named `cloud_map` and provided a default key and values:

```
variable "cloud_map" {  
  description = "map"  
  type        = map(string)  
  default = {  
    Azure = "Microsoft"  
    AWS   = "Amazon"  
    GCP   = "Google"  
  }  
}  
  
output "cloud_mapping" {  
  value = [for cloud_name, company in var.cloud_map : "${cloud_name} cloud is founded by ${company}"]  
}
```

When you run the `terraform apply` command, you can expect the following:

```
$ terraform apply  
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.  
Outputs:  
cloud_mapping = [  
  "AWS cloud is founded by Amazon",  
  "Azure cloud is founded by Microsoft",  
  "GCP cloud is founded by Google",  
]
```

In this section, we discussed the different available loops in Terraform, such as `count`, `for_each`, and `for`. Now you will understand how to iterate in Terraform configuration code. In the upcoming section, we will discuss the different available built-in functions in Terraform that you can use to transform your requirements into configuration code as and when needed.

Understanding Terraform functions

There are many built-in **Terraform functions** that you can use to transform data into the format you might be looking to consume in the respective configuration code. Often, it may be that the format of the data returned by the data source and resource is not compatible with the data format that you need for passing to other resources. Terraform provides built-in functions in the following categories:

- Numeric
- String
- Collection
- Encoding
- Filesystem
- Date and time
- Hash and crypto
- IP network
- Type conversion

It is not possible to discuss all the Terraform built-in functions here. If you are interested in knowing about all the Terraform functions, you can refer to <https://www.terraform.io/docs/configuration/functions.html>. It is most important for you to understand how to use functions and how to test the Terraform function using the Terraform console.

If you are already familiar with any programming language, then defining a Terraform function works the same. The basic format for writing a Terraform function is `function(arguments, ...)`.

Let's see a few of the Terraform function syntaxes:

- `max(number1, number2, number3, ...)`: To determine the highest number from the given numbers
- `lower("text")`: To convert a string to lower

Each Terraform function takes specific types of arguments and returns a value of a specific type. You can even combine multiple functions together to get the desired data type, for example, `abs(max(-1, -10, -5))`. Here, in this example, the `max` function will return the highest value out of the given numbers, and then the `abs` function will provide you with the absolute value of that output.

By using the Terraform console, you can test all the Terraform functions. You can simply open any CLI, such as the CMD terminal of Windows, and then type `terraform console`. Remember one thing: the Terraform console never makes any changes to the state file. If you have opened a terminal in the current directory where you have the Terraform configuration file, then it will load the current state so that you get access to the state data and you can test functions accordingly.

Let's try to understand this with an example:

```
variable "address_space" {
  type    = list(string)
  default = ["10.0.0.0/16"]
}

resource "azurerm_resource_group" "azure-rg" {
  name     = "Terraform-rg"
  location = "eastus"
}

resource "azurerm_virtual_network" "vnet" {
  name                = "prod-vnet"
  location             = azurerm_resource_group.azure-rg.location
  resource_group_name = azurerm_resource_group.azure-rg.name
  address_space       = var.address_space
  subnet {
    name                = "subnet1"
    address_prefix     = cidrsubnet(var.address_space[0], 8, 1)
  }
}
```

In the previously defined code, we have defined a virtual network address space value using a variable called `address_space`, and then in order to calculate the subnet `address_prefix` value, we use the `cidrsubnet` Terraform function.

Let's see how we can test the value of `address_prefix` using the Terraform console. In order to perform that testing, you just need to open any CLI and run the `terraform console` command, which will load the configuration file and provide the following output:

```
$ terraform console
> var.address_space[0]
10.0.0.0/16
> cidrsubnet(var.address_space[0],8,1)
10.0.1.0/24
```

As you can see, the Terraform console helps us to test the Terraform built-in functions. So, whenever you are planning to consume any of the Terraform built-in functions, you can test them and define them accordingly in the configuration code block.

In this section, we have learned about the Terraform built-in functions and how we can use Terraform functions effectively, and we even got to know how we can test a Terraform function using the Terraform console. Next, we will discuss the Terraform debugging option and get an understanding of how to use the Terraform debugging options.

Understanding Terraform debugging

Terraform provides multiple options for debugging. You can get detailed logs of Terraform by enabling an environment variable named `TF_LOG` to any value. `TF_LOG` supports any of the following log levels: `TRACE`, `DEBUG`, `INFO`, `WARN`, or `ERROR`. By default, the `TRACE` log level is enabled, which is the recommended one by Terraform because it provides the most detailed logs.

If you want to save these logs to a certain location, then you can define `TF_LOG_PATH` in the environment variable of Terraform and point it to the respective location where you want to save your log file. Just remember that in order to enable the log, you need to use `TF_LOG` with any of the earlier described log levels, such as `TRACE` or `DEBUG`. To set the `TF_LOG` environment variable, you can use the following:

```
export TF_LOG=TRACE
```

Similarly, if we want to set `TF_LOG_PATH`, then we can do so in this way:

```
export TF_LOG_PATH=./terraform.log
```


Suppose while running the Terraform configuration code you find that Terraform crashes suddenly. Then, it will save a log file called `crash.log`, which would have all the debug logs of the session as well as the panic message and backtrace. As Terraform end users, these log files are of no use to us. You need to pass on these logs to the developer via GitHub's issues page (<https://github.com/hashicorp/terraform/issues>). However, if you are curious to see what went wrong with your Terraform, then you can check the panic message and backtrace, that will be holding information related to the issue. You should see something like this:

```
panic: runtime error: invalid memory address or nil pointer
dereference
goroutine 123 [running]:
panic(0xabc100, 0xd93000a0a0)
    /opt/go/src/runtime/panic.go:464 +0x3e6
github.com/hashicorp/terraform/builtin/providers/aws.
resourceAwsSomeResourceCreate(...) /opt/gopath/src/github.
com/hashicorp/terraform/builtin/providers/aws/resource_aws_
some_resource.go:123 +0x123
github.com/hashicorp/terraform/helper/schema.(*Resource).
Refresh(...) /opt/gopath/src/github.com/hashicorp/terraform/
helper/schema/resource.go:209 +0x123
github.com/hashicorp/terraform/helper/schema.(*Provider).
Refresh(...) /opt/gopath/src/github.com/hashicorp/terraform/
helper/schema/provider.go:187 +0x123
github.com/hashicorp/terraform/rpc.(*ResourceProviderServer).
Refresh(...)
    /opt/gopath/src/github.com/hashicorp/terraform/rpc/
resource_provider.go:345 +0x6a
reflect.Value.call(...)
    /opt/go/src/reflect/value.go:435 +0x120d
reflect.Value.Call(...)
    /opt/go/src/reflect/value.go:303 +0xb1
net/rpc.(*service).call(...)
    /opt/go/src/net/rpc/server.go:383 +0x1c2
created by net/rpc.(*Server).ServeCodec
    /opt/go/src/net/rpc/server.go:477 +0x49d
```

The first two lines hold key information that involves `hashicorp/terraform`. See the following example:

```
github.com/hashicorp/terraform/builtin/providers/aws.  
resourceAwsSomeResourceCreate(...)  
  
/opt/gopath/src/github.com/hashicorp/terraform/builtin/  
providers/aws/resource_aws_some_resource.go:123 +0x123
```

The first line tells us which method failed, which in this example is `resourceAwsSomeResourceCreate`. With that, we can understand that something went wrong during AWS resource creation.

The second line tells you the exact line of code that caused the panic. This panic message is enough for a developer to quickly figure out the cause of the issue.

As a user, this information will help you to perform some level of troubleshooting whenever you observe a Terraform crash.

Important note

While uploading Terraform logs for any issue on GitHub (<https://github.com/hashicorp/terraform/issues>), ensure that it doesn't contain any confidential data or secrets.

In this section, we learned about Terraform logs, which you can collect by setting different trace levels to `TF_LOG`, and you can save these log files by defining `TF_LOG_PATH` in Terraform environment variables. We also discussed the `crash.log` file, which you can send to the developer using a GitHub issue so that they can help you out.

Summary

In this chapter, you learned about the Terraform backend configuration, which is used to store the Terraform state file, provisioners, which are mainly used to run some sort of script, and built-in functions, which are used to transform data into the desired form so that it can be used in other code configuration blocks. Moving on, we learned how you can perform different kinds of iteration using `for` and other loops in Terraform, which are helpful when you are looking to provision multiple resources at a time. Then, finally, we discussed Terraform debugging, where you can set the `TF_LOG` Terraform environment variable to some log level and store it in the provided destination accordingly by defining the Terraform environment variable, such as `TF_LOG_PATH`.

In the next chapter, we will discuss the authentication of Terraform to *Azure*, *AWS*, and *GCP*. Moving on, we will also be discussing the use of the Terraform CLI, where we will see how we can run different Terraform workflow commands and the respective output from them.

Questions

1. Which of the following environment variables needs to be set to get Terraform detailed logs?
 - A) `TF_LOG`
 - B) `TF_TRACE`
 - C) `TF_DEBUG`
 - D) `TF_INFO`
2. What is the result of the following Terraform function?
`max(2, 5, 10, - 10)`
 - A) 2
 - B) 5
 - C) 10
 - D) -10
3. The Terraform "backend" is used to keep the Terraform state file and helps with Terraform operations. Which of the following is not a supported backend type?
 - A) S3
 - B) Artifactory
 - C) `azurerm`
 - D) GitHub
4. You need to execute some PowerShell script on the virtual machine that you are planning to create using Terraform configuration code. Which of the following properties would you use?
 - A) Terraform resource
 - B) Terraform data source
 - C) Terraform provisioner (`local-exec` or `remote-exec`)
 - D) Terraform function

5. You have been asked to create multiple subnets with a specific attribute inside an Azure virtual network. Which of the following iteration methods can help you to achieve this goal?
- A) The `count` expression
 - B) The `for` expression
 - C) The `if` expression
 - D) A dynamic expression

Further reading

You can check out the following links for more information about the topics that were covered in this chapter:

- Terraform backends: <https://www.terraform.io/docs/backends/index.html>
- Terraform provisioners: <https://www.terraform.io/docs/provisioners/index.html>
- Terraform functions: <https://www.terraform.io/docs/configuration/functions.html>
- Terraform debugging: <https://www.terraform.io/docs/internals/debugging.html>
- Terraform loops: <https://www.hashicorp.com/blog/hashicorp-terraform-0-12-preview-for-and-for-each>
- Terraform GCS backend: <https://www.terraform.io/docs/language/settings/backends/gcs.html>

5

Terraform CLI

In the previous chapter, we discussed the **Terraform backend**, **Terraform provisioner**, **Terraform loops (iterations)**, and **Terraform built-in functions**. In this chapter, we are going to discuss the **Terraform command-line interface (Terraform CLI)**, which is an open source software maintained and supported by HashiCorp. We will further see how we can authenticate the Terraform CLI with major cloud providers such as **Azure**, **Amazon Web Services (AWS)**, and **Google Cloud Platform (GCP)**. In the later sections, we will discuss some of the Terraform CLI commands and try to understand the importance of each command.

The following topics will be covered in this chapter:

- Introduction to the Terraform CLI
- Integrating with Azure
- Integrating with AWS
- Integrating with GCP
- Understanding the Terraform CLI commands

Technical requirements

To follow this chapter, you need to have an understanding of writing Terraform configuration code that includes providers, resources, variables, data, output, provisioners, and functions. Along with this, some basic knowledge of major cloud providers such as *GCP*, *AWS*, and *Azure* would be of additional benefit throughout the entire course. You can find all the code used in this chapter at the following GitHub link: <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter5>.

Check out the following link to see the Code in Action video:

<https://bit.ly/36re4es>

Introduction to the Terraform CLI

The Terraform CLI is an open source command-line application provided by HashiCorp that allows you to run different commands and subcommands. The main commands that cover the Terraform workflows are `init`, `plan`, and `apply`. You can run the subcommands `flag` after the main commands. In order to see a list of the commands supported by the Terraform CLI, you can simply run `terraform` on any terminal and you will see the following output:

```
$ terraform
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init          Prepare your working directory for other
  commands
  validate      Check whether the configuration is valid
  plan          Show changes required by the current
  configuration
  apply         Create or update infrastructure
  destroy       Destroy previously-created infrastructure

All other commands:
```

<code>console command prompt</code>	Try Terraform expressions at an interactive
<code>fmt</code>	Reformat your configuration in the standard style
<code>force-unlock</code>	Release a stuck lock on the current workspace
<code>get</code>	Install or upgrade remote Terraform modules
<code>graph operation</code>	Generate a Graphviz graph of the steps in an operation
<code>import</code>	Associate existing infrastructure with a Terraform resource
<code>login</code>	Obtain and save credentials for a remote host
<code>logout</code>	Remove locally-stored credentials for a remote host
<code>output</code>	Show output values from your root module
<code>providers configuration</code>	Show the providers required for this configuration
<code>refresh</code>	Update the state to match remote systems
<code>show</code>	Show the current state or a saved plan
<code>state</code>	Advanced state management
<code>taint functional</code>	Mark a resource instance as not fully functional
<code>untaint instance</code>	Remove the 'tainted' state from a resource instance
<code>version</code>	Show the current Terraform version
<code>workspace</code>	Workspace management
Global options (use these before the subcommand, if any):	
<code>-chdir=DIR</code>	Switch to a different working directory before executing the given subcommand.
<code>-help</code>	Show this help output, or the help for a specified subcommand.
<code>-version</code>	An alias for the "version" subcommand.

If you wish to see supported commands and subcommands, then you can get these by adding a `-h` flag to the main command. This will open the help menu of that specific command. For example, to see subcommands for a Terraform graph, run the following command:

```
$ terraform graph -h
Usage: terraform graph [options] [DIR]

  Outputs the visual execution graph of Terraform resources
  according to configuration files in DIR (or the current
  directory if omitted).

  The graph is outputted in DOT format. The typical program
  that can read this format is GraphViz, but many web services
  are also available to read this format.

  The -type flag can be used to control the type of graph
  shown. Terraform creates different graphs for different
  operations. See the options below for the list of types
  supported. The default type is "plan" if a configuration is
  given, and "apply" if a plan file is passed as an argument.

Options:
  -draw-cycles      Highlight any cycles in the graph with
                   colored edges. This helps when diagnosing cycle errors.
  -type=plan        Type of graph to output. Can be: plan, plan-
                   destroy, apply, validate, input, refresh.
  -module-depth=n  (deprecated) In prior versions of Terraform,
                   specified the depth of modules to show in the output.
```

Suppose you are running Terraform commands using `bash` or `zsh` as the command shell and looking for the `Tab` completion option. In that case, you can install that specific feature by running the following command:

```
terraform -install-autocomplete
```

This would help you out in terms of completing your commands or subcommands whenever you press the `Tab` key on the command shell.

We now have an understanding of the Terraform CLI and how we can use it to see a list of all the commands it supports. Let's now try to understand how we can authenticate the Terraform CLI to Azure.

Integrating with Azure

As you now know more about Terraform CLI, moving further on, we are going to talk in this section about how you can integrate or authenticate Terraform to Azure using Terraform CLI. In order to provision or update Azure services using Terraform CLI, it is important that your version of the Terraform CLI should be able to talk to Azure. In *Chapter 2, Terraform Installation Guide*, we already described how you can install `terraform.exe` on your local machine. So, let's try to understand how Terraform CLI can talk to Azure.

Here are the methods by which you can authenticate your Terraform CLI to Azure:

- Authentication using the Azure CLI
- Authentication using a **Managed Service Identity (MSI)**
- Authentication using a Service Principal and a Client Certificate
- Authentication using a Service Principal and a Client Secret

It is difficult to discuss all the possible options relating to Terraform authentication with Azure as there are so many of these. So, in order to learn about all the different authentication options, you can read <https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs>.

We are now going to discuss one of the prominent authentication methods: using a Service Principal and a Client Secret.

Important note

Please make sure you have a valid Azure subscription to perform any service deployment in Azure. You can open a free tier account as well for learning purposes. For more information about Azure subscription sign-up, you can visit <https://azure.microsoft.com/en-in/free/>.

Authentication using a Service Principal and a Client Secret

To authenticate the Terraform CLI to Azure using a Service Principal and a Client Secret, perform the following steps:

1. Log in to the Azure portal at `http://portal.azure.com` using your Microsoft login ID and password.
2. Go to **Active Directory (AD)** through your Azure portal and then create a new app registration (Service Principal), as shown in the following screenshot:

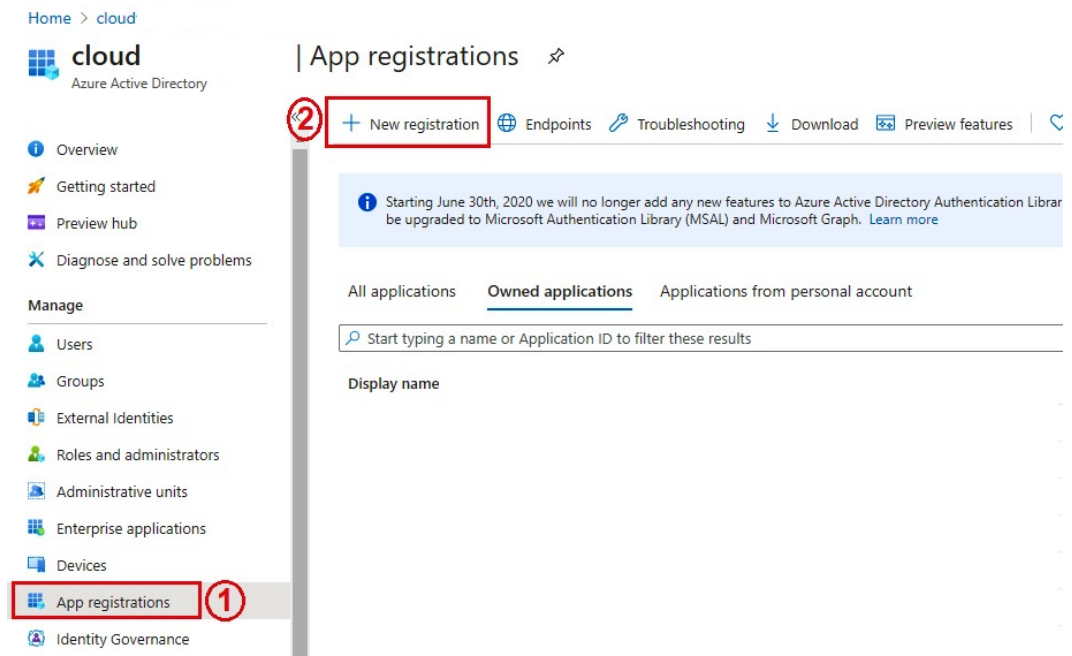


Figure 5.1 – Azure app registration

3. Provide the name of the app registration. We have chosen Terraform-Demo-SPN, as you can see in the following screenshot:

Register an application

* Name

The user-facing display name for this application (this can be changed later).

Terraform-Demo-SPN ✓

Supported account types

Who can use this application or access this API?

- Accounts in this organizational directory only (cloudtraininghub only - Single tenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)
- Personal Microsoft accounts only

[Help me choose...](#)

Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web e.g. https://myapp.com/auth

By proceeding, you agree to the [Microsoft Platform Policies](#)

Register

Figure 5.2 – New app registration

- Note down the **Application (client) ID** value and the **Directory (tenant) ID** value of the Service Principal, as illustrated in the following screenshot:

The screenshot shows the Azure portal interface for a Service Principal named "Terraform-Demo-SPN". The "Essentials" section displays the following information:

Display name	: Terraform-Demo-SPN	Supported account types	: My organization only
Application (client) ID	: 607907bc-e557-419e-81e1-a3e10e41644a	Redirect URIs	: Add a Redirect URI
Directory (tenant) ID	: abfa0b4e-cdda-49d0-8fc5-24f185a5c497	Application ID URI	: Add an Application ID URI
Object ID	: cdd6a247-f6aa-4abf-86ab-5c4ca086690c	Managed application in I...	: Terraform-Demo-SPN

The "Application (client) ID" and "Directory (tenant) ID" values are highlighted with a red box in the original image. Below the essentials section, there are informational messages and a "Call APIs" section with a "View API permissions" button. A "Documentation" section is also visible on the right side of the page.

Figure 5.3 – Service Principal

- Generate a Service Principal secret with any name. We have chosen **Terraform-Secret**, as you can see in the following screenshot:

Home > cloudtraininghub > Terraform-Demo-SPN

Terraform-Demo-SPN | Certificates & secrets

Search (Ctrl+/) << Got feedback?

Overview

Quickstart

Integration assistant | Preview

Manage

Branding

Authentication **1**

Certificates & secrets

Token configuration

API permissions

Expose an API

Owners

Roles and administrators | Preview

Manifest

Support + Troubleshooting

Troubleshooting

New support request

Add a client secret **3**

Description
Terraform-Secret

Expires

In 1 year

In 2 years

Never **4**

Add Cancel

Client secrets

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

+ New client secret **2**

Description	Expires	Value
No client secrets have been created for this application.		

Figure 5.4 – Service Principal secret

6. After adding a Service Principal secret, you will get the secret value, as shown in the following screenshot. Note down this value because it will be not available to you later:

Terraform-Demo-SPN | Certificates & secrets

Search (Ctrl+/) << Got feedback?

Overview

Quickstart

Integration assistant | Preview

Manage

Branding

Authentication

Certificates & secrets

Token configuration

API permissions

Expose an API

Owners

Roles and administrators | Preview

Manifest

Support + Troubleshooting

Troubleshooting

New support request

Copy the new client secret value. You won't be able to retrieve it after you perform another operation or leave this blade.

Credentials enable confidential applications to identify themselves to the authentication service when receiving tokens at a web addressable location (using an HTTPS scheme). For a higher level of assurance, we recommend using a certificate (instead of a client secret) as a credential.

Certificates

Certificates can be used as secrets to prove the application's identity when requesting a token. Also can be referred to as public keys.

Upload certificate

Thumbprint	Start date	Expires
No certificates have been added for this application.		

Client secrets

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

+ New client secret

Description	Expires	Value
Terraform-Secret	11/2/2021	JWT3efEWgPN.-bF2f0TEbKP7b-.B3_o-WH

Figure 5.5 – Service Principal secret key

- Provide Contributor access either at the subscription, resource group, or resource level, depending on whether you want to manage the following **identity and access management (IAM)** roles of Azure (<https://docs.microsoft.com/en-gb/azure/role-based-access-control/built-in-roles>). In our case, we have given Contributor access to the subscription level, as you can see in the following screenshot:

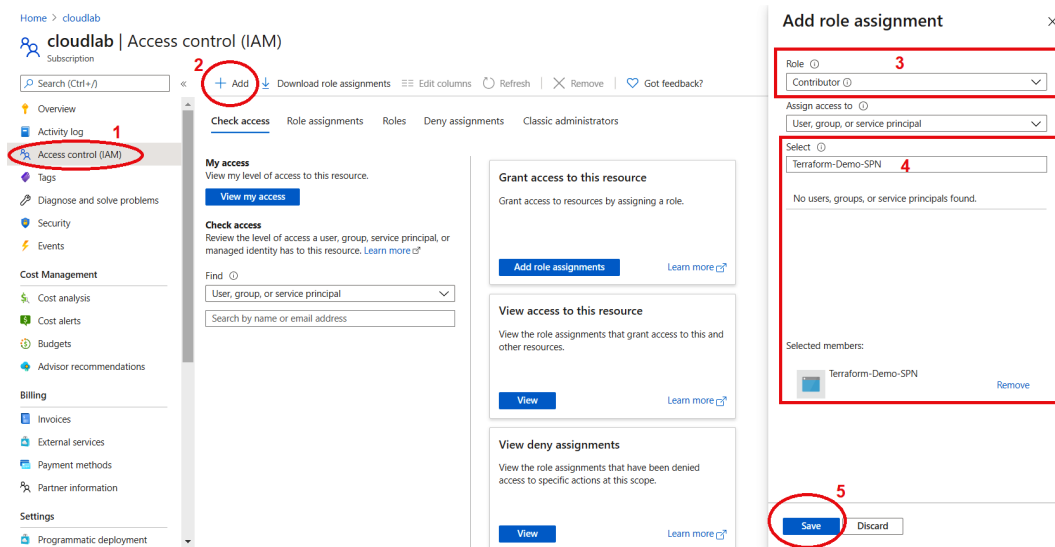


Figure 5.6 – Azure role-based access control (RBAC)

- As we have received a Service Principal credential, there are multiple ways of configuring this. We are going to place the Service Principal *client ID*, *tenant ID*, *subscription ID*, and *Client Secret* into the **Environment Variables** field of the local machine, which you can see in the following screenshot. You can refer to the **Azure Resource Manager (ARM)** provider block (<https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs>) to know about all the arguments supported:

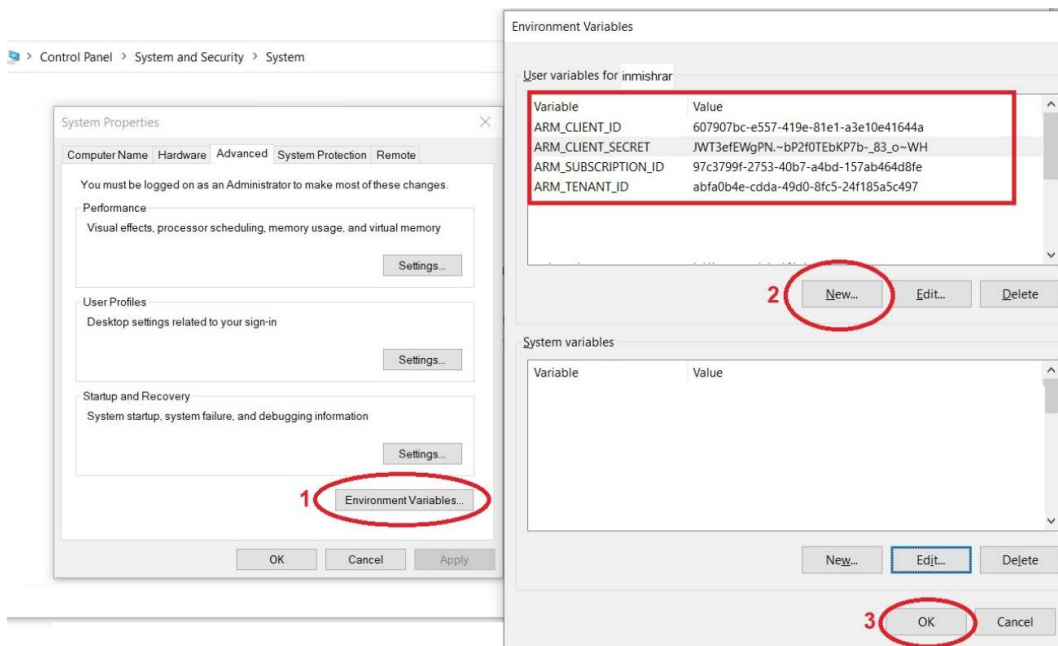


Figure 5.7 – Environment Variables

In our example, we have set environment variables in a local machine using a **user interface (UI)**, but it is strongly recommended to use any CLI, such as PowerShell, to store credentials in local environment variables in the following specified way:

```
$env:ARM_CLIENT_ID=". . ."
$env:ARM_CLIENT_SECRET=". . ."
$env:ARM_SUBSCRIPTION_ID=". . ."
$env:ARM_TENANT_ID=". . ."
```

- Now, you can write any file with a name ending with `.tf` that would be holding the Terraform configuration file. We have created a `providers.tf` file and placed the following code block within it:

```
terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "=2.55.0"
    }
  }
}
```



```

provider "azurerm" {
  features {}
}

```

10. You can open any CLI and run a `terraform init` command, which will create a folder named `.terraform`. This will contain a subfolder with a named plugin, and within that there will be an `azurerm` provider, which you can see in the following structure:

```

.
├── .terraform
│   ├── environment
│   └── providers
│       ├── registry.terraform.io
│       │   ├── hashicorp
│       │   └── azurerm
│       │       ├── 2.55.0
│       │       └── windows_amd64
│       └── terraform-provider-
│           └── azurerm_v2.55.0_x5.exe
├── .terraform.lock.hcl
└── providers.tf
7 directories, 4 files

```

Important note

Please don't refer to the Service Principal that has been used above in the Azure authentication section.

Provisioning Azure services using Terraform

We have already authenticated Terraform to Azure using a Service Principal. Let's try to create some Azure services using Terraform. We will try to create an Azure **virtual machine (VM)** using the Terraform code. The following code snippet contains a tree block where we have placed code for the VM with a key vault in the `main.tf` file. We have defined all the input variables in a `variables.tf` file and we are passing all the input variable values from a `terraform.tfvars` file:

```

inmishrar@terraform-vm:~/code# tree

```

```

.

```

```

├─ main.tf
├─ providers.tf
├─ terraform.tfvars
└─ variables.tf

0 directories, 4 files

```

You can get the full code block inside the `main.tf` file from the GitHub link presented in the *Technical requirements* section. Here is an excerpt of this:

```

resource "azurerm_resource_group" "rgname" {
  name      = var.rgname
  location  = var.location
  tags      = var.tags
}

resource "azurerm_virtual_network" "vnet" {
  name                = var.vnet_name
  address_space       = var.address_space
  location             = azurerm_resource_group.rgname.location
  resource_group_name = azurerm_resource_group.rgname.name
  tags                = var.tags
}

resource "azurerm_subnet" "subnet" {
  name                = var.subnet_name
  resource_group_name = azurerm_resource_group.rgname.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = [cidrsubnet(var.address_
space[0], 8, 1)]
}

...

```

This full code block inside the `variables.tf` file is available on the GitHub link presented in the *Technical requirements* section. You can take it directly from there, but an excerpt is provided here:

```

variable "rgname" {
  type      = string
  description = "name of resource group"
}

```

```
}  
variable "location" {  
  type      = string  
  description = "location name"  
}  
variable "vnet_name" {  
  type      = string  
  description = "vnet name"  
}  
...  
}
```

The code block inside `terraform.tfvars` looks like this:

```
rgname          = "Terraform-rg"  
location        = "West Europe"  
tags = {  
  Environment = "prod"  
  Owner       = "Azure-Terraform"  
}  
vm_size         = "Standard_F2"  
vm_name         = "Terraform-vm"  
admin_username  = "azureterraform"  
vm_publisher    = "MicrosoftWindowsServer"  
vm_offer        = "WindowsServer"  
vm_sku          = "2016-Datacenter"  
vm_version      = "latest"  
sku_name        = "premium"  
vnet_name       = "Terraform-vnet"  
address_space   = ["10.1.0.0/16"]  
subnet_name     = "Terraform-subnet"  
nic_name        = "Terraform-nic"  
keyvault_name   = "Terraform-keyvault2342"  
keyvault_secret_name = "Terraform-vm-password"
```

When you run `terraform init`, `plan`, and `apply`, it will then create the Azure services, as shown in the following screenshot:

The screenshot shows the Azure portal interface for a resource group named 'Terraform-rg'. The left sidebar contains navigation options like Overview, Activity log, Access control (IAM), Tags, Events, Settings, Quickstart, Deployments, Policies, Properties, Locks, Cost Management, Cost analysis, Cost alerts (preview), and Budgets. The main content area shows the 'Essentials' section with details for the resource group, including Subscription (cloudlab), Subscription ID (97c3799f-2753-40b7-a4bd-157ab464d8fe), and Tags (Environment: prod, Owner: Azure-Terraform). Below this is a table of resources:

Name	Type	Location
Terraform-keyvault2342	Key vault	West Europe
Terraform-nic	Network interface	West Europe
Terraform-vm	Virtual machine	West Europe
Terraform-vm_OsDisk_1_e1e8282ab6d946089a034552e0bbd062	Disk	West Europe
Terraform-vnet	Virtual network	West Europe

Figure 5.8 – Azure services

We have managed to provision Azure resources using Terraform. This way, you can provision new resources in Microsoft Azure, as well as update resources using the Terraform configuration file provided, if you have the state of those resources in the `tfstate` file.

In this section, we discussed how you can authenticate Terraform to Azure, and then we discussed how to write a Terraform configuration file to provision resources in Azure. While creating a configuration file, we mentioned how you create separate files such as `providers.tf`, `main.tf`, and `variables.tf`, and we explained how to take input variable values using `terraform.tfvars`.

In our upcoming section, we will be discussing the authentication of Terraform to AWS and will accordingly try to provision some resources in AWS in the same way we did in Azure.

Integrating with AWS

As we discussed earlier how you can authenticate Terraform to Azure, let's now try to understand how we can authenticate Terraform with AWS. Here are the methods that allow Terraform to get authenticated in AWS:

- Static credentials
- Environment variables
- Shared credentials/configuration file
- CodeBuild, **Elastic Container Service (ECS)**, and **Elastic Kubernetes Service (EKS)** roles
- **Elastic Compute Cloud (EC2) Instance Metadata Service (IMDS)**, and IMDSv2

For detailed information about how you can authenticate Terraform to AWS, you can visit <https://registry.terraform.io/providers/hashicorp/aws/latest/docs>. Discussing all of the methods is beyond the scope of this book, but we will nevertheless discuss one of the methods and show you how you can provision services in AWS.

Authentication using an access key ID and secret

To authenticate using an access key ID and secret, perform the following steps:

1. Log in to your AWS console at <https://console.aws.amazon.com/> using your IAM/Root login ID and password.
2. Go to **My Security Credentials** and create a new access key, as illustrated in the following screenshot:

Search for services, features, marketplace products, and docs [Alt+S] cloudtraininghub

Your Security Credentials

Use this page to manage the credentials for your AWS account. To manage credentials for AWS Identity and Access Management (IAM) users, use the [IAM Console](#).

To learn more about the types of AWS credentials and how they're used, see [AWS Security Credentials](#) in AWS General Reference.

- ▲ Password
- ▲ Multi-factor authentication (MFA)
- ▼ Access keys (access key ID and secret access key)

Use access keys to make programmatic calls to AWS from the AWS CLI, Tools for PowerShell, AWS SDKs, or direct AWS API calls. You can have a maximum of two access keys (active or inactive) at a time.

For your protection, you should never share your secret keys with anyone. As a best practice, we recommend frequent key rotation.

If you lose or forget your secret key, you cannot retrieve it. Instead, create a new access key and make the old key inactive. [Learn more](#)

Created	Access Key ID	Last Used	Last Used Region	Last Used Service	Status	Ac
Jun 27th 2021	AKIA5HKMGAO6R47X2ISU	N/A	N/A	N/A	Active	Make Inac
Nov 8th 2020	AKIAIMQ4GX4XAJR7ZS2A	2021-01-25 23:16 UTC+0530	us-east-1	ec2	Deleted	

2 Create New Access Key

Figure 5.9 – AWS access key

3. You can store `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` in the environment variable by executing the following commands on the Bash CLI:

```
export AWS_ACCESS_KEY_ID="AKIAIMQ4GX4XAJR7ZS2A"
export AWS_SECRET_ACCESS_KEY="qQi4E0DQItP2utPE..."
```

For setting environment variables on a Windows machine, you can refer to the *Integrating with Azure* section and follow that approach to set `AWS_ACCESS_KEY` and `AWS_SECRET_ACCESS_KEY`.

You can create a configuration file with any name ending with `.tf`. For simplicity, we have named this `providers.tf` and placed the following code block in it:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

provider "aws" {
  region = var.aws_region
}
```

4. You can run the `terraform init` command on any CLI to download the AWS provider and get it stored in the `.terraform` folder, which you can see in the following code snippet:

```
inmishrar@terraform-vm:~/code# tree -a
.
├── .terraform
│   ├── environment
│   └── providers
│       ├── registry.terraform.io
│       │   └── hashicorp
│       │       └── aws
│       │           └── 3.36.0
│       │               └── linux_amd64
```

```

| | | | | terraform-provider-aws_
v3.36.0_x5.exe
|— .terraform.lock.hcl
|— providers.tf
7 directories, 4 files

```

We have finally managed to download the AWS provider and can now provision or update AWS services using Terraform. Let's try to understand how we can write the Terraform configuration code for the AWS provider.

Provisioning AWS services using Terraform

You have managed to authenticate Terraform to AWS, so let's now see how we can provision AWS services using Terraform. It is not possible to explain all the AWS services available, so we are considering a very simple example of creating an AWS **virtual private cloud (VPC)**. If you wish to explore the Terraform resource code block for the AWS provider, you can refer to <https://registry.terraform.io/providers/hashicorp/aws/latest/docs>.

The following code block will help you to provision a VPC in AWS. In order to provision an AWS VPC, we created the following files and placed them in our GitHub repository:

```

inmishrar@terraform-vm:~/code# tree
.
|— main.tf
|— providers.tf
|— terraform.tfvars
|— variables.tf
0 directories, 4 files

```

In the `main.tf` file, we have kept the following code block:

```

resource "aws_vpc" "terraform_aws_vpc" {
  cidr_block      = var.cidr_block
  instance_tenancy = "default"
  tags = {
    Name = "terraform_aws_vpc"
  }
}

```

Similarly, in the `variables.tf` file, we have kept the following defined code:

```
variable "cidr_block" {
  description = "provide VPC range"
}
variable "aws_region" {
  description = "provide AWS region"
}
```

We are going to take user input values of the variables from `terraform.tfvars` as follows:

```
cidr_block = "10.0.0.0/16"
aws_region = "ap-southeast-1"
```

When you run the `terraform plan` and `apply` commands, this will create a new VPC in AWS, as shown in the following screenshot:

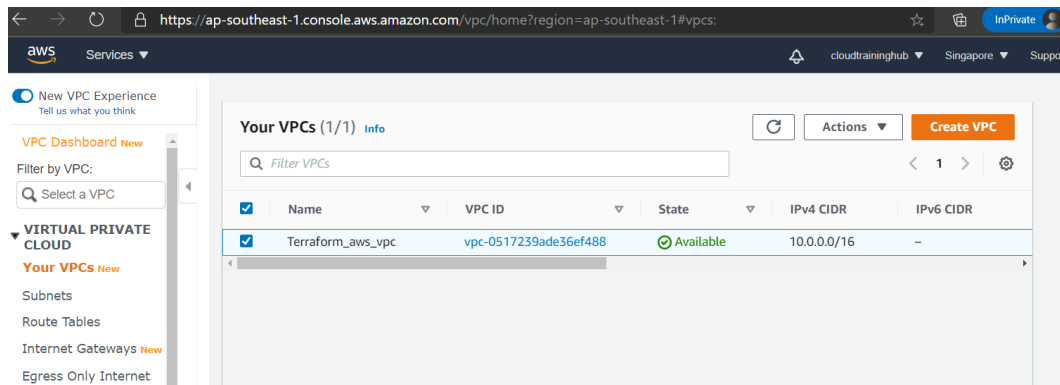


Figure 5.10 – AWS services

Finally, we managed to create an AWS VPC using the Terraform configuration code. Suppose tomorrow you want to create more services along with this VPC. You just need to define that resource or module code block in your Terraform configuration file, and Terraform will provision or update those particular services.

Important note

You should use an IAM user account rather than using a root account for integrating AWS with Terraform. For more information on IAM accounts, you can refer to https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html.

In this section, we discussed how you can authenticate Terraform to AWS, and then we explained how to write a Terraform configuration file to provision services in AWS by taking a simple example of an AWS VPC. In our next section, we will discuss the authentication of Terraform to GCP and, accordingly, we will try to create some services in GCP.

Integrating with GCP

As we have seen how we authenticate Terraform to Azure and AWS, we will similarly try to understand how Terraform can get authenticated to GCP.

There are multiple methods to authenticate Terraform to GCP, outlined as follows:

- Authentication using the Google Cloud **Software Development Kit (SDK) CLI**
- Authentication using a Google service account by storing credentials in a separate file
- Authentication using a Google service account by defining values in the environment variable
- Authentication using a Google service account by means of a valid token

Discussing all of the aforementioned methods is somewhat beyond the scope of this book, so we will just consider one method of authentication.

Authentication using a Google service account by storing credentials in a separate file

To authenticate a Google service account, perform the following steps:

1. Log in to Google Cloud Console at <https://console.cloud.google.com/> using your login ID and password.
2. Create a new project with any name. We have created a project with the name `Terraform-project`.

3. Create a service account by navigating to IAM and the admin page from the Google Cloud Console, as shown in the following screenshot:

The screenshot shows the Google Cloud Platform IAM Admin console. The browser address bar displays `https://console.cloud.google.com/iam-admin/serviceaccounts/create?project=terr`. The page title is "Create service account". The left sidebar contains navigation icons, with the IAM icon highlighted. The main content area is titled "1 Service account details" and contains the following fields:

- Service account name:** Terraform-gcp-account (Display name for this service account)
- Service account ID:** terraform-gcp-account @terraform-project-56745.iam.gserviceac (with X and refresh icons)
- Service account description:** To authenticate Terraform to GCP (Describe what this service account will do)

Below the form, there are three steps:

- 1 Service account details** (Current step)
- 2 Grant this service account access to project (optional)**
- 3 Grant users access to this service account (optional)**

At the bottom of the form, there are two buttons: **CREATE** (circled in red) and **DONE** (circled in red). The **DONE** button is accompanied by the text **CANCEL**.

Figure 5.11 – GCP service account

- Grant an `Editor` role for that service account to your GCP project so that it can perform read/write operations, as illustrated in the following screenshot:

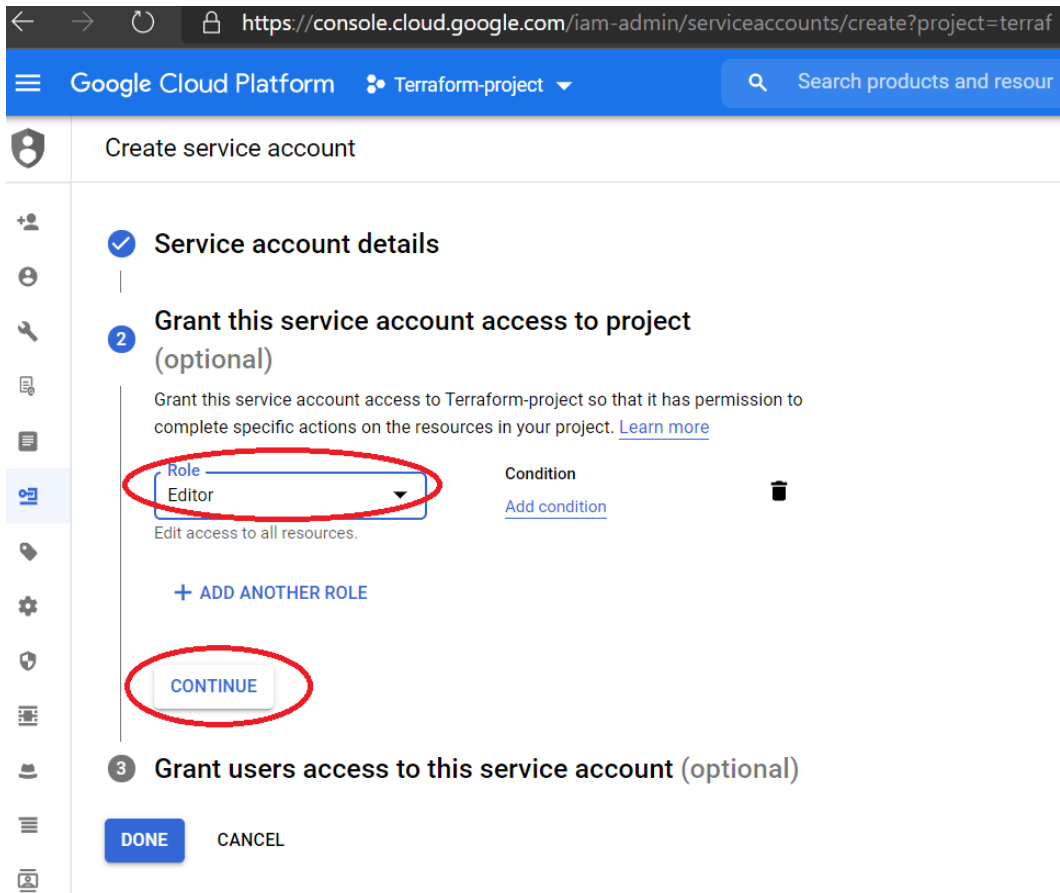


Figure 5.12 – GCP service account role

- Once you have created a service account in GCP, create a key in **JavaScript Object Notation (JSON)** format, as illustrated in the following screenshot:

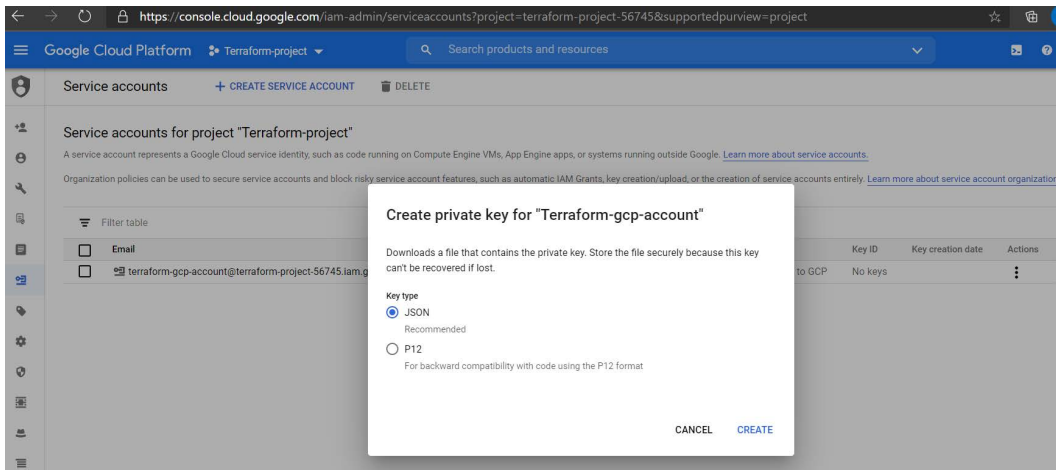


Figure 5.13 – GCP service account key

Important note

The GCP service account file contains credential information, so keep this secure as it can access services in GCP.

6. You can create a `providers.tf` file and define the following code block within it:

```
provider "google" {
  credentials = file("terraform-project-xxxx.json")
  project     = "Terraform-project"
  region     = "asia-south1"
}
```

7. You can run the `terraform init` command, which will download the Google provider plugin to the `.terraform` folder, as illustrated in the following code snippet:

```
inmishrar@terraform-vm:~/code# tree -a
.
├── .terraform
│   ├── environment
│   └── providers
│       ├── registry.terraform.io
│       │   └── hashicorp
│       │       └── google
```


In the `variables.tf` file, the following code is present:

```
variable "location_id"{
  type      = string
  description = "provide location name"
}
variable "project_id"{
  type      = string
  description = "provide Google Project ID"
}
```

In the `terraform.tfvars` file, we have kept the following code block:

```
project_id = "terraform-project-56745"
location_id = "asia-south1"
```

Now, when we run `terraform plan` and apply the command from the CLI, it will provision the App Engine service in the Google Cloud, which you can see in the following screenshot:

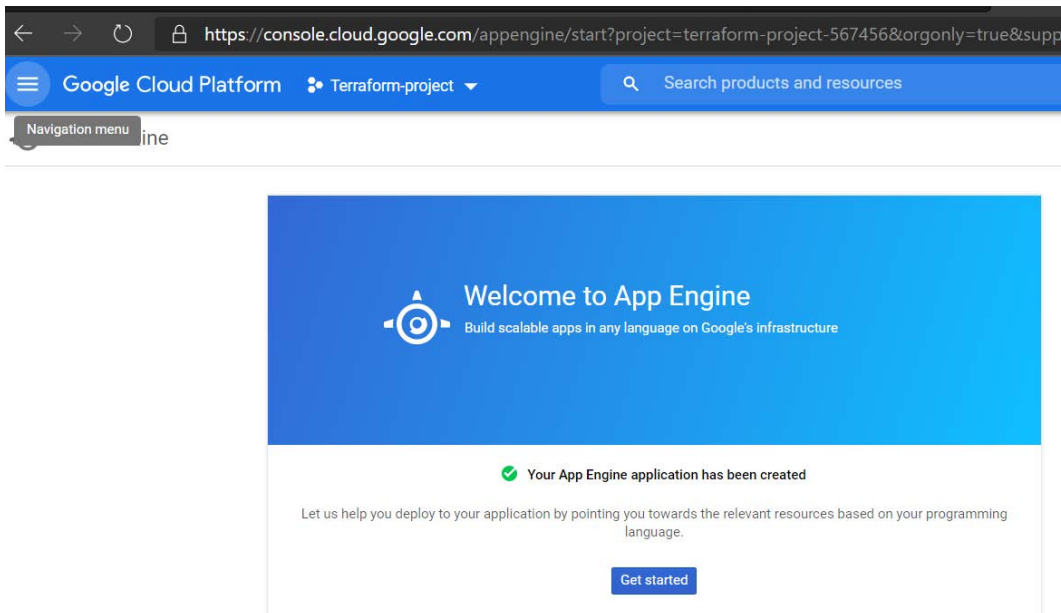


Figure 5.14 – GCP App Engine

We have finally managed to provision the GCP App Engine application service using the Terraform code. So, in future, if you are planning to make changes or add new infrastructure to Google Cloud, you can use Terraform **infrastructure as code (IaC)**. In our next section, we are going to learn about the different Terraform CLI commands and their use.

Understanding the Terraform CLI commands

The Terraform CLI supports many commands, and in this section we will try to cover some of the main commands and their respective outputs. We will discuss the Terraform workflows in our upcoming chapter, *Chapter 6, Terraform Workflows*, but we will now discuss some of the basic the Terraform CLI commands, as follows:

- `terraform console`: You can run this command on the CLI to open a Terraform console, where you can test or get the output of the code of certain Terraform functions. The following example shows how you can use this command:

```
$ terraform console
> max(5, 10, -5)
10
>
```

- `terraform fmt`: You can run this command to rewrite configuration files to a canonical format and style. This command performs some sort of adjustment so that your configuration code is in a readable format, and even helps you make some changes to follow Terraform's language-style conventions. To know more about Terraform language-style conventions, you can read <https://www.terraform.io/docs/configuration/style.html>. The following subcommand flags are supported by the `terraform fmt` command:

```
$ terraform fmt -h
Usage: terraform fmt [options] [DIR]

    Rewrites all Terraform configuration files to
    a canonical format. Both configuration files (.tf) and
    variables files (.tfvars) are updated. JSON files (.tf.
    json or .tfvars.json) are not modified.

    If DIR is not specified then the current working
    directory will be used.

    If DIR is "-" then content will be read from
    STDIN. The given content must be in the Terraform
    language native syntax; JSON is not supported.
```

Options:	
-list=false	Don't list files whose formatting differs
	(always disabled if using STDIN)
-write=false	Don't write to source files
	(always disabled if using STDIN or -check)
-diff	Display diffs of formatting changes
-check	Check if the input is formatted. Exit status will be 0 if all input is properly formatted and non-zero otherwise.
-no-color	If specified, output won't contain any color.
-recursive	Also process files in subdirectories. By default, only the given directory (or current directory) is processed.

- `terraform graph`: This command helps you to generate a visual representation of the Terraform configuration or execution plan. You can expect output in a DOT format. You can use Graphviz (<http://www.graphviz.org/>) to generate charts. This helps you to have a visual dependency graph of Terraform resources as per the defined configuration file in the DIR (or in the current directory). You can have the following subcommand flags with `terraform graph`:

\$ terraform graph -h
Usage: terraform graph [options] [DIR]
Outputs the visual execution graph of Terraform resources according to configuration files in DIR (or the current directory if omitted).
The graph is outputted in DOT format. The typical program that can read this format is GraphViz, but many web services are also available to read this format.
The <code>-type</code> flag can be used to control the type of graph shown. Terraform creates different graphs for different operations. See the options below for the list of types supported. The default type is "plan" if a configuration is given, and "apply" if a plan file is passed as an argument.
Options:

`-draw-cycles` Highlight any cycles in the graph with colored edges. This helps when diagnosing cycle errors.

`-type=plan` Type of graph to the output. Can be: `plan`, `plan-destroy`, `apply`, `validate`, `input`, `refresh`.

As you now know that the output of a Terraform graph is in DOT format, you can easily convert that into an image by using the `dot` command provided by Graphviz, as illustrated in the following diagram:

```
$ terraform graph | dot -Tsvg > graph.svg
```

Here is an example graph output:

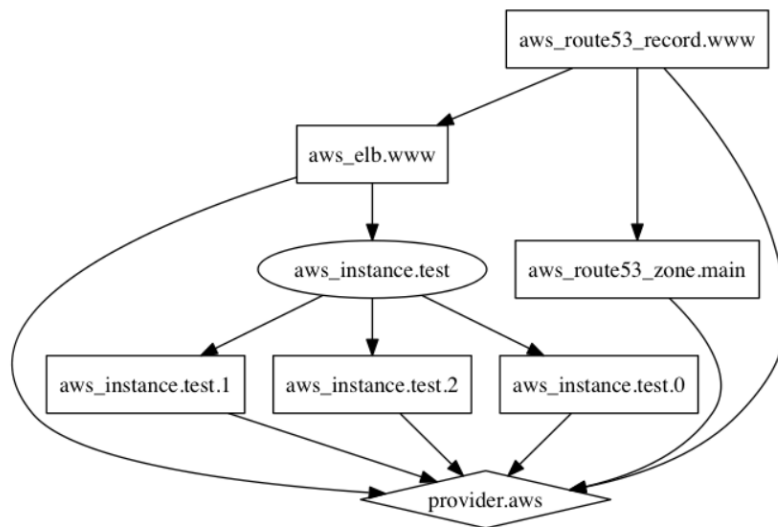


Figure 5.15 – Terraform graph

In *Figure 5.15*, you can see all the resources specific to the AWS cloud, and from this graph itself you can understand how Terraform is able to understand implicit dependency automatically. We will be discussing `depends_on` for defining explicit dependency in our upcoming chapter, *Chapter 7, Terraform Modules*.

- `terraform output`: This command yields the following output:

```
10.1.1.4
```

To understand how the preceding command works, you can read more details at <https://www.terraform.io/docs/commands/output.html>.

- `terraform refresh`: This command helps you to update the Terraform state file, comparing it with the real-world infrastructure. It doesn't make changes to the infrastructure directly, but your infrastructure may expect changes when you run the `terraform plan` and `apply` commands after `terraform refresh`. This command also helps to figure out whether there is any kind of drift from the last-known state and updates the state file accordingly. For a better understanding of this command, refer to the following link: <https://www.terraform.io/docs/commands/refresh.html>.
- `terraform show`: This command can help you see outputs on the CLI itself from the state file, and these will be in a human-readable format. If you are looking for specific information from your state file, you can use this command.
- `terraform taint`: Using this command, you can mark specific resources to get destroyed and recreated. Many times, if you encounter some sort of deployment error, then this command can be very useful. This command doesn't modify the infrastructure but it does perform changes to the state file, which means that if you are going to run `terraform apply`, it will destroy the specific resource that has been marked as tainted and recreate it. Let's see how we can taint a specific resource, as follows:

```
$ terraform taint aws_security_group.rdp_allow
```

The `aws_security_group.rdp_allow` resource has been marked as tainted, so this specific resource will get destroyed and recreated when we perform `terraform apply`.

You can read more about the `terraform taint` command at <https://www.terraform.io/docs/commands/taint.html>.

- `terraform workspace`: Terraform supports a command named `workspace` that helps you to create multiple landscapes. Suppose you want to use the same Terraform configuration in multiple landscapes. In that case, you can use this `workspace` command. For example, if you have three environments such as *dev*, *test*, and *production*, then by using this `workspace` command, you can create these virtual environments for Terraform's reference, and Terraform will maintain the state file and plugins for the respective environment accordingly. For more information about `terraform workspace`, you can read <https://www.terraform.io/docs/state/workspaces.html> and <https://www.terraform.io/docs/commands/workspace/index.html>.

- `terraform force-unlock`: Suppose your state file got locked by some other user in the remote backend provided. In that case, you could use `terraform force-unlock LOCK_ID [DIR]`. This command would help you to unlock your state file. This is only possible when you have stored your state file in a remote backend; if you have kept the state file in your local machine, then it can't be unlocked by any other process. One more thing: this command doesn't make any changes to your environment. Let's try to understand this with the following example, where the state file got locked by one of the users:

```
ID: f20d9093-5116-53c5-4037-ab2e8cb8f77d
Path: terraform
Operation: OperationTypeApply
Who: inmishrar1@tv-az29-815
Version: 1.0.0
Created: 2021-07-01 08:28:44.546798004 +0000 UTC
Info:
```

When a state file gets locked, if you try to run any Terraform command, it will not let you execute that command until you unlock the state file. We need to run the following commands to unlock a state file:

```
terraform force-unlock f20d9093-5116-53c5-4037-
ab2e8cb8f77d
```

- `terraform validate`: If you have some sort of syntax error in your configuration file, then, by using the `terraform validate` command, you can get detailed information about the error. This command should not be run individually because when you run the `terraform plan` or `apply` commands, Terraform automatically runs the `terraform validate` command in the backend to check any sort of syntax error within your configuration code block.
- `terraform import`: If you want to import some already existing resources or services into your Terraform state file, you can do this by referring to the resource ID of that specific resource. Let's try to understand how you can import a resource in the state file, with the following example:

```
terraform import aws_instance.abc i-acdf10234
```

In the previously defined command, we are trying to import an already existing AWS instance named `abc` into the state file, which has a resource ID of `i-acdf10234`.

We have now learned about different commands supported by the Terraform CLI. For more details about all the commands supported by the Terraform CLI, you can visit <https://www.terraform.io/docs/commands/index.html>.

Summary

From this complete chapter, you have received an understanding of the Terraform CLI. We also discussed how you can authenticate Terraform to Azure, AWS, and GCP. Moving ahead, we also explained different Terraform commands and subcommands supported by the Terraform CLI. After reading this entire chapter, you should be able to use the Terraform CLI for major cloud providers such as AWS, Azure, and GCP, helping you to provision and update resources in these cloud providers.

In our next chapter, we will be discussing Terraform workflows, where we will focus on the main commands of the Terraform CLI—that is, `init`, `plan`, `apply`, and `destroy`.

Questions

The answers to the following questions can be found in the *Assessments* section at the end of this book:

1. Which of the following subcommands would you use in order to unlock the Terraform state file?
 - A) `Unlock`
 - B) `force-unlock`
 - C) Removing the lock on a state file is not possible
 - D) `state-unlock`
2. Which Terraform command will force a marked resource to be destroyed and recreated on the next `apply`?
 - A) `terraform destroy`
 - B) `terraform refresh`
 - C) `terraform taint`
 - D) `terraform fmt`
3. What does the `terraform fmt` command do?
 - A) Deletes the existing configuration file
 - B) Rewrites Terraform configuration files to a canonical format and style
 - C) Updates the font of the configuration file to the official font supported by HashiCorp
 - D) Formats the state file in order to ensure the latest state of resources can be obtained

4. Person A has created one Azure virtual network in the Azure cloud using an ARM template. Now, you want to update this virtual network using a Terraform. Which command would you run to bring the Azure virtual network configurations into the Terraform state file?
 - A) `terraform import`
 - B) `terraform fmt`
 - C) `terraform output`
 - D) `terraform show`

5. Which Terraform command checks and reports errors within modules, attribute names, and value types to make sure they are syntactically valid and internally consistent?
 - A) `terraform validate`
 - B) `terraform show`
 - C) `terraform format`
 - D) `terraform fmt`

Further reading

You can check out the following links for more information on the topics covered in this chapter:

- The Terraform CLI: <https://www.terraform.io/docs/commands/index.html>
- Terraform for Azure: <https://learn.hashicorp.com/collections/terraform/azure-get-started>
- Terraform for AWS: <https://learn.hashicorp.com/collections/terraform/aws-get-started>
- Terraform for GCP: <https://learn.hashicorp.com/collections/terraform/gcp-get-started>

6

Terraform Workflows

In the previous chapter, we discussed the Terraform **command-line interface (CLI)** and saw some outputs from Terraform commands.

In this chapter, we will take a look at a core workflow of the Terraform tool, which involves creating a Terraform configuration file (`write`), previewing the changes (`plan`), then finally committing those changes to the target environment (`apply`). Once we are done with the creation of the resources, we might be required to get rid of infrastructures (`destroy`). In a nutshell, we are planning to cover Terraform core workflows, which mainly consist of `terraform init`, `terraform plan`, `terraform apply`, and `terraform destroy` operations, and the respective subcommands and their outputs. Understanding Terraform workflows will help you to provision and update infrastructure using Terraform **infrastructure as code (IaC)**. We will also be explaining the integration of Terraform workflows with major cloud providers such as Azure. Moving further, we are going to discuss Terraform workflows using Azure DevOps service.

The following topics will be covered in this chapter:

- Understanding the Terraform life cycle
- Understanding Terraform workflows using Azure DevOps

Technical requirements

To follow along with this chapter, you will need to have an understanding of the Terraform CLI and of various methods of authenticating Terraform to major cloud providers such as Azure, **Google Cloud Platform (GCP)**, and **Amazon Web Services (AWS)**. You should also know how to install Terraform on various machines and need to have a basic understanding of writing a Terraform configuration file. Some knowledge of Azure DevOps and Git would be an added advantage. You can find all the code used in this chapter at the following GitHub link: <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter6>.

Check out the following link to see the Code in Action video:

<https://bit.ly/3qVPyf8>

Understanding the Terraform life cycle

As you have become familiar with how to use the Terraform CLI and run the respective Terraform commands, you must have also been wondering how Terraform creates or updates an infrastructure. Terraform follows a sequence of commands that are defined under the Terraform life cycle. Let's try to understand how the Terraform life cycle works with `terraform init`, `terraform plan`, `terraform apply`, and `terraform destroy`. A Terraform workflow starts with writing the Terraform code file, downloading all the providers and plugins, displaying in preview which actions Terraform is going to perform, and then—finally—whether you wish to deploy the resources that have been defined in the Terraform configuration code file. After creating or updating this infrastructure, let's suppose you wish to have these resources destroyed—how would you do this? All this can be clarified by following a Terraform workflow, which is depicted in the following diagram:



Figure 6.1 – Terraform life cycle workflow

Terraform init

`terraform init` is the first and foremost Terraform command that you generally run to initialize Terraform in the working directory. There could be several reasons to run the `terraform init` command, such as the following:

- When you have added a new module, provider, or provisioner
- When you have updated the required version of a module, provider, or provisioner
- When you want to change or migrate the configured backend

Terraform modules (which we will be discussing in the upcoming chapter, *Chapter 7, Terraform Modules*) and these downloaded files get stored in the current working directory that you have provided. The `terraform init` command supports many subcommands or arguments. You can get all the supported arguments by typing `terraform init -h`, and here we share a few of them:

- `-backend=true`: Configures the backend for this configuration.
- `-backend-config=path`: This can either be a path to a **Hashicorp Configuration File (HCL)** file with key/value assignments (same format as `terraform.tfvars`) or a `key=value` format. This is merged with what is in the configuration file, which can be specified multiple times. The backend type must be in the configuration itself.
- `-from-module=SOURCE`: Copies the contents of a given module into the target directory before initialization.
- `-lock=true`: Locks the state file when locking is supported.
- `-no-color`: If specified, the output won't contain any color.
- `-plugin-dir`: Directory containing plugin binaries. This overrides all default search paths for plugins and prevents the automatic installation of plugins. This flag can be used multiple times.
- `-reconfigure`: Reconfigures the backend, ignoring any saved configuration.
- `-upgrade=false`: If installing modules (`-get`) or plugins (`-get-plugins`), ignores previously downloaded objects and installs the latest version allowed within configured constraints.
- `-verify-plugins=true`: Verifies the authenticity and integrity of automatically downloaded plugins.

Let's try to understand what exactly happens when we run the `terraform init` command with some simple example code. In this example, we are taking an **Azure Resource Manager (AzureRM)** provider. In the following code snippet, we have placed it in the `providers.tf` file:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "2.55.0"
    }
  }
}

provider "azurerm" {
  features {}
}
```

In terms of the actions performed when we run `terraform init`, if we look closely at the next code snippet, we can easily understand what `terraform init` has done. We can see that it has performed initialization of the backend to store the state file as well as downloading provider plugins from **HashiCorp**. Terraform will not be able to download third-party plugins. These can instead be manually installed in the user plugins directory, located at

`~/.terraform.d/plugins` on Linux/Mac operating systems and `%APPDATA%\terraform.d\plugins` on Windows. For detailed information about third-party plugins, you can refer to <https://www.hashicorp.com/blog/automatic-installation-of-third-party-providers-with-terraform-0-13>. You can see the code here:

```
$ terraform init
Initializing the backend...
Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "2.55.0"...
- Installing hashicorp/azurerm v2.55.0...
- Installed hashicorp/azurerm v2.55.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record
the provider selections it made above. Include this file in
your version control repository so that Terraform can guarantee
```

```
to make the same selections by default when you run "terraform
init" in the future.
```

```
Terraform has been successfully initialized!
```

```
You may now begin working with Terraform. Try running
"terraform plan" to see any changes that are required for your
infrastructure. All Terraform commands should now work. If
you ever set or change modules or backend configuration for
Terraform, rerun this command to reinitialize your working
directory. If you forget, other commands will detect it and
remind you to do so if necessary.
```

In the following code snippet, you can see that after running `terraform init`, this has created a `.terraform` folder in the current working directory, which contains `azurerm` provider plugins:

```
PS C:\terraform> tree
```

```
.
├── .terraform
│   ├── environment
│   └── providers
│       ├── registry.terraform.io
│       │   ├── hashicorp
│       │   └── azurerm
│       │       ├── 2.55.0
│       │       └── windows_amd64
│       └── terraform-provider-azurerm_
v2.55.0_x5.exe
├── .terraform.lock.hcl
└── providers.tf
```

Let's try to understand how we can perform Terraform initialization in the current local directory by providing a path of the Terraform files. In order to understand this, we have moved the `providers.tf` file to the provider folder and have then run `terraform init -backend-config='C:\provider'` from the current directory. This has downloaded all the provider plugins to the present working directory. The code is illustrated in the following snippet:

```
$ terraform init -backend-config='C:\provider'
```

```
Initializing the backend...
```

```
Initializing provider plugins...
```

```
- Finding hashicorp/azurerm versions matching "2.55.0"...
- Installing hashicorp/azurerm v2.55.0...
- Installed hashicorp/azurerm v2.55.0 (signed by HashiCorp)
```

```
Warning: Missing backend configuration
```

```
-backend-config was used without a "backend" block in the
configuration.
```

If you intended to override the default local backend configuration, no action is required, but you may add an explicit backend block to your configuration to clear this warning:

```
terraform {
  backend "local" {}
}
```

However, if you intended to override a defined backend, please verify that the backend configuration is present and valid.

```
Terraform has been successfully initialized!
```

You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.

As you can see, we are getting some sort of warning related to the backend configuration while performing `terraform init`. If we want to set the remote backend—for example, Azure Blob storage—then we can have the following code in the `providers.tf` file:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "2.55.0"
    }
  }
  backend "azurerm" {
    resource_group_name = "terraform-rg"
```

```
storage_account_name = "terraformstatestg"
container_name       = "tfstate-container"
key                  = "tfstate"
}
}
provider "azurerm" {
  features {}
}
```

This earlier defined code would help us in keeping the Terraform state file in the remote backend. If we don't define the remote backend, then Terraform would use the local backend as a default and store the state file in it.

We are not able to discuss all the subcommands of `terraform init`, which is why we have already shared a list of the subcommands/arguments and their respective descriptions at the beginning of this section so that you can get an idea of what exactly it can do for you.

Important note

The `terraform init` command is always safe to run multiple times. Though subsequent runs may give errors, this command will never delete your configuration or state.

Terraform validate

If you have some Terraform syntax errors in the configuration code, then you must be thinking: *Does Terraform provide any way to get these errors checked?* The answer to this is yes—Terraform has a built-in command, `terraform validate`, which will let you know if there are any syntax errors in the Terraform configuration code in the specified directory.

You can run the `terraform validate` command explicitly, otherwise validation of the configuration file will be done implicitly during the execution of the `terraform plan` or `terraform apply` commands, so it is not mandatory to run this command. Terraform is knowledgeable enough to run validation during other Terraform workflows such as `terraform plan` or `terraform apply` workflows.

The only thing needed before Terraform performs validation of the configuration code is the `terraform init` command, which downloads all plugins and providers by default. Here is a quick example, to give a walkthrough of some possible syntax validation errors:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    azurearm = {
      source = "hashicorp/azurearm"
      version = "2.55.0"
    }
  }
}

provider "azurearm" {
  features {}
}

resource "azurearm_resource_group" "example" {
  name          = "Terraform-lab-RG"
  location      = "eastus"
  terraform_location = "eastus"
}
```

If we run `terraform validate` against this configuration, we will get the following error:

```
PS C:\provider> terraform validate
Error: Unsupported argument
  on resourcegroup.tf line 16, in resource "azurearm_resource_group" "example":
   16:   terraform_location = "eastus"
An argument named "terraform_location" is not expected here.
```

This is showing that `terraform_location` is not a valid argument for `azurearm_resource_group`.

Let's see what will happen if we remove the `terraform_location` argument from the configuration code and then execute `terraform validate`. You can see the result in the following code snippet:

```
PS C:\provider> terraform validate
```

```
Success! The configuration is valid.
```

So, we can conclude that `terraform validate` can help us to perform a syntax check of the Terraform configuration code in the directory we are referencing.

Important note

The `terraform validate` command does not check Terraform configuration file formatting (for example, tabs versus spaces, newlines, comments, and so on). For formatting, you can use the `terraform fmt` command.

Terraform plan

After executing `terraform init`, you are supposed to run the `terraform plan` command, which would generate an execution plan. When the `terraform plan` command is being run, Terraform performs a refresh in the backend (unless you have explicitly disabled it), and Terraform then determines which actions it needs to perform to meet the desired state you have defined in the configuration files. If there is no change to the configuration files, then `terraform plan` will let you know that it is not performing any change to the infrastructure. Some of the subcommands/arguments that we can run with `terraform plan` are listed here—a complete list of the subcommands/arguments supported by the `terraform plan` phase can be seen by running the `terraform plan -h` command:

- `-destroy`: If set, a plan will be generated to destroy all resources managed by the given configuration and state.
- `-input=true`: Asks for input for variables if not directly set.
- `-out=path`: Writes a plan file to the given path. This can be used as input to the `apply` command.
- `-state=statefile`: Provides a path to a Terraform state file to use to look up Terraform-managed resources. By default, it will use the `terraform.tfstate` state file if it exists.
- `-target=resource`: Provides a resource to target. The operation will be limited to this resource and its dependencies. This flag can be used multiple times.

- `-var 'foo=bar'`: Sets a variable in the Terraform configuration. This flag can be set multiple times.
- `-var-file=foo`: Sets variables in the Terraform configuration from a file. If a `terraform.tfvars` files or any `.auto.tfvars` files are present, they will be automatically loaded.

In order to understand the `terraform plan` command, here is a line of code that's been defined in the `resourcegroup.tf` file:

```
resource "azurerm_resource_group" "example" {  
  name      = "Terraform-lab-RG"  
  location = "east us"  
}
```

When we run the `terraform plan` command, we can expect the following output:

```
PS C:\terraform> terraform plan  
Acquiring state lock. This may take a few moments...  
Refreshing Terraform state in-memory prior to plan...  
The refreshed state will be used to calculate this plan, but  
will not be persisted to local or remote state storage.  
-----  
-----  
An execution plan has been generated and is shown below.  
Resource actions are indicated with the following symbols:  
+ create  
Terraform will perform the following actions:  
# azurerm_resource_group.example will be created  
+ resource "azurerm_resource_group" "example" {  
  + id          = (known after apply)  
  + location    = "eastus"  
  + name       = "Terraform-lab-RG"  
}
```

Plan: 1 to add, 0 to change, 0 to destroy.

```
-----  
-----
```

Note: You didn't specify an "-out" parameter to save this plan, so Terraform can't guarantee that exactly these actions will be performed if "terraform apply" is subsequently run.

Releasing state lock. This may take a few moments...

From the previously defined output from the `terraform plan` command, we got to know all the resources Terraform is going to create or update. In our example, it is showing that it is going to create a resource group with the name `Terraform-lab-RG`.

Let's try to understand how to store the `terraform plan` output into any file. In order to store the `terraform plan` output, we need to run the `terraform plan -out <filename>` command, as follows:

```
PS C:\provider> terraform plan -out plan.txt
```

```
Acquiring state lock. This may take a few moments...
```

```
Refreshing Terraform state in-memory prior to plan...
```

```
The refreshed state will be used to calculate this plan, but
will not be persisted to local or remote state storage.
```

```
-----
-----
```

```
An execution plan has been generated and is shown below.
```

```
Resource actions are indicated with the following symbols:
```

```
+ create
```

```
Terraform will perform the following actions:
```

```
# azurerm_resource_group.example will be created
```

```
+ resource "azurerm_resource_group" "example" {
```

```
  + id          = (known after apply)
```

```
  + location   = "eastus"
```

```
  + name       = "Terraform-lab-RG"
```

```
}
```

```
Plan: 1 to add, 0 to change, 0 to destroy.
```

```
-----
-----
```

```
This plan was saved to: plan.txt
```

```
To perform exactly these actions, run the following command to
apply:
```

```
terraform apply "plan.txt"
```

```
Releasing state lock. This may take a few moments...
```


The terraform `plan -out plan.txt` command has created a `plan.txt` file in the local present working directory and will have a binary file of the terraform `plan` output, which you can see here in the expanded directory:

```
PS C:\provider> ls
Directory: C:\provider
Mode                LastWriteTime         Length Name
----                -
d-----           09-12-2020    03:53          .terraform
-a----           09-12-2020    16:41         2040 plan.txt
-a----           09-12-2020    03:53         316 providers.tf
-a----           09-12-2020    16:21         105
resourcegroup.tf
```

If we want to use this stored binary file (that is, `plan.txt`) during the terraform `apply` command, then we can run `terraform apply "plan.txt"`.

Moving on further, let's try to understand whether we need to pass a variable value input during the Terraform runtime or if we should pass a variable value from a local file.

In `resourcegroup.tf`, we have defined the following code:

```
resource "azurerm_resource_group" "example" {
  name      = var.rgname
  location = "eastus"
}
```

As we have defined `name = var.rgname`, we then need to declare a `rgname` variable that we have kept in a separate file, `variables.tf`, as follows:

```
variable "rgname" {
  description = "name of the resource group"
  type        = string
}
```

After running `terraform plan`, we can see in the following code snippet that Terraform is looking for the `rgname` value that we can provide during the runtime:

```
PS C:\provider> terraform plan
var.rgname
```

```
name of the resource group
```

```
Enter a value:
```

Once we provide the input, it will then proceed further and let us know which resource it is going to provision, as illustrated in the following code snippet:

```
PS C:\provider> terraform plan
```

```
var.rgname
```

```
name of the resource group
```

```
Enter a value: terraform-lab-rg
```

```
Refreshing Terraform state in-memory prior to plan...
```

```
The refreshed state will be used to calculate this plan, but
will not be persisted to local or remote state storage.
```

```
-----
-----
```

```
An execution plan has been generated and is shown below.
```

```
Resource actions are indicated with the following symbols:
```

```
+ create
```

```
Terraform will perform the following actions:
```

```
# azure_rm_resource_group.example will be created
```

```
+ resource "azure_rm_resource_group" "example" {
```

```
  + id          = (known after apply)
```

```
  + location    = "eastus"
```

```
  + name        = "terraform-lab-rg"
```

```
}
```

```
Plan: 1 to add, 0 to change, 0 to destroy.
```

```
-----
-----
```

```
Note: You didn't specify an "-out" parameter to save this plan,
so Terraform can't guarantee that exactly these actions will be
performed if "terraform apply" is subsequently run.
```

If we want to pass the variable value as input from any file, then we can do so. First of all, Terraform will look for any file ending with `.tfvars` or `.auto.tfvars` in the present working directory. If such a file is found, then Terraform will read the variable value from that file. Suppose you had defined a variable value in the `testing.txt` file—in that case, during the execution of the `terraform plan` command, you need to provide that file's pathname in this way: `terraform plan -var-file="testing.txt"`. We have defined a variable value in the `testing.txt` file—that is, `rgname = "terraform-lab-rg"`.

You will get the following code output when you run the aforementioned command:

```
PS C:\provider> terraform plan -var-file="testing.txt"
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but
will not be persisted to local or remote state storage.

-----
-----

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:
  # azurerm_resource_group.example will be created
  + resource "azurerm_resource_group" "example" {
    + id          = (known after apply)
    + location    = "eastus"
    + name        = "terraform-lab-rg"
  }

Plan: 1 to add, 0 to change, 0 to destroy.

-----
-----

Note: You didn't specify an "-out" parameter to save this plan,
so Terraform can't guarantee that exactly these actions will be
performed if "terraform apply" is subsequently run.
```

From this, we understand how `terraform plan` can take variable values in multiple ways. Terraform takes variable values in the following sequence:

```
Terraform CLI >> any filename/terraform.tfvars >> default value
provided in the variable declaration
```

We have already shared a list of the subcommands/arguments supported by the `terraform plan` command (with their description) at the very beginning of this topic, so we will not explain all the subcommands here. To read more about `terraform plan`, you can go to <https://www.terraform.io/docs/commands/plan.html>.

Terraform apply

After executing `terraform init` and `terraform plan`, if you find things are changing as per your expectations, you can then run `terraform apply` to help you provision or update the infrastructure. This will update the Terraform state file and will get it stored in the local or remote backend. Here is a list of the subcommand flags you can run with `terraform apply`—you can see this list by running the `terraform apply -h` command:

- `-auto-approve`: Skip interactive approval of the plan before applying.
- `-backup=path`: Provides a path to back up the existing state file before modifying it. Defaults to the `-state-out` path with a `.backup` extension. Set to `-` to disable backup.
- `-compact-warnings`: If Terraform produces any warnings that are not accompanied by errors, show them in a more compact form that includes only summary messages.
- `-input=true`: Asks for input for variables if not directly set.
- `-lock=true`: Locks the state file when locking is supported.
- `-lock-timeout=0s`: Duration to retry a state lock.
- `-no-color`: If specified, the output won't contain any color.
- `-parallelism=n`: Limits the number of concurrent operations. Defaults to 10.
- `-refresh=true`: Updates state prior to checking for differences.
- `-state=path`: Provides a path to read and save state (unless `state-out` is specified). Defaults to `terraform.tfstate`.
- `-state-out=path`: Provides a path to write state to that is different than `-state`. This can be used to preserve the old state.
- `-target=resource`: Provides a resource to target. The operation will be limited to this resource and its dependencies. This flag can be used multiple times.

- `-var 'foo=bar'`: Sets a variable in the Terraform configuration. This flag can be set multiple times.
- `-var-file=foo`: Sets variables in the Terraform configuration from a file. If a `terraform.tfvars` file or any `.auto.tfvars` files are present, these will be automatically loaded.

In continuation of the example from the `terraform plan` phase (that is, `testing.txt` is holding the Terraform variable value), let's try to understand how `terraform apply` will work with that example, as follows:

```
PS C:\provider> terraform apply -var-file="testing.txt"
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create
Terraform will perform the following actions:
# azurerm_resource_group.example will be created
+ resource "azurerm_resource_group" "example" {
  + id          = (known after apply)
  + location   = "eastus"
  + name       = "terraform-lab-rg"
}
Plan: 1 to add, 0 to change, 0 to destroy.
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
Enter a value:
```

Now, we can enter a `yes` value here to continue, or we can provide any value other than `yes` to cancel `terraform apply` and come out of the execution terminal. If we want that during the `terraform apply` phase, it shouldn't ask for further confirmation, and we can simply put `terraform apply -auto-approve`. In our case, we can define `terraform apply -var-file="testing.txt" -auto-approve`, to result in the following output:

```
azurerm_resource_group.example: Creating...
azurerm_resource_group.example: Creation complete after 3s
[id=/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/
resourceGroups/terraform-lab-rg]
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

As you can see, the preceding command didn't prompt for any confirmation and managed to create a resource group in Azure.

We saw all the subcommand flags supported by `terraform apply`, along with their descriptions, at the start of this section. You can read and check the output of each command on the Terraform CLI. To read more about `terraform apply`, you can visit <https://www.terraform.io/docs/commands/apply.html>.

Important note

You are not recommended to run the `terraform apply -auto-approve` command because many times, it is essential to check exactly what Terraform is going to perform, and we can cancel execution if we don't want to proceed with the `terraform apply` phase.

Terraform destroy

You might be wondering why we need to have `destroy` in the life cycle of the infrastructure. There could be cases where you want to get rid of the resources that you have provisioned using `terraform apply`, and in such cases you can run the `terraform destroy` command. This will delete all the resources or services you defined in the configuration file and update the state file accordingly. The `terraform destroy` command is a very powerful command, which is why when you execute it, it will present you with an execution plan for all resources it is going to delete and later ask for confirmation, because once the command gets executed it cannot be undone. Some of the subcommand flags or arguments supported by the `terraform destroy` command are listed here—you can get a full list of supported arguments by running `terraform destroy -h`:

- `-backup=path`: Provides a path to back up the existing state file before modifying it. Defaults to the `-state-out` path with a `.backup` extension. Set to `-` to disable backup.
- `-auto-approve`: Skip interactive approval before destroying.
- `-lock=true`: Locks the state file when locking is supported.
- `-refresh=true`: Update state file prior to checking for differences. This has no effect if a plan file is given to apply.
- `-state=path`: Provides a path to read and save state (unless `state-out` is specified). Defaults to `terraform.tfstate`.
- `-state-out=path`: Provides a path to write state to that is different from `-state`. This can be used to preserve the old state.

- `-target=resource`: Provides a resource to target. The operation will be limited to this resource and its dependencies. This flag can be used multiple times.
- `-var 'foo=bar'`: Sets a variable in the Terraform configuration. This flag can be set multiple times.
- `-var-file=foo`: Sets variables in the Terraform configuration from a file. If a `terraform.tfvars` file or any `.auto.tfvars` files are present, they will be automatically loaded.

You may have noticed that all the subcommands supported by `terraform apply` are also supported by `terraform destroy`. `terraform destroy` performs exactly the opposite operations of the `terraform apply` command. Likewise, we have seen that the `terraform apply` command helps us to provision or update infrastructure; similarly, the `terraform destroy` command helps us to delete all infrastructure that is present in the Terraform state file.

We will not discuss `terraform destroy` subcommands/arguments here but you can refer to the `terraform apply` subcommand use cases, and accordingly, if you are planning to use `terraform destroy`, just replace `terraform apply` with `terraform destroy`, followed by the appropriate subcommands.

You might be wondering how we can delete an infrastructure without using the `terraform destroy` command, so let's see how we can perform this activity. In our *terraform apply* section, we created a `terraform-lab-rg` resource group name. Let's try to delete that resource group without using the `terraform destroy` command directly, as follows:

```
PS C:\provider> terraform apply -var-file="testing.txt" -auto-approve
azurerm_resource_group.example: Creating...
azurerm_resource_group.example: Creation complete after 3s
[id=/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/terraform-lab-rg]
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Firstly, we will run `terraform plan` with a `destroy` flag and save the `terraform plan` output in `delete.txt` (that is, the `terraform plan -destroy -out delete.txt` command), and then follow this with `terraform apply "delete.txt"`, as illustrated in the following code snippet:

```
PS C:\provider> terraform plan -var-file="testing.txt" -destroy -out delete.txt
Refreshing Terraform state in-memory prior to plan...
```

The refreshed state will be used to calculate this plan, but will not be persisted to local or remote state storage.

```
azurerm_resource_group.example: Refreshing state... [id=/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/terraform-lab-rg]
```


An execution plan has been generated and is shown below.

Resource actions are indicated with the following symbols:

- destroy

Terraform will perform the following actions:

azurerm_resource_group.example will be destroyed

```
- resource "azurerm_resource_group" "example" {
  - id          = "/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/terraform-lab-rg" -> null
  - location   = "eastus" -> null
  - name       = "terraform-lab-rg" -> null
  - tags       = {} -> null
}
```


This plan was saved to: delete.txt

To perform exactly these actions, run the following command to apply:

```
terraform apply "delete.txt"
```

```
PS C:\provider> terraform apply "delete.txt"
```

```
azurerm_resource_group.example: Destroying... [id=/subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/resourceGroups/terraform-lab-rg]
```

```
azurerm_resource_group.example: Still destroying... [id=/subscriptions/97c3799f-2753-40b7-a4bd-...4d8fe/resourceGroups/terraform-lab-rg, 10s elapsed]
```

```
azurerm_resource_group.example: Destruction complete after 11s
```

Apply complete! Resources: 0 added, 0 changed, 1 destroyed.

From this, we got to know how easily we can delete resources using `terraform plan` and `terraform apply` sequentially without using `terraform destroy`. This is just another way of deleting resources, and you can use this in place of the `terraform destroy` command.

Important note

Run `terraform destroy` only when you want to perform a cleanup of resources because it will permanently delete the resource or services you have created during the `terraform apply` operation. It works in a totally opposite way to `terraform apply`.

By default, Terraform performs `terraform refresh` before executing `terraform init`, `terraform plan`, `terraform apply`, or `terraform destroy` operations.

Whenever you perform any Terraform workflow commands such as `terraform init`, `terraform plan`, `terraform apply`, or `terraform destroy`, the Terraform state file goes into a locking state to avoid anyone making changes to the state file simultaneously.

We discussed core Terraform workflows that consist of `terraform init`, `terraform plan`, `terraform apply`, and `terraform destroy`. You now have a fair understanding of how you can run a complete Terraform workflow from your local machine using the Terraform CLI. Now, we will explain Terraform workflows using an Azure DevOps Services.

Understanding Terraform workflows using Azure DevOps

It is very important to understand how we can use Terraform with any **continuous integration/continuous deployment (CI/CD)** tool because you know that these days, DevOps is in demand, and almost 90% of companies are using a DevOps approach. So, in order to understand Terraform with CI/CD tools, we will look at Azure DevOps tools.

Let's try to understand how we can keep our code in the Azure Repo and then use Azure Pipelines to perform the deployment of the infrastructure. In our example, we are going to use the Azure cloud platform, but you can use Azure DevOps with other major cloud providers such as GCP, AWS, and so on.

Have a look at the following diagram, which provides an overview of using Terraform with CI/CD workflows:

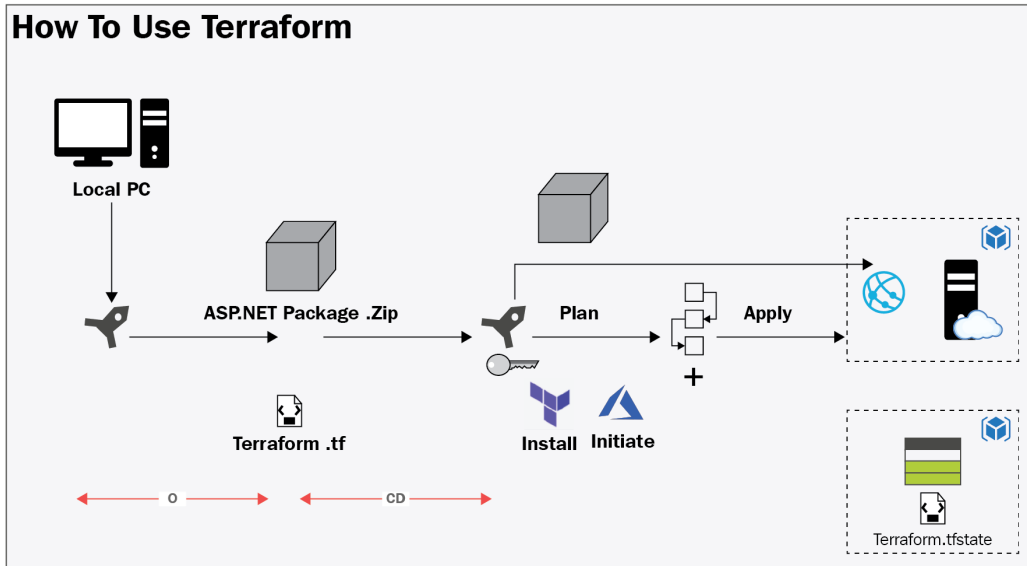


Figure 6.2 – Terraform with CI/CD workflows image

The following steps show you how to use Azure DevOps Service for infrastructure deployment in Azure using Terraform:

1. To perform this demonstration, we have created a project named `Terraform-lab-Project` in Azure DevOps, as shown in the following screenshot:

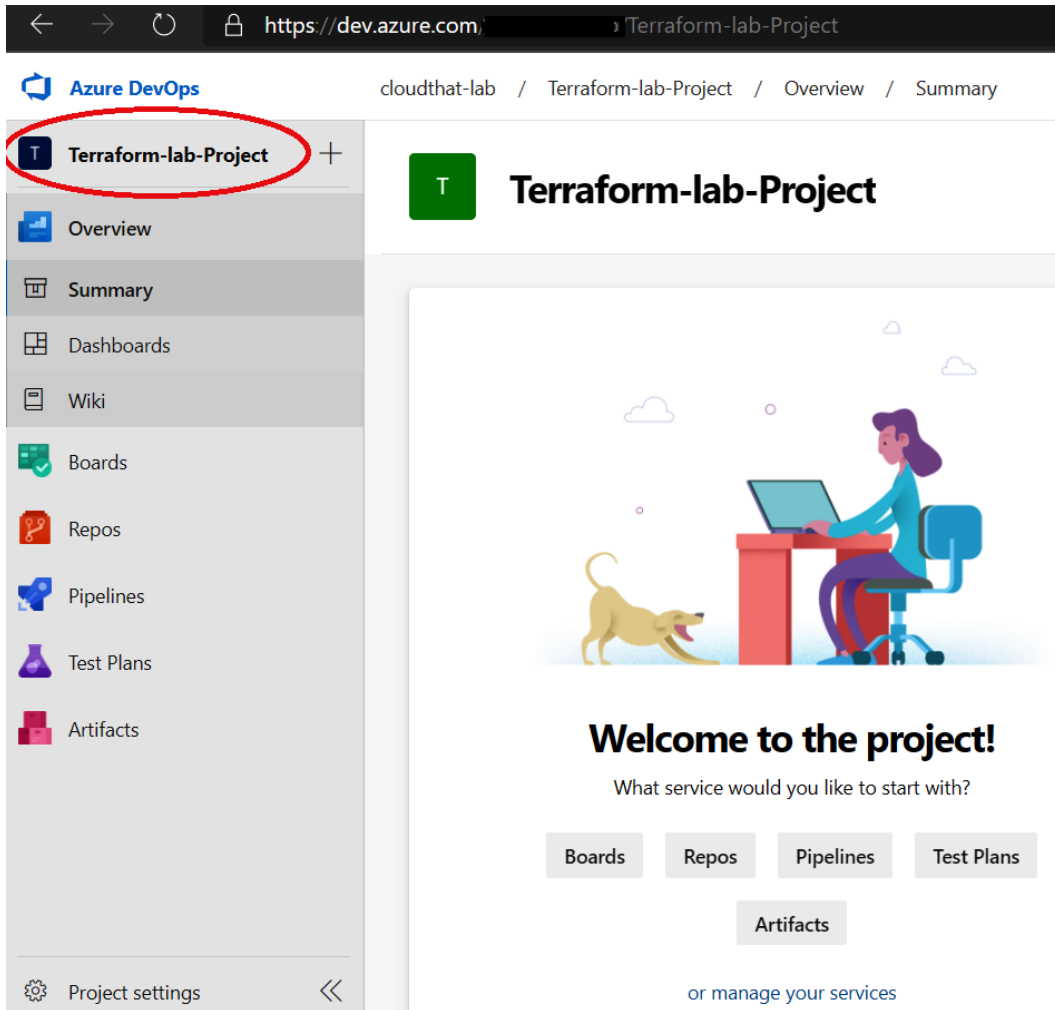


Figure 6.3 – Azure DevOps project

- The next step is to integrate our Azure DevOps project with Azure, and for that, you need to go to the **Project Settings | Service connections** option, where you will be able to see a list of possible connections that are supported by Azure DevOps. Here, select **Azure Resource Manager**, as illustrated in the following screenshot:

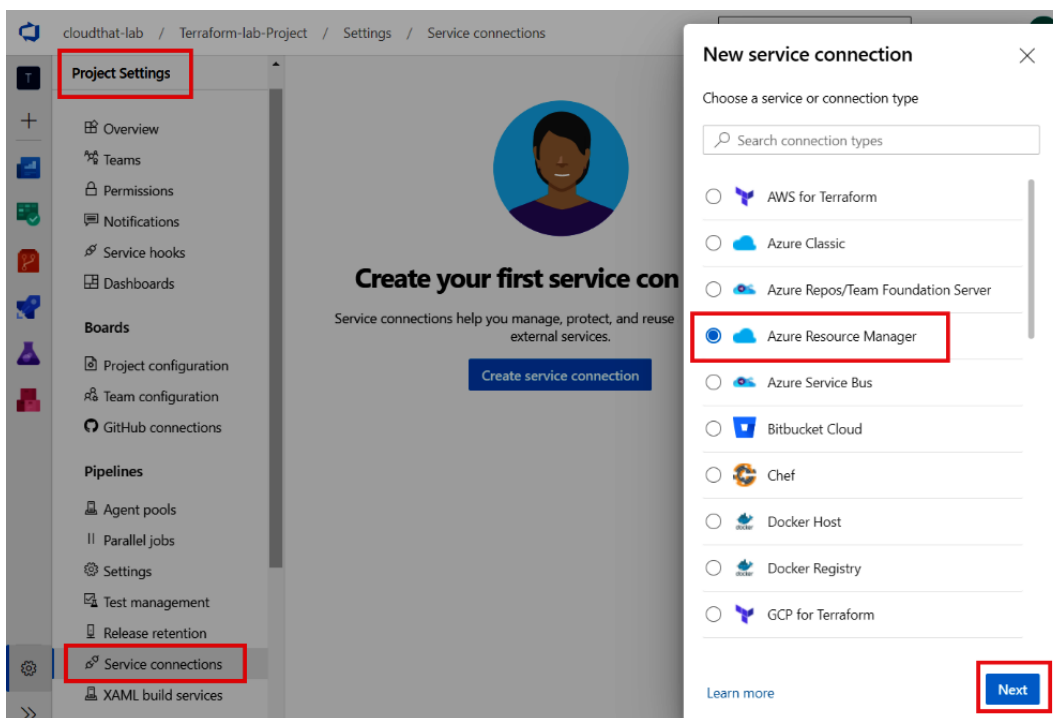


Figure 6.4 – Azure DevOps service connection

After selecting the **Azure Resource Manager** option, click on **Next**. We will then be able to see multiple options to build integration from Azure DevOps to the Azure portal, as shown in the following screenshot. We are going to use the first option (which is the recommended one) to automatically create a service principal in Azure and get proper access so that it can make changes to the Azure subscription:

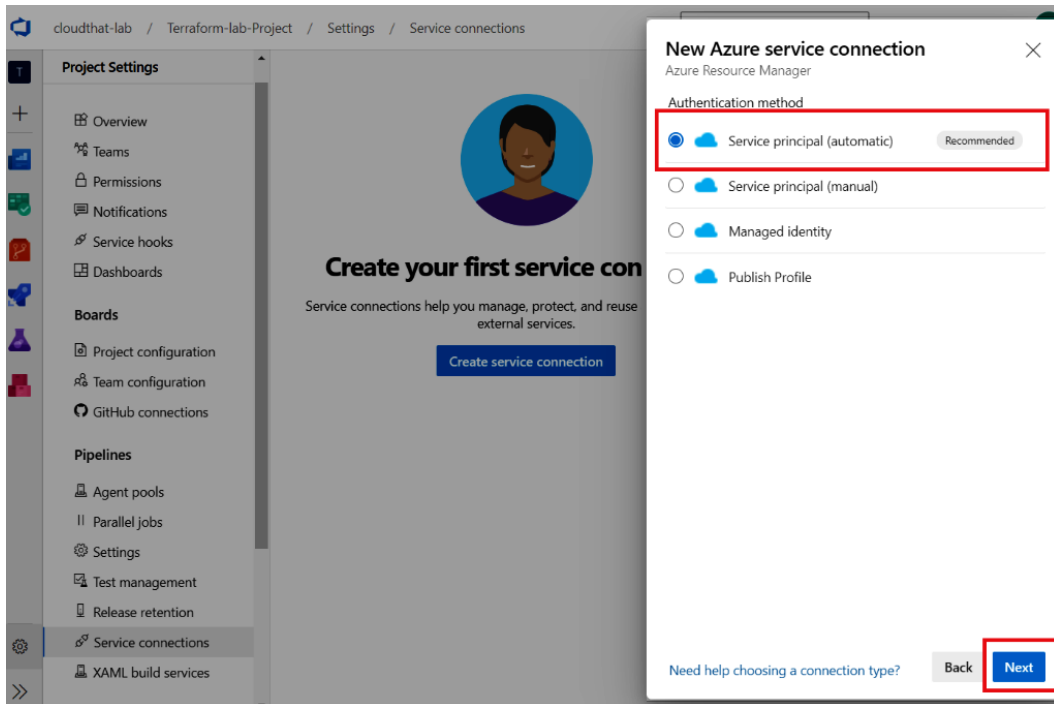


Figure 6.5 – Azure service principal

3. On the next screen, we select our Azure subscription and resource group and provide a name for the connection. We have already created a Terraform-lab-rg resource group in Azure, as well as a Terraform-lab-connection connection, as you can see in the following screenshot. You can keep the **Grant access permission to all pipelines** option checked so that you can use this connection with all pipelines in this project:

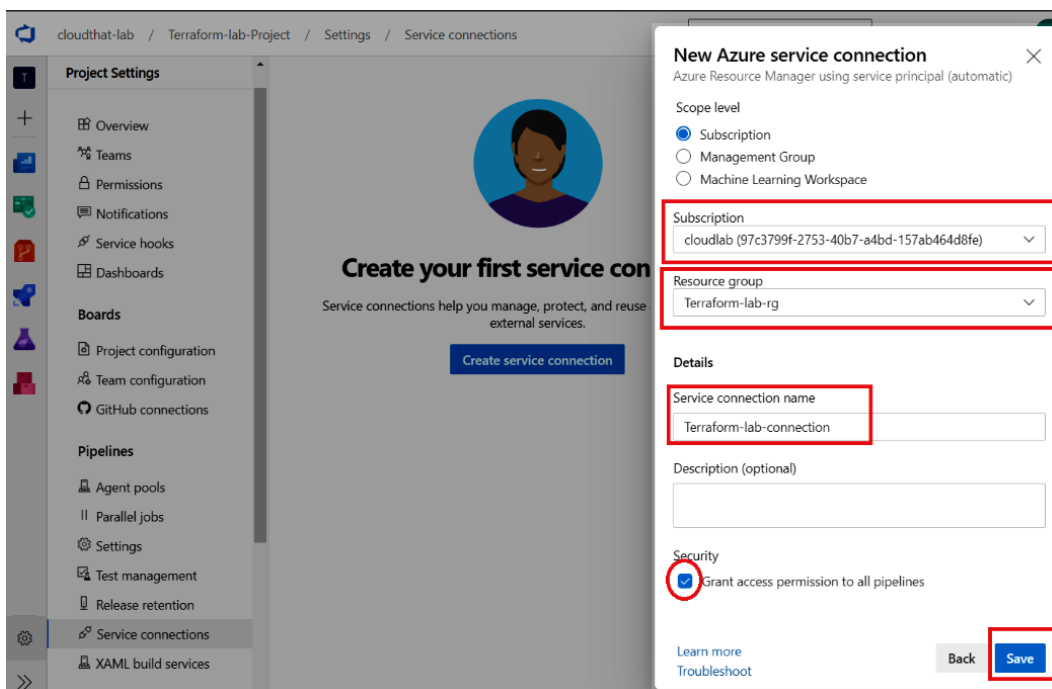


Figure 6.6 – Terraform-lab-connection

We are now all set in terms of integration of Azure DevOps with Azure, so let's make a clone of the Azure Repo and add all our Terraform configuration files to the Azure Repo. To perform this task, you should have Git Bash or any **integrated development environment (IDE)** such as **Visual Studio Code (VS Code)**. For the initial Git setup, you can follow <https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup>. We have taken an example of creating a web app in Azure using Terraform through Azure DevOps. We have kept all the required Terraform and build pipeline code in our GitHub code bundle for this chapter, which is available in the *Technical requirements* section. We have pushed all Terraform configuration files and CI pipeline code to the Azure Repo, as shown in the following screenshot:

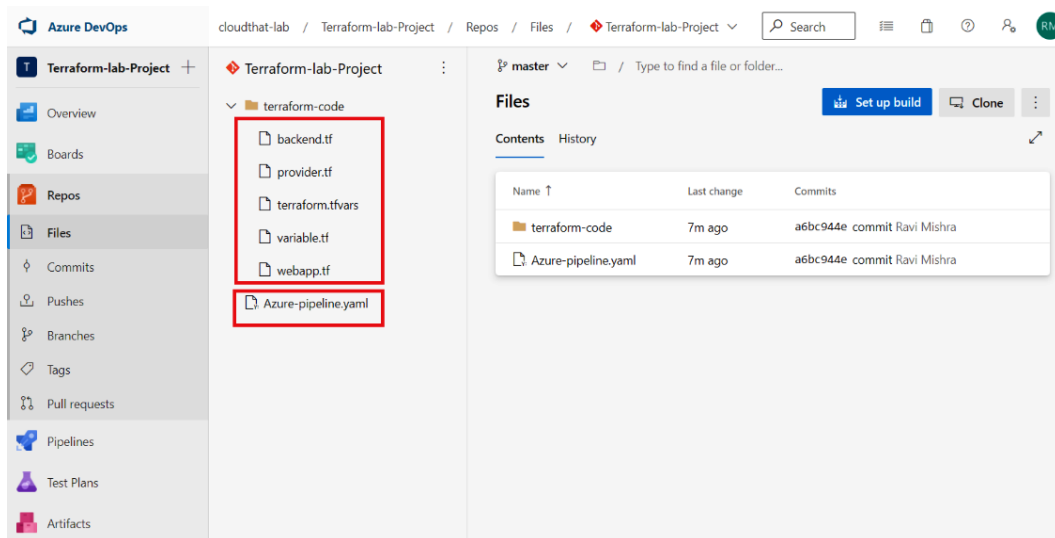


Figure 6.7 – Terraform and CI code

Now, we need to create a CI pipeline where we will be using the `Azure-pipeline.yaml` filename that's already been pushed to the Azure Repo. In the following screenshot, you can see that this CI pipeline has been successfully run:

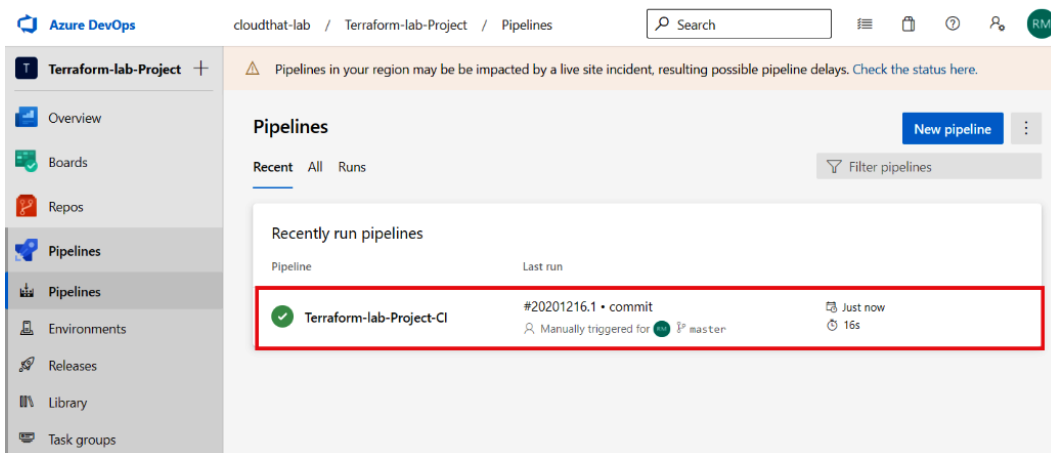


Figure 6.8 – Terraform-lab-Project-CI pipeline

What's next now that we are done with integration and our CI pipeline? Well, we now need to create a release pipeline to provision our Azure web app. So, to create the release pipeline, we need to define the Azure Artifact. Then we set the source type as **Build**, as shown in *figure 6.9*:

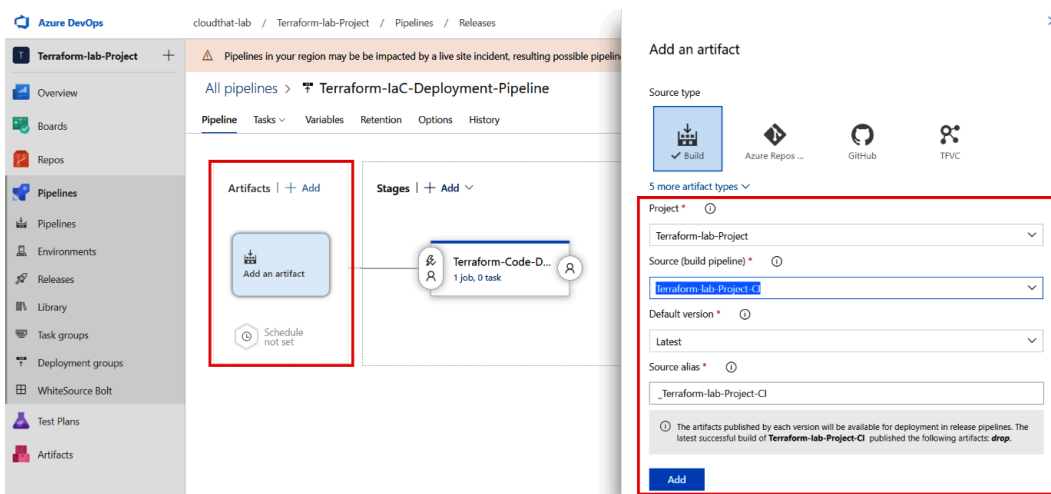


Figure 6.9 – Terraform-IaC-Deployment-Pipeline

In the release pipeline under the Terraform-IaC-Deployment-Pipeline stage name, we have defined all the tasks, as shown in the following screenshot:

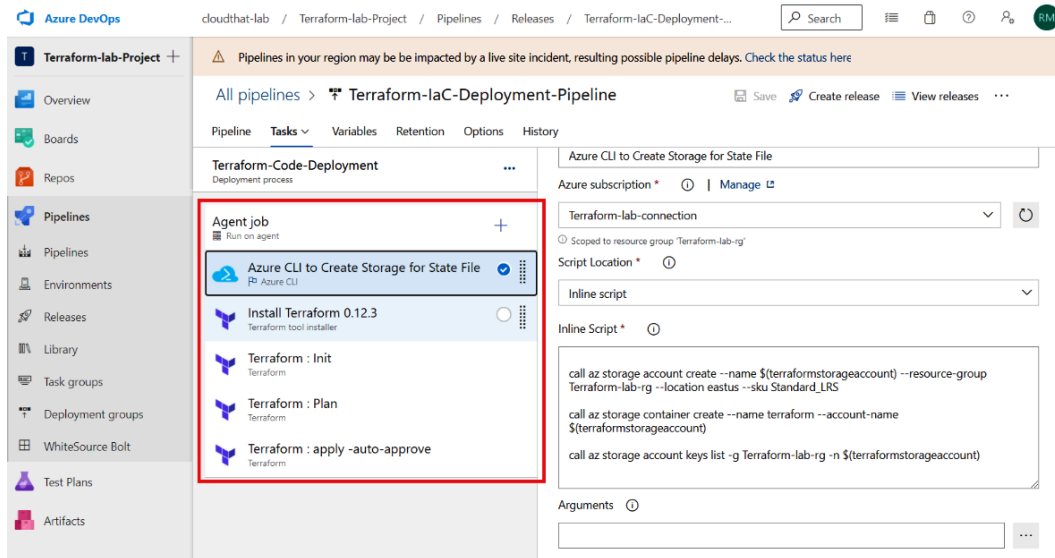


Figure 6.10 – Release pipeline tasks

Important note

You can provision an infrastructure using a CI/build pipeline—there is no need for a release pipeline. We have included a release pipeline here to help you understand the complete CI/CD workflow.

It's great to see that our pipeline ran successfully, as demonstrated in the following screenshot:

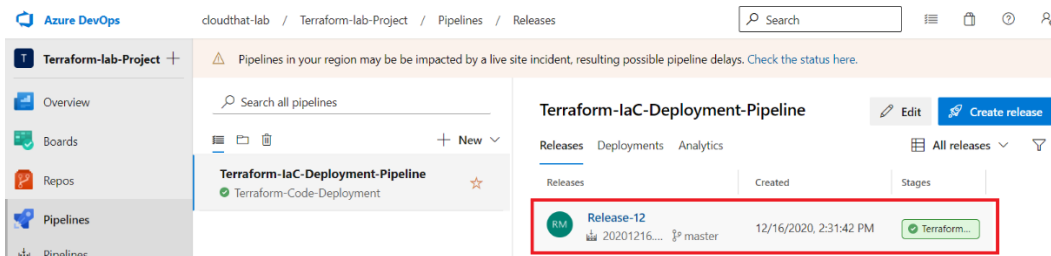


Figure 6.11 – Azure CD pipeline status

In the following screenshot, we can see that Terraform has created an Azure web app service and App Service plan for us in the Terraform-lab-rg resource group:

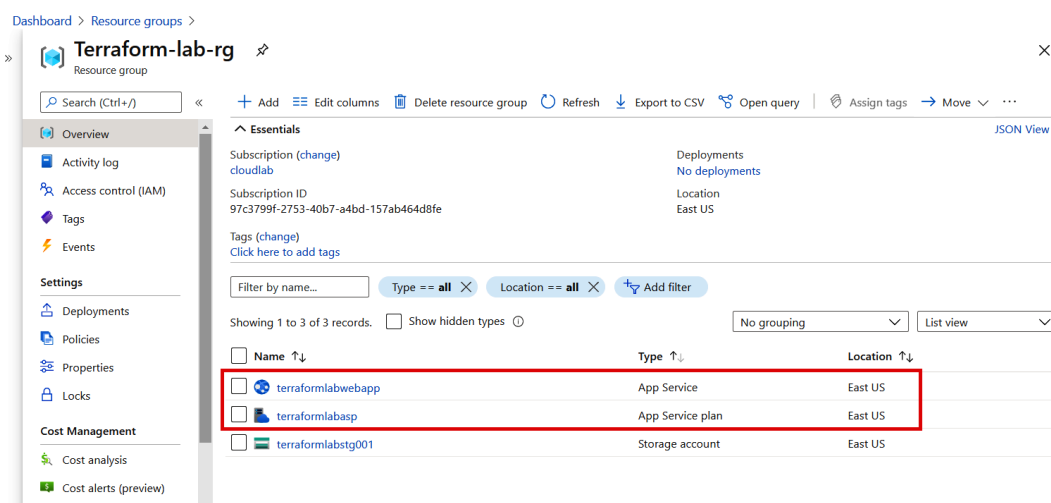


Figure 6.12 – Azure services

From this section, you should now understand how CI/CD tools (that is, Azure DevOps tools) can be used with Terraform for infrastructure management (updating or provisioning) in the Azure cloud. In the same way, you should be able to use this learning for the deployment of resources in AWS and GCP.

Summary

From this complete chapter, you will have developed an understanding of Terraform workflows, including `terraform init`, `terraform plan`, `terraform validate`, `terraform apply`, and `terraform destroy` workflows. You should now be capable of provisioning or updating infrastructure in a major cloud provider such as *AWS*, *Azure*, or *GCP*. We also discussed how we can use a CI/CD pipeline (that is, an Azure DevOps pipeline) with Terraform to provision services in Azure.

In our next chapter, we are going to discuss how we can write a Terraform module for *Azure*, *AWS*, and *Azure*, and how this can be published and consumed.

Questions

The answers to the following questions can be found in the *Assessments* section at the end of this book:

1. Which of the following actions are performed during a `terraform init` operation?
 - A. Initializes downloaded or installed providers
 - B. Downloads the declared provider
 - C. Provisions the declared resources
 - D. Initializes the backend config
2. What can the `terraform plan` command do?
 - A. Provision declared resources
 - B. Perform initialization of the backend
 - C. Create an execution plan and determine which changes need to be made to achieve the desired state in the configuration file
 - D. Perform linting on the Terraform configuration file
3. You have defined the following configuration code block:

```
resource "azurerm_resource_group" "example" {  
  name      = var.rgname  
  location = "eastus"  
}
```

In which possible ways can Terraform take the `rgname` variable value?

- A. By creating a `terraform.tfvars` file and placing it into `rgname="Terraform-lab-rg"`
 - B. By running `terraform apply -var 'rgname=Terraform-lab-rg'`
 - C. By running `terraform plan`
 - D. By running `terraform apply -var-file="example.txt"`
4. You are getting an error like this while executing the `terraform apply` command:

```
resource "azurerm_resource_group" "example" {
  name      = "Terraform-lab-RG"
  location  = "eastus"
  terraform_location = "eastus"
}
Error: Unsupported argument
   on resourcegroup.tf line 4, in resource "azurerm_
resource_group" "example":
    4:   terraform_location = "eastus"
An argument named "terraform_location" is not expected
here.
```

Which command could you have run to check such an error in advance?

- A. `terraform init`
 - B. `terraform validate`
 - C. `terraform plan`
 - D. `terraform apply`
5. Your team has created an AWS infrastructure using Terraform, and now they want to get rid of that infrastructure. Which command could you run to do this?
- A. `terraform validate`
 - B. `terraform plan`
 - C. `terraform destroy`
 - D. `terraform init`

Further reading

You can check out the following links for more information about the topics that were covered in this chapter:

- terraform init: <https://www.terraform.io/docs/commands/init.html>
- terraform plan: <https://www.terraform.io/docs/commands/plan.html>
- terraform apply: <https://www.terraform.io/docs/commands/apply.html>
- terraform destroy: <https://www.terraform.io/docs/commands/destroy.html>
- Terraform DevOps lab: <https://azuredevopslabs.com/labs/vstsextend/terraform/>

7

Terraform Modules

In our previous chapter, we discussed the core workflow of the Terraform tool, which consists of creating a Terraform configuration file (`write`), previewing the changes (`terraform plan`), and then finally committing those changes to the target environment (`terraform apply`). Once we are done with the creation of the resources, we might be required to get rid of those infrastructures (`terraform destroy`). In a nutshell, we discussed complete Terraform core workflows, which mainly consist of `terraform init`, `terraform plan`, `terraform apply`, `terraform destroy`, and the respective subcommands and their outputs.

In this chapter, we will discuss Terraform modules for **Amazon Web Services (AWS)**, **Azure**, and **Google Cloud Platform (GCP)** cloud providers. From this chapter, you will learn how you can write modules and how you can publish and consume them. Learning about Terraform modules is important as this will help you to deploy large enterprise-scalable and repeatable infrastructure and will also reduce the total number of hours required for provisioning infrastructure.

The following topics will be covered in this chapter:

- Understanding Terraform modules
- Writing Terraform modules for Azure
- Writing Terraform modules for AWS
- Writing Terraform modules for GCP
- Publishing Terraform modules

Technical requirements

To follow along with this chapter, you will need to have an understanding of the Terraform **command-line interface (CLI)**. You should know how to install Terraform on various machines and need to have a basic understanding of writing a Terraform configuration file. You can find all the code used in this chapter at the following GitHub link:

<https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter7>.

Check out the following link to see the Code in Action video:

<https://bit.ly/36pJouo>

Understanding Terraform modules

So far, we have written Terraform configuration code, but have you ever thought about how effectively we can make it reusable? Just consider that you have many product teams with whom you are working, and they wish to use Terraform configuration code for their infrastructure deployment. Now, you may be wondering if it's possible to write code once and share it with these teams so that they don't need to write the same code again and again. This is where **Terraform modules** come in. Let's try to get an idea of what these are.

A Terraform module is basically a simple configuration file ending with `.tf` or `tf.json`, mainly consisting of resources, inputs, and outputs. There is a main root module that has the main configuration file you would be working with, which can consume multiple other modules. We can define the hierarchical structure of modules like this: the root module can ingest a child module, and that child module can invoke multiple other child modules. We can reference or read submodules in this way:

```
module.<rootmodulename>.module.<childmodulename>
```

However, it is recommended to keep a module tree flat and have only one level of child module. Let's try to understand about **module composition**, with an example of a **virtual private cloud (VPC)** and a **subnet** of AWS. Here is the resource code block:

```
resource "aws_vpc" "vpc" {
  cidr_block = var.cidr_block
}
resource "aws_subnet" "subnet" {
  vpc_id           = aws_vpc.vpc.id
  availability_zone = "us-east-1a"
  cidr_block       = cidrsubnet(aws_vpc.vpc.cidr_block, 4, 1)
}
variable "cidr_block" {
```

```

default      = "10.0.0.0/16"
type         = string
description  = "provide cidr block range"
}

```

When we try to write module code blocks, the configuration appears more hierarchical than flat. Each of the modules would have its own set of resources and child modules that may go deeper and create a complex tree of resource configurations.

Here is a code example, to explain how a module can be referenced in another module:

```

module "aws_network" {
  source      = "./modules/network"
  cidr_block = "10.0.0.0/8"
}

module "aws_instance" {
  source      = "./modules/instance"
  vpc_id      = module.aws_network.vpc_id
  subnet_ids = module.aws_network.subnet_ids
}

```

In our preceding example, you can see how we referenced one module in another module. It is always suggested to have a one-level module reference rather than referencing complex modules and submodules.

In simple words, we can define a module as a container with multiple resources that can be consumed together. We will discuss in more detail writing child modules and later consuming them as and when needed by creating a Terraform stack in *Chapter 9, Understanding Terraform Stacks*.

The syntax for defining Terraform modules, whereby we need to start with the module and provide its local name, is shown here:

```

module "name" {}

```

Terraform modules support some key arguments such as `source`, `version`, and `input variable`, and loops such as `for_each` and `count`. We can define a module code block in this way:

```

module "terraform-module" {
  source = "terraform-aws-modules/vpc/aws"
  version = "2.55.0"
}

```


In the previously defined code block, we have defined a module with a local name of `terraform-module`. We can give any name to a module—it's totally up to us, because it's just a local name. You may be wondering what these `source` and `version` arguments are doing inside the module code block. An explanation for this follows.

source

This argument is mandatory, referring to either the local path where the module configuration file is located or the remote reference **Uniform Resource Locator (URL)** from where it should be downloaded. The same source path can be defined in multiple modules or in the same file and can be called multiple times, making modules more efficient for reuse. The next code example will give you an idea of how a module can be defined.

For local referencing, you can define a module in the following way:

```
module "terraform-module" {  
  source = "../terraform-module"  
}
```

In the previously defined code block, we are referencing the local path where the module configuration file is located. We have two options for referencing a local path: `./` and `../`. We have used `../`, which takes the `terraform-module` directory (that is, the parent directory). In this case, when we run `terraform init` and `terraform plan`, the module file doesn't get downloaded as it already exists on the local disk. It is also recommended to use a local file path for closely related modules that are used for the factoring of repeated code. This referencing won't work if you are working in a team and they want to consume a different version of a module. In that case, a remote file path would help because the published version of the code would be there in the remote location, and anyone could easily reference it by providing the respective file URL.

The next aspect you need to know about is the different remote ways of referencing supported by Terraform that can be defined in the source block of the Terraform module. For in-depth learning about remote referencing, you can refer to <https://www.terraform.io/docs/language/modules/sources.html>. Here are a few ways in which you can reference remotely:

- Terraform Registry
- GitHub
- Bitbucket
- Generic Git; Mercurial repositories
- **HyperText Transfer Protocol (HTTP)** URLs

- **Simple Storage Service (S3)** buckets
- **Google Cloud Storage (GCS)** buckets

Terraform Registry

Terraform Registry has a list of published modules written by community members. This is a public registry, and it's very easy for others to start consuming it directly in their code. Terraform Registry supports versioning of published modules. It can be referenced using the specified syntax `<NAMESPACE>/<NAME>/<PROVIDER>`—for example, `hashicorp/consul/aws`, as illustrated in the following code snippet:

```
module "Terraform-consul" {  
  source = "hashicorp/consul/aws"  
  version = "0.8.0"  
}
```

For a detailed understanding about usage of the `consul` module for AWS, you can read more at <https://registry.terraform.io/modules/hashicorp/consul/aws/latest>.

If you want to use modules that are hosted in a private registry similar to Terraform Cloud, then you can reference them in code by providing a source path with the syntax `<HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>`. For a detailed understanding of private registry referencing, you can read more at <https://www.terraform.io/docs/cloud/registry/using.html>.

Have a look at the following code example:

```
module "aws-vpc" {  
  source = "app.terraform.io/aws_vpc/vpc/aws"  
  version = "1.0.0"  
}
```

In this example, we have shown the **Software-as-a-Service (SaaS)** version of Terraform Cloud, which has a private registry hostname of `app.terraform.io`. If you are going to use any other private registry, you can replace this hostname with your private registry hostname.

Important note

You might be required to configure credentials to access modules, depending upon the private registry you are referencing for the module source path.

GitHub

Terraform will be able to recognize `github.com` URLs and understand them as Git repository sources. The following code snippet shows cloning over **HTTP Secure (HTTPS)**:

```
module "terraform-module" {  
  source = "github.com/hashicorp/terraform-module"  
}
```

To clone over **Secure Shell (SSH)**, you can use the following code:

```
module "terraform-module" {  
  source = "git@github.com:hashicorp/terraform-module.git"  
}
```

GitHub supports a `ref` argument that helps select a specific version of a module. You would be required to pass credentials to get authenticated to a private repository.

Bitbucket

Similar to GitHub, Terraform manages to recognize `bitbucket.org` and is able to understand it as Bitbucket repository sources. The following code snippet illustrates this:

```
module "terraform-module" {  
  source = "bitbucket.org/hashicorp/terraform-module"  
}
```

If you are using a public repository, then the previously defined code form will help Terraform use the Bitbucket **application programming interface (API)** to learn whether the defined source is using a Git source or a Mercurial source. When using a private repository, you would be required to have proper authentication.

Generic Git repository

You can define any valid Git repository by prefixing the address with `git::`. Both SSH and HTTPS can be defined in the following ways:

```
module "terraform-module" {  
  source = "git::https://example.com/terraform-module.git"  
}  
  
module "terraform-module" {
```

```
source = "git::ssh://username@example.com/terraform-module.git"
}
```

Terraform downloads modules from the Git repository by executing `git clone`. If you are trying to access a private repository, you would then be required to provide an SSH key or credentials so that it can be accessed.

If you are looking for a specific version of a module from the Git repository, then you can pass that version value into the `ref` argument, as follows:

```
module "terraform-module" {
  source = "git::https://example.com/terraform-module.git?ref=v1.2.0"
}
```

Generic Mercurial repository

Terraform will be able to understand any generic Mercurial URL through an `hg::` prefix, as illustrated in the following code snippet:

```
module "terraform-module" {
  source = "hg::http://example.com/terraform-module.hg"
}
```

Terraform is able to install modules from Mercurial repositories by running `hg clone`, but if you are accessing a non-public repository then an SSH key or credentials need to be provided. Terraform helps you to download a specific version of the module from Mercurial repositories by supporting the `ref` argument, as illustrated in the following code snippet:

```
module "terraform-module" {
  source = "hg::http://example.com/terraform-module.hg?ref=v1.2.0"
}
```

HTTP URLs

Terraform has the capability to perform a GET operation if you are providing an HTTP or HTTPS URL. It manages to install modules over HTTP/HTTPS easily, provided there is no authentication required. If authentication is required, then you would be required to provide credential details in the `.netrc` file of the home directory.

Terraform manages to read the following archival file extensions:

- `zip`
- `tar.bz2` and `tbz2`
- `tar.gz` and `tgz`
- `tar.xz` and `txz`

Here is an example that shows how Terraform can access archival files using HTTPS:

```
module "terraform-module" {  
  source = "https://example.com/terraform-module.zip"  
}
```

Let's suppose you are not sure about the archived file extension and you want Terraform to be able to read it. In that case, you can define an `archive` argument to force Terraform to interpret it, as follows:

```
module "terraform-module" {  
  source = "https://example.com/terraform-module?archive=zip"  
}
```

S3 bucket

You can publish and store a module in an AWS S3 bucket. If you want to reference a S3 bucket in a Terraform module, just place a `S3::` prefix followed by the S3 bucket object URL, as follows:

```
module "terraform-module" {  
  source = "s3::https://s3-eu-west-1.amazonaws.com/terraform-modules/tfmodules.zip"  
}
```

Terraform can access S3 buckets using the S3 API, provided authentication to AWS has been handled. The Terraform module installer will look for AWS credentials in the following locations:

- The defined value of `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` in the environment variables
- Credentials stored in the `.aws/credentials` file in your home directory

GCS bucket

As with a S3 bucket in AWS, you can archive your module configuration in a GCS bucket. If you wish to reference it in a module code block, then just place a `gcs::` prefix followed by the GCS bucket object URL, as illustrated in the following code snippet:

```
module "terraform-module" {  
  source = "gcs::https://www.googleapis.com/storage/v1/modules/  
terraform-module.zip"  
}
```

The module installer uses the Google Cloud **software development kit (SDK)** to get authenticated with GCS. You can provide the file path of the Google service account key file so that it can get authenticated.

We have discussed different available sources for Terraform modules. For more information, you can go to <https://www.terraform.io/docs/modules/sources.html>.

version

This argument is optional—if you are referencing the local path in the source, then you won't be able to define a version. Providing a version value is very important if you want to download a specific version of a module when you run `terraform init`, and it will also protect you from any unwanted outages due to a change in the configuration code. You need to define version arguments in a module code block in this way:

```
module "terraform-module" {  
  source = "terraform-aws-modules/vpc/aws"  
  version = "2.5.0"  
}
```

In the preceding code block, we have defined the version value as `"2.5.0"` so that whenever we are running `terraform init`, it will download that specific version of the module. You are probably wondering what will happen if you don't provide this version in a module code block. If you don't define the version in a module code block, then Terraform will always install the latest module version. So, in order to protect infrastructure from any breakage, it's recommended to provide version constraints in a module. You can provide a version value with the simple `=` operator, but there are many other version constraint operators supported by Terraform, such as the following:

- `>= 2.5.0`: version 2.5.0 or newer
- `<= 2.5.0`: version 2.5.0 or older

- `~>2.5.0`: any version in the 2.5.x family; version 2.6.x would not work
- `>= 2.0.0, <= 2.5.0`: any version between 2.0.0 and 2.5.0 inclusive

If you are installing a module from a module registry such as Terraform Registry or Terraform Cloud's private module registry, then you can provide version constraints. Other registries such as GitHub, Bitbucket, and so on provide their own way of defining version details in the source URL itself, which we already discussed in our *source* section.

The Terraform modules support some meta-arguments, along with `source` and `version`. These are outlined as follows:

- `count`—You can define this argument within a module code block if you willing to create multiple instances from a single module block. For a greater understanding of the `count` expression, you can refer to the *Understanding Terraform loops* section of *Chapter 4, Deep Dive into Terraform*, where we already discussed how you can use `count` for a resource code block. The same approach is followed in a module block as well, so we won't discuss it again here. For further reading, you can go to the `count` expression page of the Terraform documentation, at <https://www.terraform.io/docs/configuration/meta-arguments/count.html>.
- `for_each`—As with the `count` expression, the `for_each` expression is also used for creating multiple instances of a module from a single module block. We already discussed this in *Chapter 4, Deep Dive into Terraform*, where we discussed `for_each` loops respective to a resource code block, and you can apply the same approach here as well; therefore, we won't discuss this again here. If you want to read more about the `for_each` expression, you can visit https://www.terraform.io/docs/configuration/meta-arguments/for_each.html.
- `providers`—This is an optional meta-argument that you can define within a module code block. You may be wondering why we need to define a `providers` argument inside a module code block. Basically, you need to define this when you want to have multiple configurations in a module of the same provider. You can then call them by placing `alias` for the provider in the following way:

```
terraform {  
  required_version = ">= 1.0"  
  required_providers {  
    aws = {  
      source = "hashicorp/aws"  
      version = "~> 3.0"  
    }  
  }  
}
```

```

    }
  }
  provider "aws" {
    region = "us-east-1"
  }
  # An alternate configuration is also defined for a
  # different
  # region, using the alias "terraform_aws_west".
  provider "aws" {
    alias   = "terraform_aws_west"
    region = "us-west-1"
  }
  module "terraform-module" {
    source      = "./terraform-module"
    providers = {
      aws = aws.terraform_aws_west
    }
  }
}

```

In this example, we defined the same provider multiple times, one with an alias and another without an alias. If we hadn't defined a `providers` arguments in the child module code block, it would have taken the default provider that we defined without an alias. So, in order to use a specific provider within the module, we defined `aws=aws.terraform_aws_west`. From this expression, you can easily understand that the `providers` argument inside the child module is a map that will be supporting only the key and the value, without placing them in double quotes.

- `depends_on`—This meta-argument is an optional one, whereby if we need to define an explicit dependency then we can define a `depends_on` argument within a module code block. This should be used only as a very last resort because Terraform is smart enough to follow an implicit dependency of resources, but in some cases we need to forcefully tell Terraform that there is a dependency so that Terraform can follow that sequence for the creation or deletion of resources. Let's try to understand how we can define the `depends_on` argument within a module, as follows:

```

resource "aws_iam_role" "terraform-role" {
  name = "terraform-vpc-role"
}

```



```
# We need to create AWS IAM role specific to VPC then we
# are supposed to create VPC
module "terraform-module" {
  source = "terraform-aws-modules/vpc/aws"
  version = "~>2.5.0"
  depends_on = [
    aws_iam_role.terraform-role,
  ]
}
```

In this example, we mentioned explicit dependency in the module by defining `depends_on`, which means that Terraform will deploy the `terraform-module` module only when we already have the AWS **Identity and Access Management (IAM)** role in place. Similar to this example, we can even define explicit dependency of one module to other child modules.

There are many more aspects related to Terraform modules and we will try to discuss them in our upcoming sections, providing examples so that you will gain a better understanding. For now, let's try to understand how we can taint a resource within a module. To taint a specific resource within a module, we can use Terraform's `taint` command, as illustrated in the following code snippet:

```
$ terraform taint module.terraform_module.aws_instance.
terraform_instance
```

This will tell Terraform to destroy and recreate a specific resource of modules that's been tainted during the next `apply` operation. Remember—you won't be able to taint a complete module; only specific resources within a module can be tainted. For further understanding of the Terraform `taint` command, you can read *Chapter 5, Terraform CLI*. You can also read more about the `taint` command at <https://www.terraform.io/docs/commands/taint.html>.

From this entire section, you will have gained a fair understanding of Terraform modules, so let's move ahead and try to write some modules for major cloud providers such as Azure, AWS, and GCP.

Writing Terraform modules for Azure

When learning about Terraform modules, we need to understand how we can draft modules for an **Azure Resource Manager** (azurerm) provider. This will help you out when starting to create azurerm modules and provisioning resources in Microsoft Azure using these modules. For a further understanding about Terraform modules for Azure, let's try to create a **virtual machine (VM)** with the keyvault module. We have created all the module Terraform files and placed them in our chapter GitHub repository at <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter7/azurerm/azurerm-virtual-machine-module>.

The following files are present in our GitHub repository under the azurerm directory, which you can clone to your local machine and then start consuming it:

```
inmishrar@terraform-lab-vm:~/HashiCorp-Infrastructure-Automation-Certification-Guide/chapter7/azurerm# tree
```

```
.
├── azurerm-virtual-machine-module
│   ├── VERSION
│   ├── main.tf
│   ├── outputs.tf
│   └── variables.tf
└── azurerm-virtual-machine-module-use-case
    ├── main.tf
    ├── outputs.tf
    ├── providers.tf
    ├── terraform.tfvars
    └── variables.tf
```

```
2 directories, 9 files
```

In our `main.tf` file, we have defined the following code block (as mentioned, the entire code can be copied from GitHub):

```
resource "azurerm_resource_group" "rgname" {
  name      = var.rgname
  location = var.location
  tags     = var.custom_tags
}
...
```

In this `main.tf` file, we have defined one code block named `locals`, as illustrated in the following code snippet:

```
resource "random_string" "password" {
  length      = 16
  special     = true
  min_upper   = 2
  min_lower   = 2
  min_numeric = 2
  min_special = 2
}
locals {
  vmpassword = random_string.password.result
}
```

`locals` is just a local variable, having a map data type that contains a key and a value. It is used within the code by referencing `local.key`. In our example, we are storing a value of `"random_string" "password"` in `vmpassword`. So, wherever we want to reference or use this local variable in the Terraform configuration code, we can define `local.vmpassword` and it will automatically take the content of it. Here is a code example, just to give you an idea of how you can use `locals`:

```
resource "azurerm_key_vault_secret" "key_vault_secret" {
  name          = var.keyvault_secret_name
  value         = local.vmpassword
  key_vault_id = azurerm_key_vault.key_vault.id
  tags         = var.custom_tags
}
```

There is a `variables.tf` file that contains the following code that you can copy from the GitHub repository:

```
variable "rgname" {
  type      = string
  description = "name of resource group"
}
variable "location" {
  type      = string
  description = "location name"
}
...
```

One more file with the name `outputs.tf` contains the following code:

```
output "nic" {
  value = azurerm_network_interface.nic.private_ip_address
}
output "vm_id" {
  value = azurerm_windows_virtual_machine.virtual_machine.id
}
output "vm_name" {
  value = azurerm_windows_virtual_machine.virtual_machine.name
}
```

This whole code that we defined inside the `azurerm-virtual-machine-module` directory explains to you how you can create a module and get this published. In our case, we have kept the code in our GitHub repository so that you can create any modules accordingly and get them published.

Now, a very important question is: *How can we consume our created module?* So, for this, we have written code and placed this inside our `Chapter7/azurerm/azurerm-virtual-machine-module-use-case/` GitHub directory.

In the `main.tf` file, the following code is present:

```
module "terraform-vm" {  
  source = "github.com/PacktPublishing/HashiCorp-  
Infrastructure-Automation-Certification-Guide.git//chapter7/  
azurerm/azurerm-virtual-machine-module?ref=v0.0.1"  
  rgname      = var.rgname  
  location    = var.location  
  custom_tags = var.custom_tags  
  vm_size     = var.vm_size  
  vm_name     = var.vm_name  
  admin_username = var.admin_username  
  vm_publisher = var.vm_publisher  
  vm_offer    = var.vm_offer  
  vm_sku      = var.vm_sku  
  vm_version  = var.vm_version  
  sku_name    = var.sku_name  
  vnet_name   = var.vnet_name  
  address_space = var.address_space  
  subnet_name = var.subnet_name  
  nic_name    = var.nic_name  
  keyvault_name = var.keyvault_name  
  keyvault_secret_name = var.keyvault_secret_name  
}
```

If you notice, we have defined `source` in the module code block and have put `ref` in the source URL itself rather than defining any version argument separately because GitHub doesn't support version argument in the module code block. If we want to define versions, then we have to get them defined in the source URL itself.

There is one catch here—in order to define `ref` in the source URL of a module, you would be required to create a `releases/tags` in the GitHub repository, which you can see in the following screenshot. We have created release `v0.0.1` manually in our GitHub repository and associated it with our master branch. If you wish, you can define a script that can automatically create a specific release version, and you can define this accordingly in the source URL of the module code. You can refer to a given URL (that is, <https://stackoverflow.com/questions/18216991/create-a-tag-in-a-GitHub-repository>) to get an idea of creating a Releases/Tags on GitHub:

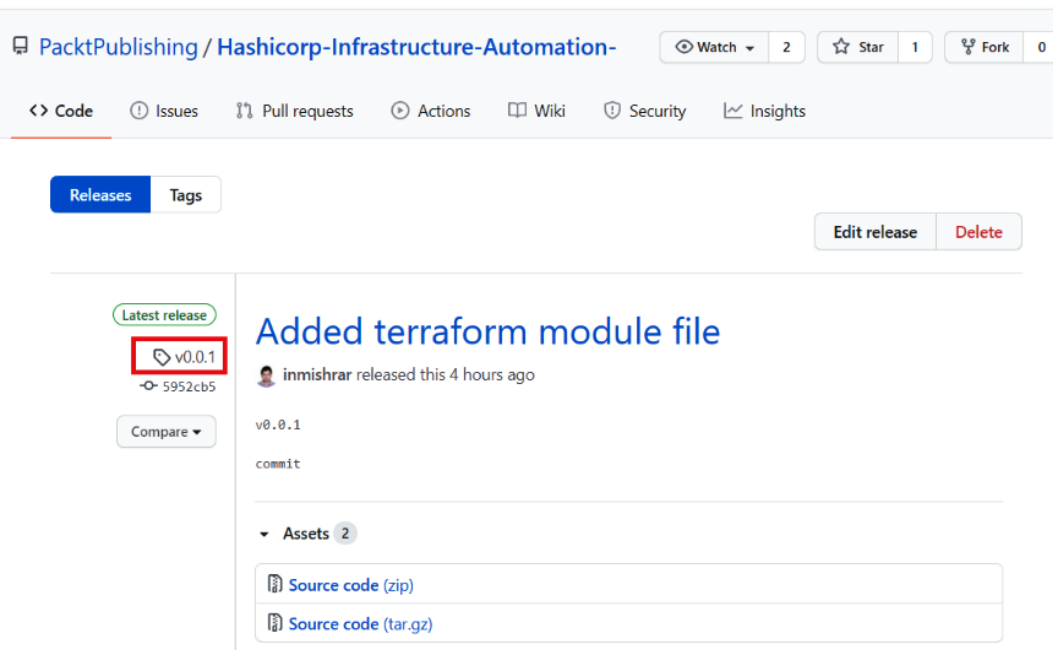


Figure 7.1 – GitHub release v0.0.1

So, depending upon your repository, you can define your source URL accordingly in the module code block.

The next aspect is the values we would be required to provide in order to consume this module. We highlighted all the arguments (such as `rgname`) on the left side of the module code block. `keyvault_name` is actually the variable name that we defined while creating the module code block, and this contains all our resources and respective variables. While consuming that module, all those variables will be working as an input argument for us. After that, you can provide respective argument values either through a variable or by hardcoding. We would recommend providing values through input variables. In our case, we are going to provide all modules' variable values from the `terraform.tfvars` file, as illustrated in the following code snippet:

```
rgname    = "Terraform-rg"
location  = "eastus"
custom_tags = {
  Environment = "prod"
  Owner       = "Azure-Terraform"
```

```
}
vm_size           = "Standard_F2"
vm_name           = "Terraform-vm"
admin_username    = "azureterraform"
vm_publisher      = "MicrosoftWindowsServer"
vm_offer          = "WindowsServer"
vm_sku            = "2016-Datacenter"
vm_version        = "latest"
sku_name          = "premium"
vnet_name         = "Terraform-vnet"
address_space     = ["10.1.0.0/16"]
subnet_name       = "Terraform-subnet"
nic_name          = "Terraform-nic"
keyvault_name     = "terraform-vm-keyvault"
keyvault_secret_name = "Terraform-vm-password"
```

We have kept the same variable that we defined while creating modules so that it is easy for you to understand. In the `variables.tf` file, the following code is present. You can copy the full code from our GitHub repository:

```
variable "rgname" {
  type      = string
  description = "name of resource group"
}
variable "location" {
  type      = string
  description = "location name"
}
variable "vnet_name" {
  type      = string
  description = "vnet name"
}
...
```

To store the Terraform state file into Azure Blob storage, we have defined a Terraform backend code block inside the `providers.tf` file. We also mentioned which version of the `azurerm` provider should be used. The code is illustrated in the following snippet:

```
terraform {  
  required_version = ">= 1.0"  
  required_providers {  
    azurerm = {  
      source = "hashicorp/azurerm"  
      version = "2.55.0"  
    }  
  }  
  backend "azurerm" {  
    storage_account_name = "terraformstg2345"  
    container_name       = "terraform"  
    key                  = "terraform.tfstate"  
    access_key           = "KRqtJIA0Gp4oKBsElDU7RGN..."  
  }  
}  
provider "azurerm" {  
  features {}  
}
```

Important note

While configuring the remote backend with Blob storage, you should pass the `access_key` value through environment variables. It is not recommended to hardcode the `access_key` value in your configuration code as this can cause a security risk.

Finally, we want to see whether our VM got deployed or not. So, for that, we have created a file named `outputs.tf`, which contains the following code:

```
output "vm_private_ip" {  
  value = module.terraform-vm.nic  
}  
output "vm_name" {  
  value = module.terraform-vm.vm_name  
}
```



```
output "vm_id" {  
  value = module.terraform-vm.vm_id  
}
```

In this `outputs.tf` file, you can see how we defined output values referencing modules. The syntax for defining the output of a module is `module.<MODULE NAME>.<OUTPUT NAME>`. In our example, while creating the module code, in the resource code block we mentioned the `outputs.tf` file where all the output names are, so when we are calling them as output in a module, we then need to provide that same name.

Let's try to run the code that is defined in the `azurerm-virtual-machine-module-use-case` directory of our GitHub repository. After executing `terraform init`, we can see the following activities being performed by Terraform, and in the listed activities, it has downloaded the following module from our GitHub repository:

```
$terraform init  
Initializing modules...  
Downloading github.com/PacktPublishing/HashiCorp-  
Infrastructure-Automation-Certification-Guide.git?ref=v0.0.1  
for terraform-vm...  
- terraform-vm in .terraform\modules\terraform-vm\chapter7\  
azurerm\azurerm-virtual-machine-module  
Initializing the backend...  
Successfully configured the backend "azurerm"! Terraform will  
automatically use this backend unless the backend configuration  
changes.  
Initializing provider plugins...  
- Finding hashicorp/azurerm versions matching "2.55.0"...  
- Finding latest version of hashicorp/random...  
- Installing hashicorp/azurerm v2.55.0...  
- Installed hashicorp/azurerm v2.55.0 (signed by HashiCorp)  
- Installing hashicorp/random v3.0.0...  
- Installed hashicorp/random v3.0.0 (signed by HashiCorp)  
The following providers do not have any version constraints in  
configuration, so the latest version was installed.  
To prevent automatic upgrades to new major versions  
that may contain breaking changes, we recommend adding  
version constraints in a required_providers block in your  
configuration, with the constraint strings suggested below.
```

```
* hashicorp/random: version = "~> 3.0.0"
```

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.

Further, we executed `terraform plan` and `terraform apply`. We can see the following output, which has been defined in the `outputs.tf` file present inside the `azurerms-virtual-machine-module-use-case` directory:

Outputs:

```
vm_id = /subscriptions/97c3799f-2753-40b7-a4bd-157ab464d8fe/
resourceGroups/Terraform-rg/providers/Microsoft.Compute/
virtualMachines/Terraform-vm
```

```
vm_name = Terraform-vm
```

```
vm_private_ip = 10.1.1.4
```

And finally, our defined module managed to create resources in Azure, which can be seen in the following screenshot:

The screenshot shows the Microsoft Azure portal interface. The main content area displays the 'Terraform-rg' resource group. The 'Essentials' section shows the subscription ID, location (East US), and tags (Environment: prod, Owner: Azure-Terraform). Below this, a table lists the resources created within the resource group:

Name	Type	Location
Terraform-nic	Network interface	East US
Terraform-vm	Virtual machine	East US
terraform-vm-keyvault	Key vault	East US
Terraform-vm_OsDisk_1_9faed3fcbbb344779db7290e847d1a79	Disk	East US
Terraform-vnet	Virtual network	East US

Figure 7.2 – Azure resources

Important note

To execute all Terraform code specific to Azure, you need to have authentication to Azure in place, which we have already explained in *Chapter 5, Terraform CLI*.

From this section, we managed to write a Terraform module for the Azure VM with keyvault and were easily able to consume it. In the same way, you could write a module for any other Azure resources and get it published to Terraform Cloud or any other repository from where it would be easy for you to consume it as and when needed.

After learning how to write Terraform modules for Azure, you will now be interested in learning about writing Terraform modules for AWS. Therefore, we are going to discuss further how these can be written and consumed in our next section.

Writing Terraform modules for AWS

After successfully writing an Azure module in our previous section, you might be wondering: *Is there is any difference between drafting Terraform modules for an AWS provider as compared to an azure provider?* To answer that question, concept-wise it's the same, but we just need to take care of the specific AWS resource/service arguments supported. Thus, learning about Terraform modules for AWS shouldn't be a big deal for you. We should be able to easily draft AWS modules and consume them for infrastructure provisioning in AWS. Let's try to understand how we can write a module for AWS. To explain this, we have taken an example of a VPC with a subnet. We have written modules and published them into our GitHub repository at <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter7/aws/aws-vpc-subnet-module>.

This is a list of the files that have been placed into the `aws` folder of our GitHub repository:

```
inmishrar@terraform-lab-vm: ~/HashiCorp-Infrastructure-
Automation-Certification-Guide/chapter7/aws# tree
.
├── aws-vpc-subnet-module
│   ├── VERSION
│   ├── main.tf
│   ├── outputs.tf
│   └── variables.tf
└── aws-vpc-subnet-module-use-case
```

```
|— main.tf
|— outputs.tf
|— providers.tf
|— terraform.tfvars
└— variables.tf
```

```
2 directories, 9 files
```

Let's discuss how we have written that module. In the `main.tf` file, we have placed the resource code block of the VPC and subnet that's been taken from <https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/vpc> and <https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/subnet>. You can get the following code from our GitHub repository:

```
resource "aws_vpc" "terraform-vpc" {
  cidr_block      = var.cidr_block
  instance_tenancy = "default"
  tags = {
    Name = var.vpc_name
  }
}

resource "aws_subnet" "terraform-subnet" {
  vpc_id      = aws_vpc.terraform-vpc.id
  cidr_block = cidrsubnet(var.cidr_block, 8, 1)
  tags = {
    Name = var.subnet_name
  }
}
```

In `variables.tf`, we need to ensure that we have declared all the input variables. We have defined the following code block, which you can copy from our GitHub repository:

```
variable "vpc_name" {
  type      = string
  description = "vpc name"
}

variable "cidr_block" {
```

```
type          = string
description   = "address space of the vpc"
default      = "10.0.0.0/16"
}
variable "subnet_name" {
  type          = string
  description   = "subnet name"
}
```

In `outputs.tf`, you see the output name and its respective values that are coming from the attribute that can be exported from the VPC resource (that is, <https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/vpc>). We have defined the following code block; you can define an attribute that can be exported accordingly:

```
output "vpc_id" {
  value = aws_vpc.terraform-vpc.id
}
output "vpc_cidr_block" {
  value = aws_vpc.terraform-vpc.cidr_block
}
```

All this code helped us to develop a module named `vpc` with `subnet` and publish that module to our GitHub repository. Now, you may be wondering how we can consume this published module. To understand this, we have defined the next code block.

In our `main.tf` file, we need to write a small line of code that has all the variables as arguments that have been defined while writing the module, as follows:

```
module "terraform-aws-vpc" {
  source      = "github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git//chapter7/
aws/aws-vpc-subnet-module?ref=v1.0.0"
  vpc_name    = var.vpc_name
  cidr_block  = var.cidr_block
  subnet_name = var.subnet_name
}
```

Here, in our `main.tf` file, we mentioned `source`, which is pointing to our GitHub repository and having a version number defined using `ref`, and which we created as a `release/tag` to our GitHub repository, as you can see in the following screenshot. This will help us to download a specific version of the module:

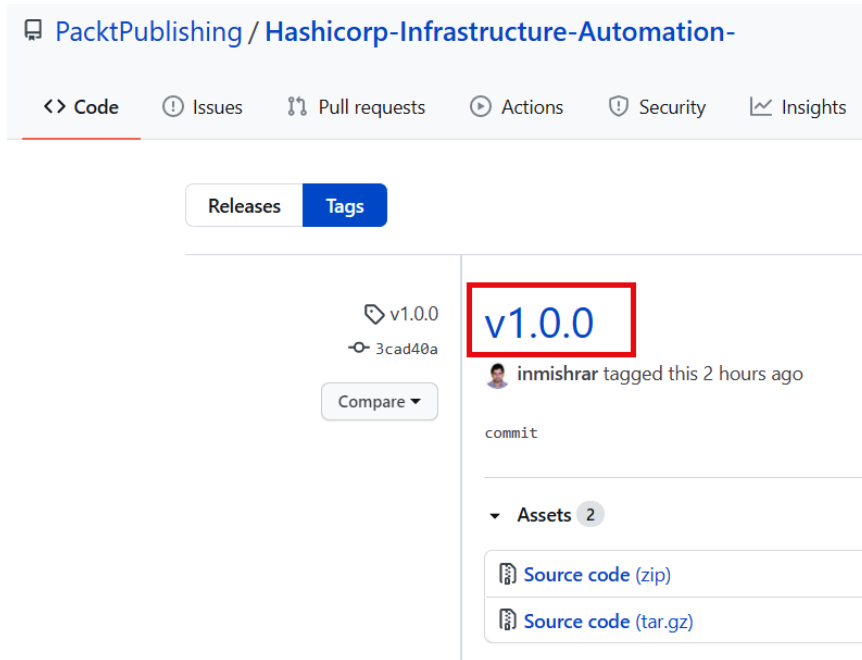


Figure 7.3 – GitHub release v1.0.0

In `variables.tf`, we define the following code block:

```
variable "vpc_name" {
  type      = string
  description = "vpc name"
}
variable "cidr_block" {
  type      = string
  description = "address space of the vpc"
```

```
    default      = "10.0.0.0/16"
  }
  variable "subnet_name" {
    type      = string
    description = "subnet name"
  }
  variable "region" {
    type      = string
    description = "provide region where you want to deploy
resources"
  }
}
```

In the following code block, you see that it has almost the same variables that were defined while creating the module, to make this easy for you to understand. As you can see, there is one variable named `region` that we have defined in the `providers.tf` file:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

provider "aws" {
  region = var.region
}
```

Finally, we have taken all the values of the defined variables from the `terraform.tfvars` file, as illustrated in the following code snippet:

```
subnet_name = "Terraform-aws-subnet"
vpc_name     = "Terraform-aws-vpc"
cidr_block   = "10.0.0.0/16"
region      = "us-east-1"
```

To get output from the module, we have defined an `outputs.tf` file that contains the following code:

```
output "vpc_id" {
  value = module.terraform-aws-vpc.vpc_id
}
output "vpc_cidr_block" {
  value = module.terraform-aws-vpc.vpc_cidr_block
}
```

When you execute `terraform init`, you can see the following activities have been performed by the Terraform tool, and it clearly shows that it has downloaded the module from our GitHub repository:

```
$terraform init
Initializing modules...
Downloading github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git?ref=v1.0.0
for terraform-aws-vpc...
- terraform-aws-vpc in .terraform\modules\terraform-aws-vpc\
chapter7\aws\aws-vpc-subnet-module

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching "~> 3.0"...
- Installing hashicorp/aws v3.22.0...
- Installed hashicorp/aws v3.22.0 (signed by HashiCorp)
```

any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.

After executing `terraform plan` and `terraform apply`, we can see the following output that we have defined in the `outputs.tf` file:

```

Outputs:
vpc_cidr_block = 10.0.0.0/16
vpc_id = vpc-0636e37e2a87b6d5a

```

From the AWS Console, we can see that a VPC with subnet got created, as shown in the following screenshot:

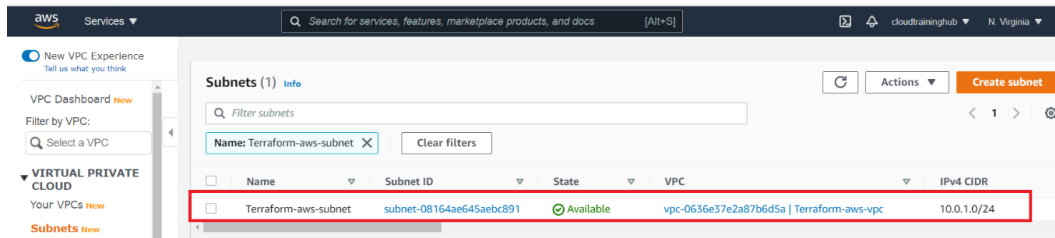


Figure 7.4 – AWS resources

From this section, you should have understood how to write a module for AWS, and along with that, you should have learned how modules can be consumed in your day-to-day infrastructure deployment. Let's try to use this knowledge and see how we can write a module for GCP and provision our infrastructure using that module in GCP.

Writing Terraform modules for GCP

We now have a fair understanding of writing modules for AWS and Azure. Now, we have a question for you: *What are your thoughts regarding Terraform modules for GCP?* Will there be any difference as compared to AWS and Azure modules? If you are thinking that writing Terraform modules for GCP would be difficult and different from your previous experience with AWS and Azure, then you are wrong, as you will be able to use this previous learning. Let's try to write a module for GCP. To explain this, we have taken a GCP resource named `google cloud storage bucket`. We have written all the required Terraform code and placed it in our GitHub repository (that is, <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter7/gcp>).

Here is a list of files that have been placed under the `gcp` folder of our GitHub repository:

```
inmishrar@terraform-lab-vm:~/HashiCorp-Infrastructure-
Automation-Certification-Guide/chapter7/gcp# tree
```

```
.
├── gcp-storage-module
│   ├── main.tf
│   ├── outputs.tf
│   └── variables.tf
└── gcp-storage-module-use-case
    ├── main.tf
    ├── outputs.tf
    ├── providers.tf
    ├── terraform-lab-project-001c988dafca.json
    ├── terraform.tfvars
    └── variables.tf
```

```
2 directories, 9 files
```

While writing the GCP storage module, we created three files: `main.tf`, `outputs.tf`, and `variables.tf`.

In the `main.tf` file, the following defined code is present:

```
resource "google_storage_bucket" "gcp-stg" {
  name           = var.gcp_stg_name
  location       = var.gcp_location
  force_destroy = var.force_destroy
  storage_class  = var.storage_class
  project        = var.project
  labels         = var.labels
  versioning {
    enabled = true
  }
}
```

We define most of the values as a variable so that the user can provide input variable values in their own way, and this increases the consumption of the Terraform module.

Next, we declare all the variables in the `variables.tf` file, as follows:

```
variable "gcp_stg_name" {
  type      = string
  description = "name of the GCP storage"
}

variable "gcp_location" {
  type      = string
  description = "name of the location"
}

variable "force_destroy" {
  type      = bool
  description = "provide whether true or false"
  default   = true
}

variable "storage_class" {
  type      = string
  description = "Provide Storage Class and Supported values
include: MULTI_REGIONAL, REGIONAL, NEARLINE, COLDLINE, ARCHIVE"
}

variable "project" {
  type      = string
  description = "provide project ID"
}

variable "labels" {
  type      = map
  description = "provide name of the labels"
}
```

Once the Google storage bucket gets created, we need to see its output, so the following code is present in the `outputs.tf` file:

```
output "gcs_self_link" {
  value = google_storage_bucket.gcp-stg.self_link
}

output "gcs_url" {
  value = google_storage_bucket.gcp-stg.url
}
```

```

}
output "gcs_name" {
  value = google_storage_bucket.gcp-stg.name
}

```

So, after writing the module and publishing it on our GitHub repository, we created a release/tag (that is, v2.0.0), which you can see in the following screenshot:

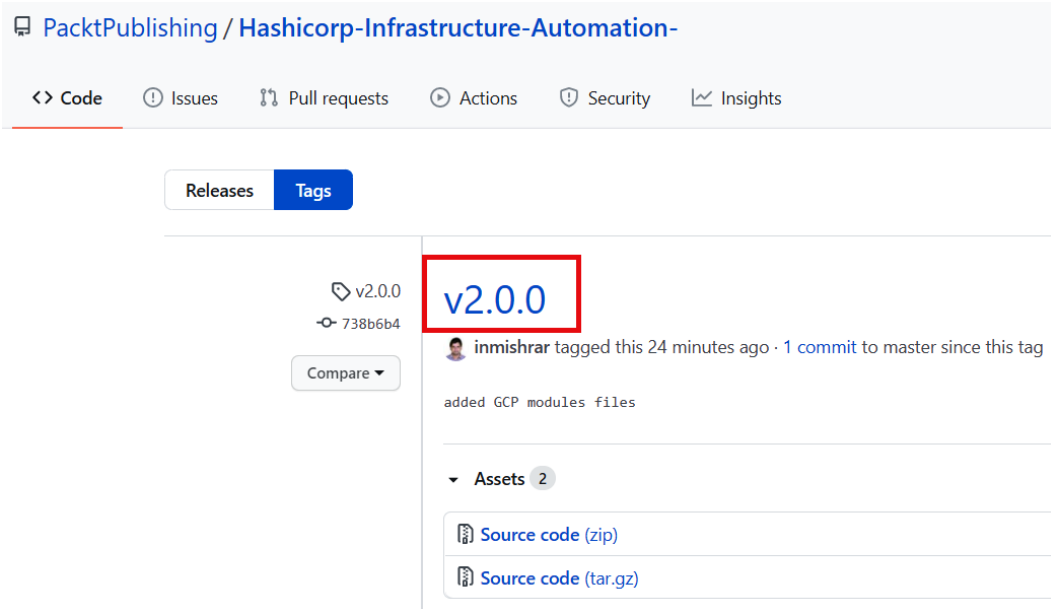


Figure 7.5 – GitHub release v2.0.0

To consume the published module, you can directly take it from our GitHub repository; we won't discuss the code here. Just after that, you can run `terraform init`, which will provide you a complete insight into all activities Terraform is performing. The code is shown in the following snippet:

```

$terraform init
Initializing modules...
Downloading github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git?ref=v2.0.0
for terraform-gcp-gcs...
- terraform-gcp-gcs in .terraform\modules\terraform-gcp-gcs\
chapter7\gcp\gcp-storage-module
Initializing the backend...

```

Initializing provider plugins...

```
- Finding hashicorp/google versions matching "~> 3.0"...
- Installing hashicorp/google v3.51.0...
- Installed hashicorp/google v3.51.0 (signed by HashiCorp)

Terraform has been successfully initialized!
```

In the sequence following the Terraform workflow, you can run the `terraform plan` and `terraform apply` commands. This will provision the Google storage bucket and show us the following output, which we mentioned in our `outputs.tf` file:

Outputs:

```
gcs_name = gcpstg23423
```

```
gcs_self_link = https://www.googleapis.com/storage/v1/b/gcpstg23423
```

```
gcs_url = gs://gcpstg23423
```

We can confirm by checking the Google Cloud Console that a Google storage bucket got created, as you can see in the following screenshot:

<input type="checkbox"/>	Name ↑	Created	Location type	Location	Default storage class	Updated	Public access	Access control	Lifecycle rules
<input type="checkbox"/>	gcpstg234...	Dec 28, 2020, 12:30:31 AM	Multi-region	us (multiple re...	Standard	Dec 28, 2020, 12:30:31 AM	Subject to object ACLs	Fine-grained	None

Figure 7.6 – Google Cloud resources

Important note

To execute all Terraform code specific to GCP, you need to have authentication to GCP in place—this has been already discussed in *Chapter 5, Terraform CLI*. We published our Google service account credential file on GitHub just for demonstration purposes so that you can see that you need to have a credential file to authenticate Terraform to the GCP cloud platform. It is recommended that you store this credential file in your local system or in environment variables—never push/publish any GCP credential file to a GitHub project.

From this section, we learned about writing Terraform module code for Google Cloud, how it can be published on GitHub, and how effectively we can consume it according to our requirements. We also understood how a specific version of a module can be called by defining a `source` in the module code block. Using this created GCP module, we managed to deploy GCS. Similarly, we can use this knowledge to write other GCP modules and get them consumed according to our requirements.

What if we want to draft a module and make it available to everyone by contributing and getting our modules published to Terraform Registry? We are going to explain this concept in our next section.

Publishing Terraform modules

We have already learned how to publish Terraform modules to a GitHub repository specific to your project. How about if you want to contribute to the Terraform community? For that, HashiCorp provides you with an option of publishing your well-written code to Terraform Registry.

Anyone can write a module and get it published to Terraform Registry. Terraform Registry supports versioning and generates documentation, and you can even browse all the version history. Always remember to try to write more generic modules so that they are reusable. All the published public modules are managed through Git and GitHub. Writing and publishing a module is very easy and doesn't take much time; once a module gets published, you can get it updated by pushing the updated code with the respective Git tag following proper semantic versioning (that is, either `v1.03` or `0.6.0`).

Key requirements

To publish a module to Terraform Registry, the following key requirements should be in place:

- **GitHub:** The module must be available on GitHub as a public repository.
- **Named:** Naming of the published module should be done properly. It must follow a prescribed syntax (that is, `terraform-<PROVIDER>-<NAME>`). Examples of this are `terraform-google-storage-bucket` or `terraform-aws-ec2-instance`.
- **Repository description:** A simple one-line description of the GitHub repository is needed because this is used to populate a short description of the module.
- **Standard module structure:** The module you are publishing should adhere to the standard structure of a module. This will allow the registry to generate documentation, track resource usage, parse submodules and examples, and more.
- **x.y.z tags for releases:** A published module needs to have proper versioning, which is done using release tags. The Terraform Registry is able to identify specific versions of modules through a release tag. Release tags should follow semantic versioning (that is, `v1.0.0` or simply `0.5.0`). You can define them either way.

Publishing a module

After writing a module by following the requirements mentioned earlier, you can visit <https://registry.terraform.io/> and select the **publish module** option, which will redirect you to the sign-in page, where you can log in with your GitHub account. If you don't have a GitHub account, then you will be required to sign up for one. There is no other option to log in to Terraform Registry, as this is currently only allowed from GitHub. Once you are logged in, it will ask you to select a specific module that has a proper naming convention, as you can see in the following screenshot:

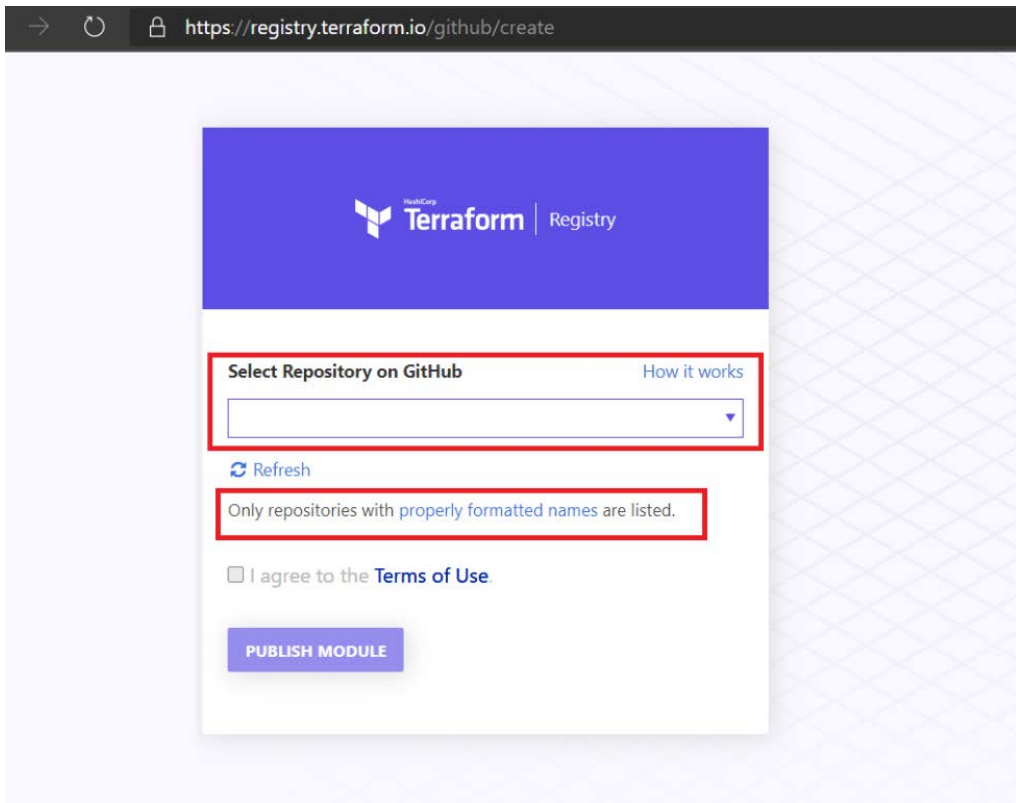


Figure 7.7 – Select a GitHub repository

After selecting a specific repository on GitHub, you can click on **PUBLISH MODULE**. Your module will then shortly get published and will be available for use, provided your written modules have followed the correct sort of formatting. The published module will be visible to you on Terraform Registry, as shown in the following screenshot. For your understanding, we have shown a screenshot of the AWS VPC module. Your published module name will vary, depending totally upon the module name and provider:

The screenshot shows the Terraform Registry page for the 'aws vpc' module. The URL is <https://registry.terraform.io/modules/terraform-aws-modules/vpc/aws/latest>. The page features a blue header with the Terraform logo and a search bar. The main content area displays the 'aws vpc' logo (circled in red), the version '2.66.0 (latest)', and a 'Provision Instructions' box (circled in red) containing the following Terraform code snippet:

```
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
  version = "2.66.0"
  # Insert the 15 required variables here
}
```

Below the code snippet, there are links for 'Readme', 'Inputs (409)', 'Outputs (288)', 'Dependency (1)', and 'Resources (142)'. The page also includes a 'code helpers' section with a 'tag v2.66.0' button.

AWS VPC Terraform module

code helpers **id** tag **v2.66.0**

Terraform module which creates VPC resources on AWS.

Figure 7.8 – Terraform published module

From this section, you will have got an idea of how you can draft a module and get it published to Terraform Registry. This is one way you can contribute to the Terraform open source community.

Summary

From this complete chapter, you have gained an understanding of Terraform modules and the different arguments and meta-arguments that are supported by Terraform modules, such as `depends_on`, `providers`, `source`, and `version`. Moving further on, we also discussed how you can write Terraform modules for Azure, AWS, and GCP. We then discussed how they can be consumed, and finally, we discussed how you can contribute to the Terraform community by writing and publishing your modules to Terraform Registry. With this knowledge, you will be able to draft a Terraform module and consume it for the deployment or manageability of an enterprise infrastructure.

In our next chapter, we are going to discuss Terraform configuration files and which industry best practices can be followed while writing Terraform configuration files, covering all three cloud providers, *Azure*, *AWS*, and *GCP*.

Questions

The answers to the following questions can be found in the *Assessments* section at the end of this book:

1. Which of the following source definitions for a module is incorrect?
 - A. `../../terraform-module`
 - B. `github.com/terraform-module`
 - C. `\github.com/terraform-module`
 - D. `bitbucket.com/terraform-module`
2. You have published an AWS **Elastic Compute Cloud (EC2)** instance module to the Bitbucket repository. What will happen when you run the `terraform init` command?
 - A. It downloads the EC2 module to the local directory.
 - B. It deletes the EC2 module from the repository.
 - C. It updates the version of the EC2 module.
 - D. It moves the EC2 instance module to other subdirectories.
3. What is the correct syntax for defining output from a module?
 - A. `module.terraform-aws-vpc.vpc_id`
 - B. `terraform-aws-vpc.vpc_id`
 - C. `vpc_id`
 - D. `terraform-aws-vpc`
4. You are working with many developers and they are all tired of writing their Terraform **Infrastructure as Code (IaC)**. Which of the following options would you recommend to them so that they can have reusable IaC?
 - A. Terraform data sources
 - B. Terraform locals
 - C. Terraform module
 - D. Terraform resource

5. From the following defined code snippet, see if you can tell what the version of the module is:

```
module "servers" {  
  source = "./app-cluster"  
  servers = 5  
}
```

Which of these options is correct?

- A. 5
- B. No version
- C. >5
- D. <5

Further reading

You can check out the following links for more information about the topics that were covered in this chapter:

- Terraform module overview: <https://www.terraform.io/docs/configuration/blocks/modules/index.html>
- Terraform module sources: <https://www.terraform.io/docs/modules/sources.html>
- Terraform module publishing: <https://www.terraform.io/docs/modules/publish.html>
- Terraform tutorials: https://learn.hashicorp.com/terraform?_ga=2.137479925.498906489.1609060300-1551442629.1609060300

Section 3: Managing Infrastructure with Terraform

This part of the book gives you some real-time experience, implementing the knowledge of preparing stacks using Terraform code. Terraform configuration files consist of all the defined resources, modules, variables, output, and so on. This chapter helps you to deploy and upgrade complex infrastructure using stacks and modules.

Along with this, you will get insights into the Terraform cloud and its features such as policy as code, that is, Sentinel.

The following chapters will be covered under this section:

- *Chapter 8, Terraform Configuration Files*
- *Chapter 9, Understanding Terraform Stacks*
- *Chapter 10, Terraform Cloud and Terraform Enterprise*
- *Chapter 11, Terraform Glossary*

8

Terraform Configuration Files

In our last chapter, we discussed Terraform modules, module use cases, and different arguments supported by the modules. We also discussed how we can contribute to the Terraform community by writing Terraform modules and getting them published to the Terraform registry so that everyone is able to consume them.

In this chapter, we are planning to discuss industry best practices for writing Terraform configuration files. While drafting Terraform configuration, we will be using our previous learning such as resources, data sources, locals, variables, and modules. Further, we will be discussing how to write a configuration file for the respective cloud platforms of *GCP*, *Azure*, and *AWS*.

The following topics will be covered in this chapter:

- Understanding Terraform configuration files
- Writing Terraform configuration files for GCP
- Writing Terraform configuration files for AWS
- Writing Terraform configuration files for Azure

Technical requirements

To follow this chapter, you need to have an understanding of the Terraform CLI. You should know about the installation of Terraform on various machines. You need to have some basic understanding of writing the Terraform configuration file. You can find all the code used in this chapter at the GitHub link: <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter8>.

Check out the following link to see the Code in Action video:

<https://bit.ly/3hPvkiR>

Understanding Terraform configuration files

We keep mentioning Terraform configuration files throughout the book but so far we haven't had a chance to discuss this in detail. We will get a thorough understanding of Terraform configuration files in this chapter.

A Terraform configuration file is a well-defined and structured file written in the Terraform language that tells Terraform how to manage the complete infrastructure. A Terraform configuration file can have one or more files and directories.

A Terraform configuration file can be written in two formats. A Terraform configuration file written with HashiCorp configuration language is the recommended approach for the writing configuration files and it ends with a file extension, that is, `.tf`. There is one more way of writing a Terraform configuration file, which ends with `.tf.json`. The configuration files that end with `.tf` are human-readable and the JSON format is machine-readable. Every cloud provider, such as AWS, GCP, Azure, and so on, has exposed their service/resource APIs as an endpoint that needs to be defined in the configuration file. Terraform has the capability to understand how to call providers' APIs. The services/resources of the providers needs to be defined in the Terraform configuration file and accordingly it uses the respective cloud provider's service APIs to manage those endpoints.

Terraform native configuration syntax

Here is the defined syntax of the human-readable configuration file in the Terraform language:

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {  
  # Block body  
  <IDENTIFIER> = <EXPRESSION> # Argument  
}
```

Let's discuss this syntax further:

- **Blocks** are the containers that represent the configuration of the objects. Having a block type generally represents what functionality Terraform is planning to perform, such as a resource, data, and so on. Block type has one or more block labels, and has a body that contains arguments and nested blocks. In our following code block for the resource, Terraform needs to have two block labels, that is, "google_storage_bucket" "gcp-stg". After defining the block type and labels, the block body is defined within the delimiter, that is, the { and } characters. Within the block body, we can define arguments and blocks that may be nested.
- **Arguments** help to assign a value to a particular name. The identifier that has been defined before the equal sign is the name of the argument and after the equal sign has the argument value, for example, the expression `image_id = "xyz354"`.
- **Identifiers** could be argument names, block names, and block type names such as resources, data, and input variables. The identifiers contain letters, digits, underscores (_), and hyphens (-). Remember, an identifier name shouldn't start with a digit as it can create confusion with literal numbers so it must always start with a letter.
- **Expression** is the respective value that has been defined against a specific identifier. It could be a direct value or reference or combine multiple other values.

The Terraform language is declarative, has all the steps to reach the defined goal. The order in which the blocks and files have been written is not so important. Terraform only understands the implicit and explicit dependencies and accordingly, it follows the sequence of the resource deployment.

Just to give you a sense of how the Terraform language is written with its complete syntax, we have taken an example code of a Google cloud storage bucket; following this syntax, you can write a Terraform code block for any service or resource:

```
# Create Google Storage Bucket
resource "google_storage_bucket" "gcp-stg" {
  name          = "gcp-bucket"
  location      = "US"
  versioning {
    enabled = true
  }
}
```


In the preceding code, you can see that we have defined a comment `# Create Google Storage Bucket`. The following are the ways in which we can define comments in the code block:

- `#`: Used for a single-line comment.
- `//`: This has also been used for a single-line comment and can work as an alternative to `#`.
- `/*` and `*/`: You can define code within this delimiter; Terraform will treat it as a comment.

Out of these defined comment methods, `#` is the default one. If you had defined the comment with `//`, the automatic formatting tools will convert it to `#` as the double-slash style is not idiomatic.

Terraform configuration files support UTF-8 encoding. The delimiters used in the configuration file are all ASCII characters and accept non-ASCII characters for identifiers, comments, and string values.

Terraform configuration files support both Unix and Windows line ending styles. The automatic formatting tools may transform Windows CRLF to Linux LF automatically.

Terraform override file

We learned that Terraform manages to read `.tf` or `.tf.json` files within a directory and each file will have a different set of the configuration so that there won't be any conflict. What will happen if we define the same configuration in the two different files and the second file contains some additional argument? For example, if there are two files, that is, `main.tf` and `main.tf.json` files, within the directory, then definitely when you execute Terraform workflows such as `terraform plan` and `terraform apply`, it will throw an error as shown in the following code block:

```
$ terraform plan
Error: Duplicate resource "azurerm_resource_group"
configuration
  on main.tf.json line 4, in resource.azurerm_resource_group:
   4:         "rgname":{
A azurerm_resource_group resource named "rgname" was already
declared at main.tf:1,1-43. Resource names must be unique per
type in each module.
Error: Duplicate variable declaration
  on main.tf.json line 16, in variable:
```

```

16:         "location": {
A variable named "location" was already declared at main.
tf:10,1-20. Variable names must be unique within a module.
Error: Duplicate output definition
  on main.tf.json line 21, in output:
21:         "id":{
An output named "id" was already defined at main.tf:14,1-12.
Output names must
be unique within a module.
Error: Duplicate output definition
  on main.tf.json line 24, in output:
24:         "name":{
An output named "name" was already defined at main.tf:17,1-14.
Output names must be unique within a module.

```

We can see that it is giving us an error showing duplicate entries. Now the question is how we can avoid this.

In many cases, you might be interested in overriding the values of the specific argument of the existing configuration object by using a separate file. For example, you wish to override some of the configurations written by the programmatically generated file in JSON syntax with a human-edited configuration file written in the native Terraform language.

For such scenarios, Terraform has a special handling option for files ending with `_override.tf`, `override.tf`, `_override.tf.json`, or `override.tf.json`.

If you have defined an `override` file while loading the configuration files, Terraform initially will skip these files and later on try to execute it sequentially, finding the respective argument values in the main file and the `override` file. Finally, it helps you prepare a configuration with the latest values, which it got from the `override` file, which means it will try to merge and overwrite the configuration files.

It is recommended that when you are using `override` files, you should define proper comments in the code so that the reader can understand what made you create an `override` file.

Let's try to explain how Terraform works with the `override` file:

We have defined the following code block in `main.tf`:

```

resource "azure_rm_resource_group" "rgname" {
  name      = "Terraform-rg"
  location = var.location

```

```
tags = {  
  "environment" = "development"  
  "costcenter"  = "B3478"  
}  
}
```

And in the `override.tf` file, the following code has been defined:

```
# trying to update tags using override file  
resource "azurerm_resource_group" "rgname" {  
  tags = {  
    "environment" = "preprod"  
    "costcenter"  = "C3478"  
  }  
}
```

Let's see what happens when we execute `terraform plan`. As you can see that tags is being updated with the latest values, which means Terraform reads the contents from the override file and it gets updated:

```
$ terraform plan
```

```
Refreshing Terraform state in-memory prior to plan...
```

```
The refreshed state will be used to calculate this plan, but  
will not be persisted to local or remote state storage.
```

```
-----  
-----
```

```
An execution plan has been generated and is shown below.
```

```
Resource actions are indicated with the following symbols:
```

```
+ create
```

```
Terraform will perform the following actions:
```

```
# azurerm_resource_group.rgname will be created
```

```
+ resource "azurerm_resource_group" "rgname" {
```

```
  + id          = (known after apply)
```

```
  + location   = "eastus"
```

```
  + name       = "Terraform-rg"
```

```
  + tags       = {
```

```
    + "costcenter" = "C3478"
```

```
    + "environment" = "preprod"
```

```

    }
  }
}
Plan: 1 to add, 0 to change, 0 to destroy.

```

We are not discussing all the possible cases with the `override` file; you can refer to <https://www.terraform.io/docs/configuration/override.html>.

JSON configuration syntax

We have seen how easily we can write a Terraform configuration file, which is a human-readable format. Let's try to understand how it is different from the machine-readable Terraform configuration file, which is a JSON file. This JSON file, which ends with `.tf.json`, is generated programmatically. Sometimes it is very difficult to write all the construct of native Terraform configuration code to JSON due to limitations of JSON grammar. It is not necessary to know HCL syntax mapping with JSON syntax. We will try to discuss it with one example:

```

{
  "resource": {
    "azure_rm_resource_group": {
      "rgname": {
        "://" : "Create resource group in Azure",
        "name": "Terraform-rg",
        "location": "${var.location}",
        "tags": {
          "environment": "development",
          "costcenter": "B41892"
        }
      }
    }
  },
  "variable": {
    "location": {
      "default": "string",
      "default": "eastus"
    }
  },
  "output": {

```

```
    "id": {
      "value": "${azurerm_resource_group.rgname.id}"
    },
    "name": {
      "value": "${azurerm_resource_group.rgname.name}"
    }
  }
}
```

In the previously defined code block, we have written code using JSON configuration syntax, where you can see that, at the root of a JSON based Terraform configuration file, it has a JSON object. The top-level object property should match with the name of the top-level block types. Suppose you are defining a block type such as a `resource` that expects two labels, so it will have two levels of nesting. Similarly for the `variable` where we have only one label, it follows one level of nesting of JSON objects.

In the preceding JSON code block, Terraform will ignore any value defined with `///. So you can use this for placing comments within the JSON code block.`

For referencing any value in the code, we have to define the respective contents with the specified syntax, that is, `"${<reference name>}"` as we defined in the output code block for getting `id` value, that is, `"${azurerm_resource_group.rgname.id}"`.

We have written same JSON code in the Terraform language and placed it as a `main.tf` into our GitHub repository, that is, <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter8/Override-Files>:

```
resource "azurerm_resource_group" "rgname" {
  #Create resource group in Azure
  name      = "Terraform-rg"
  location  = var.location
  tags = {
    "environment" = "development"
    "costcenter"  = "B3478"
  }
}

variable "location" {
  type    = string
}
```

```
    default = "eastus"
  }
  output "id" {
    value = "azurerm_resource_group.rgname.id"
  }
  output "name" {
    value = "azurerm_resource_group.rgname.name"
  }
}
```

We will not discuss the entire JSON code block; for further reading, you can visit <https://www.terraform.io/docs/configuration/syntax-json.html>.

Data types

After learning about the Terraform configuration file, that is, `.tf` or `.tf.json` files, you must have questions about the data types supported by both the configuration file types. Let's see the different data types supported by the human-readable Terraform language:

- **string:** This represents the sequence of the characters with some text, such as `«Hello-Terraform»`.
- **number:** This is a numeric value representing both whole numbers such as `10` and fractional values such as `6.28`.
- **bool:** This represents a Boolean value, either `true` or `false`.
- **list (or tuple):** This represent a sequence of values, such as `[«abc», 15, false]`. Elements in a list or tuple are identified with the whole number, starting with zero.
- **map (or object):** This is a data type where it will have a key and value such as `{name = «Terraform», environment = «development»}`. In the map data type, it can be separated by a comma or a line break.
- **null:** This is a special value that won't have any data type. Terraform will not hold any value if you set an argument of a resource or module to `null`.

String, number, and bool data types are also called *primitive types*. Lists/tuples and maps/objects are also called *complex types*, *structural types*, or *collection types*.

We can do mapping of the data type of the Terraform language to JSON in the following way:

JSON	Terraform Language Interpretation
Boolean	A literal <code>bool</code> value.
Number	A literal <code>number</code> value.
String	Parsed as a <code>string template</code> and then evaluated as described below.
Object	Each property value is mapped per this table, producing an <code>object(...)</code> value with suitable attribute types.
Array	Each element is mapped per this table, producing a <code>tuple(...)</code> value with suitable element types.
Null	A literal <code>null</code> .

Figure 8.1 – Terraform data types

Writing the Terraform configuration content in a proper format is very important because it not only makes it readable for the user, but also helps you to have proper alignment of the code. So, let's discuss in detail the different style conventions Terraform supports or follows.

Terraform style conventions

A Terraform configuration file has an idiomatic style convention, which we encourage users to follow while writing the files. This will help you to maintain consistency across all the Terraform configuration files. Automatic source code formatting tools will be able to apply the following conventions:

- Each nesting level needs two spaces of indentation.
- For all the arguments and their values that have been defined consecutively at the same nesting level, the equals sign should be aligned:

```
ami           = "xyz356"  
instance_type = "t1.micro"
```

- Inside a block body, when you have both arguments and blocks together, place all the arguments together at the top and nested blocks should be placed below them. You can use one blank line to separate both arguments and the blocks.

- Use empty lines to separate logical groups of arguments within a block.
- For blocks containing both arguments and "meta-arguments" (as defined by the Terraform language semantics), separate meta-arguments from other arguments with one blank line. Define meta-argument blocks at the very end and separate them from other blocks with one blank line:

```
resource "aws_instance" "Terraform-ec2" {  
  count          = 3 # meta-argument first  
  ami           = "xyz139"  
  instance_type = "t2.micro"  
  
  network_interface {  
    # ...  
  }  
  
  lifecycle { # meta-argument block last  
    create_before_destroy = true  
  }  
}
```

- There should be one blank line that will separate top-level blocks from one another. Even nested blocks should also be separated by one blank line.

You don't need to remember all the Terraform style conventions. We will suggest a simple way of getting things done: first draft the Terraform configuration file then run the `terraform fmt -recursive` command. This command will perform all sorts of formatting to the Terraform configuration file.

In this section, we learned about the Terraform configuration file, how it is different from the JSON syntax, the best practices for writing Terraform configuration files, and the different data types supported by the Terraform language file and JSON. We also discussed the `override` file and saw a use case of it. We also discussed style conventions supported by Terraform.

We will use this learning and try to write a Terraform configuration file for different cloud providers such as GCP, AWS, and Azure. This will help us understand more about Terraform configuration files and industry best practices for drafting them.

Writing Terraform configuration files for GCP

We have learned about writing a Terraform configuration file. Let's concentrate and try to write a simple Terraform configuration file for the GCP cloud. To have a detailed explanation about the Terraform configuration file for GCP, we have considered the *Google cloud storage bucket* and *virtual private connection* services of Google.

The following files are present in our GitHub repository under the `gcp-files` directory, which you can clone to your local machine and start consuming:

```
inmishrar@terraform-lab-vm:~/HashiCorp-Infrastructure-Automation-Certification-Guide/chapter8/gcp-files# tree
```

```
.
├── gcp-files
│   ├── main.tf
│   ├── outputs.tf
│   ├── providers.tf
│   ├── terraform.tfvars
│   └── variables.tf
```

```
1 directory, 5 files
```

In our `main.tf` file, we have defined the following code block. The code is available in the GitHub repository of this chapter:

```
module "terraform-gcp-gcs" {
  source      = "github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide.git//chapter7/gcp/gcp-storage-module?ref=v2.0.0"
  gcp_stg_name = "${local.gcp_string}-gcpstg"
  gcp_location = var.gcp_location
  force_destroy = var.force_destroy
  storage_class = var.storage_class
  project      = var.project_id
  labels       = var.labels
}
...
```

If you look at the code defined in the `main.tf` file, you will see we have defined all the arguments' values as an input variable. You must be wondering why we haven't hardcoded all the arguments' values. The answer is, we want to make sure that our code is more reusable so in the future, if we need to update any values, then just by updating the values of the variables inside the `terraform.tfvars` file, we can get our service/resource, that is, the Google cloud storage bucket and VPC, updated with that information.

As an industry best practice, we have created separate files, that is, `variables.tf`, `terraform.tfvars`, `outputs.tf`, `providers.tf`, and `main.tf`, so that if we need to make any amendment to the files then we should easily be able to do it. Some users keep all the code in the single file itself, which introduces more complexity and the written code will be vague so it's better to place them in a separate file. Terraform is intelligent enough to combine all the `.tf` and `terraform.tfvars` files together and perform implicit and explicit dependency and accordingly execute configuration files in sequence.

The following are some of the key practices that can be followed while writing the Terraform configuration file for GCP:

- In `providers.tf`, you can mention credentials and provide a filename or file path; it is recommended to keep this file secure as it will be holding confidential information that would be used for accessing Google Cloud. You can define the `credentials` value in the environment variables so that it will remain secure or else you can define it in the `.gitignore` file so that when you push your Terraform configuration code for GCP, it shouldn't get uploaded to the source control. In order to set Terraform environment variables for GCP, you can refer to https://registry.terraform.io/providers/hashicorp/google/latest/docs/guides/getting_started and to know about `.gitignore`, you can read <https://git-scm.com/docs/gitignore>. You can use the following code block for the Google provider:

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    google = {
      version = "~> 3.0"
    }
  }
}

provider "google" {
  project = var.project_name
```

```
    region      = var.gcp_region
    zone        = var.zone
  }
```

- Run `terraform fmt -recursive` so that all the Terraform configuration files are canonically arranged and become clearer and more readable.
- Using the input variables rather than hardcoding any of the values of the arguments will help you to make an update as and when needed. This will make Terraform configuration code more reusable.
- Passing the input variable values from `terraform.tfvars` would be the preferred choice, so that Terraform is easily able to recognize and take values from it during the runtime itself.
- It is recommended to define output in the `outputs.tf` file so that you can validate which resource or service was created or updated.
- Using more and more modules in the Terraform configuration file is recommended because when you want to consume the module then the length of the code will be small enough and you will easily be able to reuse it again and again. In our example, we have used our *Google cloud storage bucket* modules that we wrote in *Chapter 7, Terraform Modules*.
- Use data sources in the Terraform configuration file when you need to read already existing services or resources in GCP.
- Use locals in the Terraform configuration file if you need to use a local variable within the code.
- Use Terraform inbuilt functions in the configuration files to perform some sort of automation and it will help you to convert the current data into the required format.
- To store the Terraform state file, you can store it securely in the *Google cloud storage bucket* as it contains some confidential information. For more information about the *Google Cloud Storage* encryption, you can refer to <https://cloud.google.com/storage/docs/encryption>.

When we execute `terraform apply -auto-approve`, we can see that Terraform manages to deploy our Google cloud storage bucket and VPC:

```
$ terraform apply -auto-approve
random_string.string_name: Creating...
random_string.string_name: Creation complete after 0s
[id=vqrhfe]
```

```
module.terraform-gcp-gcs.google_storage_bucket.gcp-stg:
  Creating...
module.terraform-gcp-gcs.google_storage_bucket.gcp-stg:
  Creation complete after 3s [id=vqrhfe-gcpstg]
google_compute_network.gcp-network: Creating...
google_compute_network.gcp-network: Creation complete after 13s
[id=projects/terraform-project-2342/global/networks/vqrhfe-vpc]
google_compute_subnetwork.gcp-subnetwork: Creating...
google_compute_subnetwork.gcp-subnetwork: Creation complete
after 14s [id=projects/terraform-project-2342/regions/us-west1/
subnetworks/vqrhfesubnet]
google_compute_address.internal_with_subnet_and_address:
  Creating...
google_compute_address.internal_with_subnet_and_address:
  Creation complete after 3s [id=projects/terraform-project-2342/
regions/us-west1/addresses/vqrhfeinternal]
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
Outputs:
compute_address_id = projects/terraform-project-2342/regions/
us-west1/addresses/vqrhfeinternal
gcs_name = vqrhfe-gcpstg
vpc_id = projects/terraform-project-2342/global/networks/
vqrhfe-vpc
```

From this section, you have now got an idea of how you can draft Terraform configuration files for GCP and what the recommended approach for writing them is.

We will use this knowledge to try to write Terraform configuration files for AWS and try to understand if there are any differences.

Writing Terraform configuration files for AWS

How easy is it for us to write a Terraform configuration file for the AWS cloud? Writing a Terraform configuration file for AWS is similar to writing one for GCP. For better understanding, we are going to write some Terraform configuration code for creating a VPC and S3 *bucket* in AWS and keep them in our GitHub repository, that is, <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter8/aws-files>.

The following files are present in our GitHub repository that you can clone and practice with:

```
inmishrar@terraform-lab-vm:~/HashiCorp-Infrastructure-Automation-Certification-Guide/chapter8/aws-files# tree
.
├── aws-files
│   ├── main.tf
│   ├── outputs.tf
│   ├── providers.tf
│   ├── terraform.tfvars
│   └── variables.tf

```

1 directory, 5 files

In our `main.tf` file, we have written a code block for a *VPC* modules and an *S3* bucket; you can clone it from our GitHub and practice with it further. The following is the code present in the `main.tf` file:

```
module "terraform-aws-vpc" {
  source      = "github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide.git//chapter7/aws/aws-vpc-subnet-module?ref=v1.0.0"
  vpc_name    = var.vpc_name
  cidr_block  = var.cidr_block
  subnet_name = var.subnet_name
}

resource "aws_s3_bucket" "s3_bucket" {
  bucket = var.bucket_name
  acl    = var.bucket_acl
  tags   = var.custom_tags
}
```

In the preceding code block, you will notice we have defined all the arguments' values as input variables so that it will be easy for us to pass those input variables' values during the runtime itself.

Important note

Remember the AWS S3 bucket name should always be unique so it is better if you can use a random string from a Terraform resource code block while defining its name, so that you receive a unique name for the S3 bucket.

We will not explain the contents of each and every file, that is, `variables.tf`, `terraform.tfvars`, `outputs.tf`, and `providers.tf`. You can visit our GitHub and check the code described in the respective files. There is not much difference; we have used our previous knowledge of writing a Terraform configuration code block from the GCP section and followed the same approach while writing the Terraform configuration code block for AWS.

Once you have all the Terraform configuration code blocks, you can run `terraform init`, `plan`, and `apply` to check how Terraform is behaving with the defined configuration file. And finally, we manage to see that our *VPC* and *S3 bucket* got created in AWS, which can be confirmed by looking at the output values:

```
$ terraform apply -auto-approve
module.terraform-aws-vpc.aws_vpc.terraform-vpc: Creating...
aws_s3_bucket.s3_bucket: Creating...
module.terraform-aws-vpc.aws_vpc.terraform-vpc: Creation
complete after 16s [id=vpc-022e1e388557fcbbc]
module.terraform-aws-vpc.aws_subnet.terraform-subnet:
Creating...
module.terraform-aws-vpc.aws_subnet.terraform-subnet: Creation
complete after 4s [id=subnet-0d9dfb87b52abbb6e]
aws_s3_bucket.s3_bucket: Creation complete after 24s
[id=terraform-s3-bucket01]
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
Outputs:
bucket_domain_name = terraform-s3-bucket01.s3.amazonaws.com
s3_id = terraform-s3-bucket01
vpc_cidr_block = 10.0.0.0/16
vpc_id = vpc-022e1e388557fcbbc
```

The following are some of the key practices that you should follow when you are writing Terraform configuration files for AWS:

- Configure the Terraform backend and store the Terraform state file in our *AWS S3 bucket* so that it will remain secure as it contains some confidential information. For more on the encryption of S3 buckets, you can visit <https://docs.aws.amazon.com/AmazonS3/latest/userguide/bucket-encryption.html>.
- It is always good to use more and more modules rather than writing a resource code block in the configuration file. Consumption of modules will help to reduce the length of the code. In our example, we have used the *AWS VPC subnet module* that we wrote in *Chapter 7, Terraform Modules*.
- For the authentication to AWS, you can provide `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` values in the system environment variable or you can store them securely inside a CI/CD pipeline or in the *AWS Secrets Manager*. For more details, you can visit <https://aws.amazon.com/secrets-manager/>.
- Terraform's inbuilt function will help you to transform data into the required format. So, it's good to use the inbuilt function while drafting the Terraform configuration file for AWS.
- Use of locals in the Terraform configuration files should be encouraged if you need to use variables locally a number of times within the code.
- It is always recommended to run the `terraform validate` and `fmt` commands so that the `validate` command can detect syntax errors and `fmt` can help you to arrange the configuration code into canonical format and apply indentation.

From this section, you have got a fair amount of knowledge about drafting a Terraform configuration file for AWS. We also discussed what industry best practices can be followed while writing a configuration file for the AWS cloud.

Now, we will be using all our knowledge of Terraform configuration files to try to write Terraform configuration for the Azure cloud.

Writing Terraform configuration files for Azure

After writing a Terraform configuration file for *GCP* and *AWS*, we will use that knowledge and try to draft Terraform configuration files for Azure. You will notice that it also uses the same approach; the only difference comes in terms of defining the specific supported arguments. We explain this by considering an example of writing a configuration file for *Azure VM* with the *keyvault* module and an Azure storage account.

The following files are present in our GitHub repository; you can directly clone it from there:

```
inmishrar@terraform-lab-vm:~/HashiCorp-Infrastructure-Automation-Certification-Guide/chapter8/azure-files# tree
```

```
.
├── azure-files
│   ├── main.tf
│   ├── outputs.tf
│   ├── providers.tf
│   ├── terraform.tfvars
│   └── variables.tf
1 directory, 5 files
```

In the `main.tf` file, we have defined the following configuration code; you can copy the full code from our GitHub repository:

```
module "terraform-vm" {
  source          = "github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide.git//chapter7/azurerms/azurerms-virtual-machine-module?ref=v0.0.1"
  rgname          = var.rgname
  location        = var.location
  custom_tags     = var.custom_tags
  vm_size         = var.vm_size
  vm_name         = var.vm_name
  admin_username  = var.admin_username
  vm_publisher    = var.vm_publisher
  vm_offer        = var.vm_offer
  vm_sku          = var.vm_sku
  vm_version      = var.vm_version
  sku_name        = var.sku_name
  vnet_name       = var.vnet_name
  address_space   = var.address_space
  subnet_name     = var.subnet_name
  nic_name        = var.nic_name
  keyvault_name   = var.keyvault_name
}
```



```
keyvault_secret_name = var.keyvault_secret_name
}
...
```

We have followed the same approach that we used in the GCP and AWS sections, such as defining the input variable in a separate `variables.tf` file, the expected output in the `outputs.tf` file, the input variable values from the `terraform.tfvars` file, and the providers, backend, and Terraform version related information in the `providers.tf` file.

After executing `terraform apply`, we can see that Terraform manages to provision an *Azure VM*, *keyvault*, and *storage account* in Azure:

```
$ terraform apply -auto-approve
random_string.string_name: Refreshing state... [id=lgfkrwh]
module.terraform-vm.random_string.password: Refreshing state...
[id=Q7+ZaG9]t#-EFnQV]
module.terraform-vm.data.azurearm_client_config.current_config:
Refreshing state... [id=2021-01-06 11:01:46.2776914 +0000 UTC]
module.terraform-vm.azurearm_resource_group.rgname: Refreshing
state... [id=/subscriptions/933d9410-c867-4660-bf64-
07cc62c47deb/resourceGroups/Terraform-rg]
module.terraform-vm.azurearm_virtual_network.vnet: Refreshing
state... [id=/subscriptions/933d9410-c867-4660-bf64-
07cc62c47deb/resourceGroups/Terraform-rg/providers/Microsoft.
Network/virtualNetworks/Terraform-vnet]
module.terraform-vm.azurearm_key_vault.key_vault: Refreshing
state... [id=/subscriptions/933d9410-c867-4660-bf64-
07cc62c47deb/resourceGroups/Terraform-rg/providers/Microsoft.
KeyVault/vaults/terraform-vm-keyvault]
module.terraform-vm.azurearm_subnet.subnet: Refreshing state...
[id=/subscriptions/933d9410-c867-4660-bf64-07cc62c47deb/
resourceGroups/Terraform-rg/providers/Microsoft.Network/
virtualNetworks/Terraform-vnet/subnets/Terraform-subnet]
module.terraform-vm.azurearm_key_vault_secret.key_vault_
secret: Refreshing state... [id=https://terraform-vm-
keyvault.vault.azure.net/secrets/Terraform-vm-password/
f53f55be408f4863b8e67e3b7fb13dfb]
module.terraform-vm.azurearm_network_interface.nic: Refreshing
state... [id=/subscriptions/933d9410-c867-4660-bf64-
07cc62c47deb/resourceGroups/Terraform-rg/providers/Microsoft.
Network/networkInterfaces/Terraform-nic]
```

```
module.terraform-vm.azure_rm_Windows_virtual_machine.virtual_
machine: Refreshing state... [id=/subscriptions/933d9410-c867-
4660-bf64-07cc62c47deb/resourceGroups/Terraform-rg/providers/
Microsoft.Compute/virtualMachines/Terraform-vm]
azure_rm_storage_account.stg_account: Creating...
azure_rm_storage_account.stg_account: Creation complete after
35s [id=/subscriptions/933d9410-c867-4660-bf64-07cc62c47deb/
resourceGroups/Terraform-rg/providers/Microsoft.Storage/
storageAccounts/lgfkwrhazurestg]
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
Outputs:
stg_id = /subscriptions/933d9410-c867-4660-bf64-07cc62c47deb/
resourceGroups/Terraform-rg/providers/Microsoft.Storage/
storageAccounts/lgfkwrhazurestg
stg_name = lgfkwrhazurestg
vm_id = /subscriptions/933d9410-c867-4660-bf64-07cc62c47deb/
resourceGroups/Terraform-rg/providers/Microsoft.Compute/
virtualMachines/Terraform-vm
vm_name = Terraform-vm
vm_private_ip = 10.1.1.4
```

The following are some of the key best practices that we can consider while writing a Terraform configuration for Azure:

- You should define the Terraform backend and store the Terraform state file into *Azure Blob Storage* so that the state file remains secure. For further information about the encryption method of Blob storage, you can read <https://docs.microsoft.com/en-us/azure/storage/common/storage-service-encryption>.
- Create separate `.tf` files so that it will be easy for you to manage and make amendments to the files.
- We encourage users to consume modules while writing the Terraform configuration file for Azure. In our case, we used the Azure VM module that we wrote in *Chapter 7, Terraform Modules*. Likewise, we can write modules and consume them while writing Terraform configuration files.

- For the authentication, we can use several methods. The best would be defining `Client_id`, `Client_secret`, `Tenant_id`, and `Subscription_id` in the system environment variables or they can be defined in CI/CD pipeline variables so that they will remain secure. For further information about different authentication methods, you can read the *Integrating with Azure* section in *Chapter 5, Terraform CLI*.
- For troubleshooting, it is good to see different syntax errors and for that, it's better to run the `terraform validate` command, which will show you whether there is any sort of syntax error in the Terraform configuration files.
- You should use the `terraform fmt` command so that the content inside the configuration file becomes idiomatic or canonically arranged; that will make configuration files readable.
- You can define the expected output in the `outputs.tf` file so that you can check which resources/services in Azure were created or updated.
- It's always good to use the Terraform-provided inbuilt function so that data can be formatted to the desired one, so that it can be consumed.

From this section of the chapter, you have got an understanding of Terraform configuration files for Azure. We also discussed the best practices that we can follow while writing Terraform configuration files for Azure.

Summary

From this chapter, you have received an understanding of the Terraform configuration files, how they are different from the JSON syntax, and the different data types supported by the Terraform language file and the JSON file. Moving further, we also discussed how we can draft Terraform configuration for major cloud providers of GCP, AWS, and Azure. We also learned about some of the best practices that can be followed while writing Terraform configuration files.

In the next chapter, we will discuss Terraform stacks, where we will learn what a Terraform stack is and how we can create a Terraform stack for the different cloud platforms of GCP, AWS, and Azure. This will help you to provision a very large enterprise infrastructure.

Questions

The answers to the following questions can be found in the *Assessments* section at the end of this book:

1. How many spaces of indentation are recommended for each nesting level in the Terraform configuration file?
 - A. One
 - B. Two
 - C. Three
 - D. Four
2. With which of the following ways can you place comments in the Terraform configuration file?
 - A. #
 - B. !
 - C. </
 - D. />
3. Which command can be dedicatedly used to perform a syntax check of the Terraform configuration file?
 - A. `terraform validate`
 - B. `terraform syntax`
 - C. `terraform fmt`
 - D. `terraform plan`
4. Suppose you have written a Terraform configuration file for Azure that will provision a storage account in Azure. How can you validate the name of the storage account that was provisioned?
 - A. Define output values in the `outputs.tf` file.
 - B. Manually go inside the state file and look for those specific values.
 - C. Run the `terraform show` command.
 - D. Run the `terraform apply` command.

5. You have written a Terraform configuration file where you need to take variable values as input. Which of the following would be your preference?
- A. `terraform.tf`
 - B. `terraform.txt`
 - C. `terraform.tfvars`
 - D. `terraform.vars`

Further reading

You can check out the following links for more information about the topics that were covered in this chapter:

- Terraform configuration files: <https://www.terraform.io/docs/configuration/syntax/index.html>
- Terraform JSON syntax: <https://www.terraform.io/docs/configuration/syntax-json.html>
- Terraform data types: <https://www.terraform.io/docs/configuration/expressions/types.html>
- Terraform expressions: <https://www.terraform.io/docs/configuration/expressions/index.html>

9

Understanding Terraform Stacks

In the previous chapter, we started our journey by understanding the Terraform configuration file, and we explored how Terraform language files, which are human-readable, differ from JSON files, which are machine-readable. Moving further, we saw the different data types supported by both JSON and Terraform files. We also discussed industry best practices for writing Terraform configuration files with major cloud providers, such as **Google Cloud Platform (GCP)**, Azure, and **Amazon Web Services (AWS)**.

In this chapter, we will discuss how we can handle a large enterprise infrastructure deployment, upgrading, and so on using Terraform configuration code. We will be discussing infrastructure deployment to GCP, Azure, and AWS using a Terraform stack. In this chapter, you will gain a thorough understanding of Terraform stacks and modules and how stacks can be used effectively for infrastructure deployment and updates.

The following topics will be covered in this chapter:

- Understanding Terraform stacks
- Writing a Terraform stack for GCP
- Writing a Terraform stack for AWS
- Writing a Terraform stack for Azure

Technical requirements

To follow along with this chapter, you need to have an understanding of the Terraform CLI and its workflow. You need to have a good command of writing Terraform modules and other Terraform configuration files. You can find all the code used in this chapter at <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter9>.

Check out the following link to see the Code in Action video:

<https://bit.ly/3yCOUHG>

Understanding Terraform stacks

Suppose you are working with one of your colleagues, John. You and John have been assigned to deploy 50 virtual machines, 20 virtual networks, 10 web apps, and 5 function apps in Azure. You have many questions about this infrastructure deployment in Azure, such as how you will provision this whole infrastructure, what will be the easiest way to perform the deployment, and how you would you scale this infrastructure up or down if, in the near future, the addition or deletion of resources needs to be performed. In response to these questions, John says that you should be able to use Terraform configuration files for the deployment and management of the complete infrastructure. He also suggests writing Terraform modules for each resource and getting them published to GitHub or Terraform Registry; then, later on, stacks of the infrastructure can be prepared by referencing the modules. Preparing these stacks for infrastructure provisioning and management would be the best way to go, John says. Terraform stacks are the result of one or more modules being combined with environment-specific input parameter values. We already know that Terraform modules are written in such a way that they can be consumed again and again. In the same way, we can bring modules together and form a stack that will help us manage large enterprise infrastructures, such as the one that you and John are building:

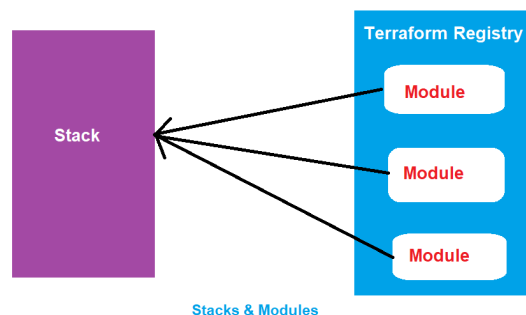


Figure 9.1 – Terraform stacks

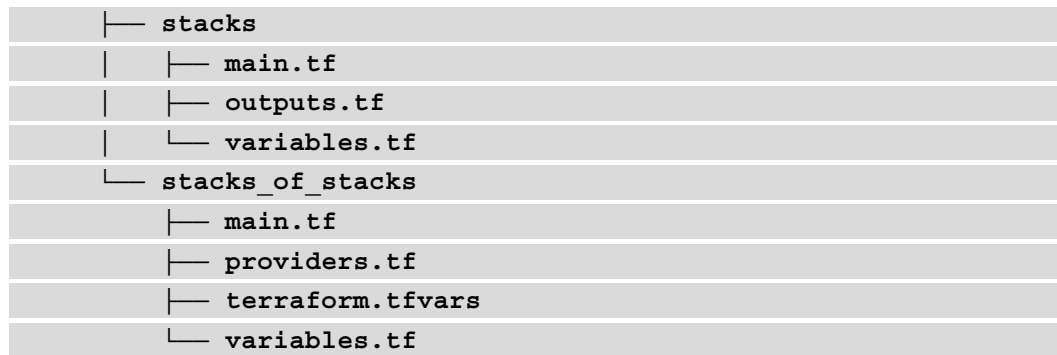
That should have given you a basic understanding of what exactly Terraform stack is; you will get an even better understanding of it when we discuss them in relation to GCP, AWS, and Azure.

Writing Terraform stacks for GCP

To get a better understanding of Terraform stacks for GCP, let's try to write some GCP modules and prepare a stack using them. We will prepare some modules, names `vpc`, `subnet`, `route`, and `storage`, and using them, we will show you how you can prepare a stack. Along with it, we will try to discuss the code that has been written while drafting these modules. You can find all our code at our GitHub repository: <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter9/gcp/modules>. You can see the following files present inside the `gcp` directory of our GitHub:

```
inmishrar@terraform-vm:~/HashiCorp-Infrastructure-Automation-Certification-Guide/chapter9# tree
```

```
.
├── gcp
│   ├── modules
│   │   ├── route
│   │   │   ├── main.tf
│   │   │   ├── outputs.tf
│   │   │   └── variables.tf
│   │   ├── storage
│   │   │   ├── main.tf
│   │   │   ├── outputs.tf
│   │   │   └── variables.tf
│   │   ├── subnet
│   │   │   ├── main.tf
│   │   │   ├── outputs.tf
│   │   │   └── variables.tf
│   │   └── vpc
│   │       ├── main.tf
│   │       ├── outputs.tf
│   │       └── variables.tf
```

Let's discuss the code that we had defined while creating the `vpc` module; in `main.tf`, the following code is present:

```
resource "google_compute_network" "vpc" {
  name                = var.vpc_name
  mtu                 = var.vpc_mtu
  description         = var.vpc_description
  routing_mode       = var.vpc_routing_mode
  project             = var.project_id
  delete_default_routes_on_create = var.delete_default_routes_on_create
  auto_create_subnetworks = var.auto_create_subnetworks
}
```

To draft the `vpc` module, we need to provide some of the arguments from Terraform Registry: https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/compute_network. We also need to provide their respective variables, declared in the `variables.tf` file, which includes the following code:

```
variable "project_id" {
  type = string
  description = "The ID of the project where this VPC will be created"
}

variable "vpc_name" {
  type = string
```

```
    description = "The name of the network being created"
  }
  variable "vpc_routing_mode" {
    type      = string
    default   = "GLOBAL"
    description = "The network routing mode (default 'GLOBAL')"
  }
  variable "vpc_description" {
    type      = string
    description = "An optional description of this resource.
The resource must be recreated to modify this field."
    default   = ""
  }
  variable "auto_create_subnetworks" {
    type      = bool
    description = "When set to true, the network is created in
'auto subnet mode' and it will create a subnet for each region
automatically across the 10.128.0.0/9 address range. When set
to false, the network is created in 'custom subnet mode' so
the user can explicitly connect subnetwork resources."
    default   = false
  }
  variable "delete_default_routes_on_create" {
    type      = bool
    description = "If set, ensure that all routes within
the network specified whose names begin with 'default-route'
and with a next hop of 'default-internet-gateway' are deleted"
    default   = false
  }
  variable "vpc_mtu" {
    type      = number
    description = "The network MTU. Must be a value between 1460
and 1500 inclusive. If set to 0 (meaning MTU is unset), the
network will default to 1460 automatically."
    default   = 0
  }
}
```

We defined all the argument values as input variables so that our code would be reusable, and we should easily be able to pass the respective input variable values during runtime for any file ending with `.tfvars`.

In the `outputs.tf` file, we kept following code block to help us to export any output during the Terraform execution:

```
output "vpc_id" {
  value = google_compute_network.vpc.id
}
output "vpc_self_link" {
  value = google_compute_network.vpc.self_link
}
output "vpc_name" {
  value = google_compute_network.vpc.name
}
```

You can refer to the Terraform documentation of `vpc` at https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/compute_network to get an idea of the attributes that can be exported from the `vpc` resource.

If you look at the `vpc` module code, you'll see we kept it short and sweet, without adding any complexity to it. So, we believe you should be able to write the `vpc` module on your own.

Moving on, let's try to understand the code block present inside the `subnet` module. In the `main.tf` file, the following code is present:

```
locals {
  subnets = {
    for x in var.subnets :
      "${x.subnet_region}/${x.subnet_name}" => x
  }
}
/*****
Subnet Code
*****/
resource "google_compute_subnetwork" "subnet" {
  for_each = local.subnets
```

```

name                = each.value.subnet_name
ip_cidr_range       = each.value.subnet_ip
region              = each.value.subnet_region
private_ip_google_access = lookup(each.value, "subnet_
private_access", "false")
dynamic "log_config" {
  for_each = lookup(each.value, "subnet_flow_
logs", false) ? [{
    aggregation_interval = lookup(each.value, "subnet_flow_
logs_interval", "INTERVAL_5_SEC")
    flow_sampling         = lookup(each.value, "subnet_flow_
logs_sampling", "0.5")
    metadata              = lookup(each.value, "subnet_flow_
logs_metadata", "INCLUDE_ALL_METADATA")
  }] : []
  content {
    aggregation_interval = log_config.value.aggregation_
interval
    flow_sampling         = log_config.value.flow_sampling
    metadata              = log_config.value.metadata
  }
}
network             = var.vpc_name
project             = var.project_id
description         = lookup(each.value, "description", null)
secondary_ip_range = [
  for i in range(
    length(
      contains(
        keys(var.secondary_ranges), each.value.subnet_
name) == true
      ? var.secondary_ranges[each.value.subnet_name]
      : []
    )) :
    var.secondary_ranges[each.value.subnet_name][i]
  ]
}

```

Do you find the `subnet` module to be a little bit complex? We do too! You can see that in that code block, we used some of the key items of Terraform stacks, such as these:

- `locals`
- `for` loops
- `for_each` loops
- lookup functions
- length functions
- `contains` functions
- Dynamic iteration

We discussed all these items in our previous chapters, so we will not discuss them here. We encourage you to go back and read *Chapter 4, Deep Dive into Terraform*, to understand their use.

You might be wondering why we have written such complex code for the `subnet` module. The benefit of having this code is as follows: suppose you wish to deploy hundreds of subnets within a VPC. You could easily do so by just providing the respective subnet argument input values for Terraform to read from any file ending with `.tfvars` or `.tfvars.json`. To provide the flexibility of defining arguments and iterations in our code, we have used dynamic iteration and `for_each` and `for` loops.

Similarly, we wrote code blocks for `route` and `storage` modules. You can refer to and get them directly from our GitHub repository, so we will not explain them here.

Now the question is, how can we consume these modules? We have created a directory named `stacks` where we have included all the necessary code. In the `main.tf` file, you can see the following:

```
module "vpc" {
  source = "github.com/
  PacktPublishing/HashiCorp-Infrastructure-Automation-
  Certification-Guide.git//chapter9/gcp/modules/vpc?ref=v1.12"
  vpc_name = var.vpc_name
  vpc_mtu = var.vpc_mtu
  vpc_description = var.vpc_description
  vpc_routing_mode = var.vpc_routing_mode
  project_id = var.project_id
  delete_default_routes_on_create = var.delete_default_routes_
  on_create
```

```
    auto_create_subnetworks = var.auto_create_subnetworks
  }
  module "subnet" {
    source = "github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git//chapter9/
gcp/modules/subnet?ref=v1.13"
    project_id = var.project_id
    vpc_name   = var.vpc_name
    subnets  = var.subnets
    secondary_ranges = var.secondary_ranges
    depends_on = [module.vpc.id]
  }
  module "routes" {
    source = "github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git//chapter9/
gcp/modules/route?ref=v1.10"
    project_id = var.project_id
    vpc_name   = var.vpc_name
    routes     = var.routes
    depends_on = [module.vpc.id]
  }
  module "storage" {
    source = "github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git//chapter9/
gcp/modules/storage?ref=v1.11"
    stg_name   = var.stg_name
    location   = var.location
    force_destroy = var.force_destroy
    storage_class = var.storage_class
    project_id = var.project_id
    labels     = var.labels
  }
}
```

By now, you should understand what stacks are: put simply, they are collections of modules defined in a single `.tf` file or multiple `.tf` files inside the same directory.

While calling a module, if the input variables values in the resource or module code block is variable name then that variable name should be defined as an input argument in the new module code block. Now this input argument in the module code block can take either value or again we can define variable. For example, in the preceding code block, we wrote the `storage` module. You can see `stg_name` on the left side of the `=` sign, which is acting as an argument, and on the right side, we define it again as a variable, `var.stg_name`. Now, `var.stg_name` can be passed either from the CLI or from any file ending with `.tfvars` or `.auto.tfvars`.

The benefits of creating stacks are that you get better flexibility in defining code blocks, it reduces the length and complexity of the Terraform configuration code, and it makes modules reusable.

Now in the `gcp` directory, you can see that there is a directory with the name `stacks_of_stacks`. You must be wondering why we created that directory and what exactly we defined inside it. In the `main.tf` file, we have the following code:

```
module "gcp_stacks" {
  source           = "../stacks"
  zone            = var.zone
  region         = var.region
  project_name    = var.project_name
  vpc_name       = var.vpc_name
  vpc_mtu        = var.vpc_mtu
  vpc_description = var.vpc_description
  vpc_routing_mode = var.vpc_routing_mode
  project_id     = var.project_id
  delete_default_routes_on_create = var.delete_default_routes_on_create
  auto_create_subnetworks = var.auto_create_subnetworks
  subnets       = var.subnets
  secondary_ranges = var.secondary_ranges
  routes        = var.routes
  stg_name      = var.stg_name
  location     = var.location
  force_destroy = var.force_destroy
  storage_class = var.storage_class
  labels       = var.labels
}
```

In the preceding code block, you can see that module "gcp_stacks" been referenced as a module only. Here, we defined all the arguments that you can see on the left side of the = sign, which are basically declared variables in our earlier-defined `variables.tf` file, present inside the `stacks` folder. While creating stacks of stacks, we need to provide `source = "<Local Path>";` that is why we defined `source = "../stacks"`: so that the module will be able to take all the code present inside the `stacks` directory. This stack of stacks will help to reduce the declaration of the common variables and the code complexity.

We are all set now. Let's see what happens when we execute `terraform init`, `plan`, and `apply`. The following is the code snippet we get when we run the `terraform init` command:

```
$ terraform init
Initializing modules...
- gcp_stacks in ../stacks
Downloading github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git?ref=v1.10 for
gcp_stacks.routes...
- gcp_stacks.routes in .terraform\modules\gcp_stacks.routes\
chapter9\gcp\modules\route
Downloading github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git?ref=v1.11 for
gcp_stacks.storage...
- gcp_stacks.storage in .terraform\modules\gcp_stacks.storage\
chapter9\gcp\modules\storage
Downloading github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git?ref=v1.13 for
gcp_stacks.subnet...
- gcp_stacks.subnet in .terraform\modules\gcp_stacks.subnet\
chapter9\gcp\modules\subnet
Downloading github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git?ref=v1.12 for
gcp_stacks.vpc...
- gcp_stacks.vpc in .terraform\modules\gcp_stacks.vpc\chapter9\
gcp\modules\vpc
Initializing the backend...
Initializing provider plugins...
- Finding hashicorp/google versions matching "~> 3.0"...
```



```
- Installing hashicorp/google v3.53.0...
- Installed hashicorp/google v3.53.0 (signed by HashiCorp)
Terraform has been successfully initialized!
```

You may now begin working with Terraform. Try running `terraform plan` to see any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.

After running `terraform init`, we can see that it downloads all the modules from our GitHub repository. If we go ahead and run `terraform plan` and use `apply`, it will provision `vpc`, `subnet`, `route`, and `storage` in GCP. The following is the code snippet we get when we run `terraform apply`:

```
$ terraform apply
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions: #module.
gcp_stacks.module.routes.google_compute_route.route["egress-
internet"] will be created

+ resource "google_compute_route" "route" {
  + description      = "route through IGW to access
internet"
  + dest_range       = "0.0.0.0/0"
  + id               = (known after apply)
  . . .
}
. . .
#
Plan: 6 to add, 0 to change, 0 to destroy.
Do you want to perform these actions?
Terraform will perform the actions described above. Only 'yes'
will be accepted to approve.
```

We have written Terraform modules for GCP, and we have brought multiple modules together to form stacks. We also learned how we can prepare stacks of stacks. With this, we can easily write Terraform modules and consume them as and when required for our GCP infrastructure.

Will there be any difference in writing Terraform stacks for AWS? In our next section, we will write Terraform modules and prepare stacks with those modules for AWS.

Writing Terraform stacks for AWS

We have learned how to develop Terraform modules for GCP. We will use that learning and try to write some AWS modules and combine those modules to prepare stacks. For better understanding, it's best to write some simple modules for AWS. We have written simple modules for `vpc`, `subnet`, and `s3_bucket`. You can find all the code in our GitHub repository, inside the `aws` directory of `chapter9`. Here is the directory structure:

```

.
├── aws
│   ├── modules
│   │   ├── s3
│   │   │   ├── main.tf
│   │   │   ├── outputs.tf
│   │   │   └── variables.tf
│   │   ├── vpc-subnet
│   │   │   ├── main.tf
│   │   │   ├── outputs.tf
│   │   │   └── variables.tf
│   ├── stacks
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   └── variables.tf
│   ├── stacks_of_stacks
│   │   ├── main.tf
│   │   ├── providers.tf
│   ├── terraform.tfvars
│   └── variables.tf

```

We are not discussing all the code, just quickly demonstrating what we have written for preparing stacks.

In the `stacks` directory, there is a `main.tf` file that contains the following code:

```
module "vpc" {
  source = "github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide.git//chapter9/aws/modules/vpc-subnet?ref=v1.14"
  cidr_block = var.cidr_block
  instance_tenancy = var.instance_tenancy
  enable_dns_hostnames = var.enable_dns_hostnames
  enable_dns_support = var.enable_dns_support
  enable_classiclink = var.enable_classiclink
  enable_classiclink_dns_support = var.enable_classiclink_dns_support
  assign_generated_ipv6_cidr_block = var.assign_generated_ipv6_cidr_block
  vpc_name = var.vpc_name
  custom_tags = var.custom_tags
  subnet_cidr = var.subnet_cidr
  subnet_name = var.subnet_name
}

module "s3" {
  source = "github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide.git//chapter9/aws/modules/s3?ref=v1.14"
  create_bucket = var.create_bucket
  bucket_name = var.bucket_name
  bucket_acl = var.bucket_acl
  force_destroy = var.force_destroy
  acceleration_status = var.acceleration_status
  custom_tags = var.custom_tags
  depends_on = [module.vpc.id]
}
```

The preceding code is the normal way of defining stacks. What if we want to reduce the length of the code? Well, we can achieve that by building stacks of stacks. For this, there is the following code in the `main.tf` file of the `stacks_of_stacks` directory:

```
module "aws_stacks" {
  source                = "../stacks"
  cidr_block            = var.cidr_block
  instance_tenancy     = var.instance_tenancy
  enable_dns_hostnames = var.enable_dns_hostnames
  enable_dns_support   = var.enable_dns_support
  enable_classiclink   = var.enable_classiclink
  enable_classiclink_dns_support = var.enable_classiclink_dns_support
  assign_generated_ipv6_cidr_block = var.assign_generated_ipv6_cidr_block
  vpc_name             = var.vpc_name
  custom_tags          = var.custom_tags
  subnet_name         = var.subnet_name
  subnet_cidr         = var.subnet_cidr
  create_bucket       = var.create_bucket
  bucket_name         = var.bucket_name
  bucket_acl          = var.bucket_acl
  force_destroy       = var.force_destroy
  acceleration_status = var.acceleration_status
}
```

When we prepare stacks of stacks, we need to provide the source path of the *local directory*. This method of preparing stacks of stacks is very beneficial when we need to deploy many environments, such as dev, test, and prod environments, that have the same infrastructure that is already defined in the stacks.

We now run `terraform init`. That will open and read all the `.tf` files containing defined modules and start downloading the modules and placing them in the `.terraform` directory. As you can see here, after running `terraform init`, Terraform initializes the modules and downloads them:

```
$ terraform init
Initializing modules...
- aws_stacks in ../stacks
```

```
Downloading github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git?ref=v1.14 for
aws_stacks.s3...
- aws_stacks.s3 in .terraform\modules\aws_stacks.s3\chapter9\
aws\modules\s3
Downloading github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git?ref=v1.14 for
aws_stacks.vpc...
- aws_stacks.vpc in .terraform\modules\aws_stacks.vpc\chapter9\
aws\modules\vpc-subnet
Initializing the backend...
Initializing provider plugins...
- Using previously-installed hashicorp/aws v3.25.0
Terraform has been successfully initialized!
You may now begin working with Terraform. Try running
"terraform plan" to see any changes that are required for your
infrastructure. All Terraform commands should now work.
If you ever set or change modules or backend configuration
for Terraform, rerun this command to reinitialize your working
directory. If you forget, other commands will detect it and
remind you to do so if necessary.
```

You can run `terraform plan` after `terraform init`, which will show you all the resources that it is going to create or update. If you wish to get infrastructure deployed, then you can run `terraform apply -auto-approve`. Terraform will deploy or update infrastructure. In our case, as you can see, Terraform created `vpc` with `subnet` and `s3_bucket` in AWS:

```
$ terraform apply -auto-approve
module.aws_stacks.module.vpc.aws_vpc.vpc: Creating...
module.aws_stacks.module.s3.aws_s3_bucket.s3_bucket[0]:
Creating...
module.aws_stacks.module.vpc.aws_vpc.vpc: Creation complete
after 16s [id=vpc-002c1a898b6caa9dc]
module.aws_stacks.module.vpc.aws_subnet.subnet: Creating...
module.aws_stacks.module.vpc.aws_subnet.subnet: Creation
complete after 4s [id=subnet-0b87c63201112d0b1]
module.aws_stacks.module.s3.aws_s3_bucket.s3_bucket[0]:
Creation complete after 29s [id=tf-s3-bucket32342]
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

One of the main challenges when preparing stacks of stacks is that you may not be able to define or validate the specific output from a stack because it will consist of many different modules combined and defined as an argument.

We learned about creating modules for AWS and combining those modules to prepare stacks and stacks of stacks. So, now it should be easy for you to write a module and prepare stacks for AWS. Let's use this learning and try to write modules and prepare stacks for Azure.

Writing Terraform stacks for Azure

Earlier, we discussed writing Terraform modules and preparing stacks using those modules for AWS and GCP. Now the question is, will there be any difference in writing modules and preparing stacks for Azure? In answer to that question, no, there is no major difference: if you followed and understood the earlier processes of creating stacks and modules for AWS and GCP, you can use the same knowledge to prepare stacks and modules. Let's see how we can prepare modules for Azure Storage and Azure App Service. We have placed all our code into our GitHub repository in the following directory:

```
.
├── azure
│   ├── modules
│   │   ├── storage
│   │   │   ├── main.tf
│   │   │   ├── outputs.tf
│   │   │   └── variables.tf
│   │   ├── webapp
│   │   │   ├── main.tf
│   │   │   ├── outputs.tf
│   │   │   └── variables.tf
│   ├── stacks
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   └── variables.tf
│   └── stacks_of_stacks
│       ├── main.tf
│       ├── providers.tf
│       ├── terraform.tfvars
│       └── variables.tf
```

We will not be discussing all the code; you can refer to our GitHub and copy the code directly from there to gain a full understanding of it.

After writing the module code for `storage` and `webapp`, we combined both of the prepared modules and made a stack. Furthermore, to reduce the code length and instead of declaring variables multiple times, we prepared stacks of stacks and kept all the code in the `stacks_of_stacks` directory.

The following code is present in the `main.tf` file of the `stacks_of_stacks` directory:

```
module "azure_stacks" {
  source                = "../stacks"
  create_resource_group = var.create_resource_group
  resource_group_name   = var.resource_group_name
  location              = var.location
  tags                  = var.tags
  app_config            = var.app_config
  ip_address            = var.ip_address
  app_settings          = var.app_settings
  connection_string     = var.connection_string
  asp_config            = var.asp_config
  storage_account_name  = var.storage_account_name
  account_kind          = var.account_kind
  skuname               = var.skuname
  allow_blob_public_access = var.allow_blob_public_access
  soft_delete_retention  = var.soft_delete_retention
  containers_list       = var.containers_list
}
```

If you observe the `main.tf` file present in the `stacks` directory, you'll see that we defined many input variables and declared all the variables in the `variables.tf` file. The declared variables will act as arguments, which will be defined on the left-hand side of the `=` sign in the `main.tf` file of `stacks_of_stacks`.

We passed all the required input variables using the `terraform.tfvars` file, which is also present inside the `stacks_of_stacks` directory.

When we run `terraform init`, Terraform downloads the `storage` and `webapp` modules from our GitHub repository and places them inside the `.terraform` folder, which you can see in the following code snippet:

```
$ terraform init
Initializing modules...
Downloading github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git?ref=v1.15 for
azure_stacks.storage...
- azure_stacks.storage in .terraform\modules\azure_stacks.
storage\chapter9\azure\modules\storage
Downloading github.com/PacktPublishing/HashiCorp-
Infrastructure-Automation-Certification-Guide.git?ref=v1.15 for
azure_stacks.webapp...
- azure_stacks.webapp in .terraform\modules\azure_stacks.
webapp\chapter9\azure\modules\webapp
Initializing the backend...
Initializing provider plugins...
- Using previously-installed hashicorp/azurerm v2.56.0
- Using previously-installed hashicorp/random v3.0.1
The following providers do not have any version constraints in
configuration, so the latest version was installed.
To prevent automatic upgrades to new major versions
that may contain breaking changes, we recommend adding
version constraints in a required_providers block in your
configuration, with the constraint strings suggested below.
* hashicorp/random: version = "~> 3.0.1"
Terraform has been successfully initialized!
You may now begin working with Terraform. Try running
"terraform plan" to see any changes that are required for your
infrastructure. All Terraform commands should now work.
If you ever set or change modules or backend configuration
for Terraform, rerun this command to reinitialize your working
directory. If you forget, other commands will detect it and
remind you to do so if necessary.
```


In order to provision webapp and storage in Azure, we can run `terraform apply -auto-approve` and see that webapp and storage are provisioned in Azure, as shown in the following figure:

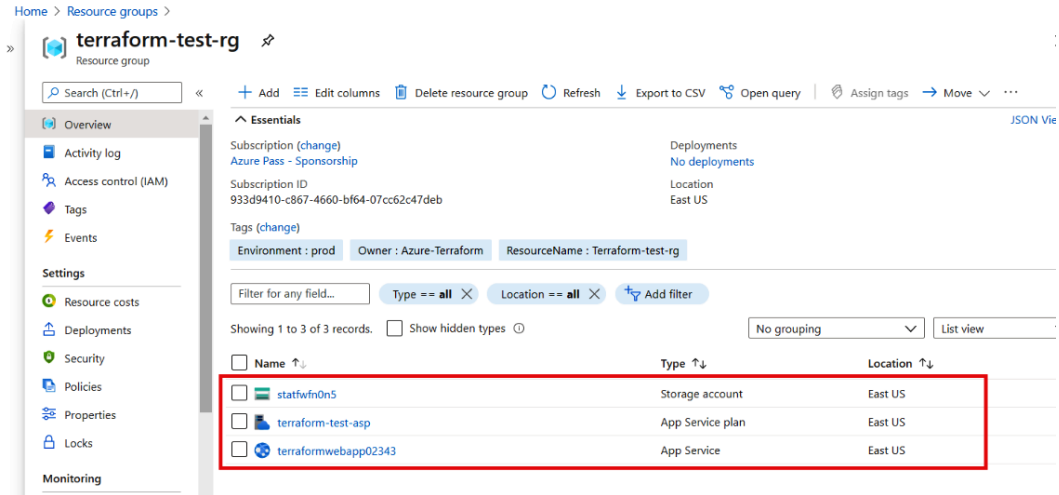


Figure 9.2 – Azure resources

From this section of the chapter, you should have gained an understanding of how to write Terraform modules and prepare stacks from those modules in Azure. You should have understood how large and repeated enterprise infrastructure in Azure can easily be provisioned or updated using Terraform stacks.

Summary

In this chapter, we discussed Terraform stacks. We learned about what Terraform stacks are and how we can create Terraform stacks for different clouds, such as GCP, AWS, and Azure. This should help you to provision or update a very large enterprise infrastructure using Terraform IaC. In our next chapter, we are going to discuss Terraform Cloud and the enterprise version of Terraform. We will look at how Terraform Cloud and the enterprise product help enterprise customers and we will discover the benefits that an enterprise customer can get from them.

Questions

1. Which of the following is the best option for managing a large enterprise infrastructure environment using Terraform?
 - A. Data sources
 - B. Stacks and modules
 - C. Resources
 - D. Provisioners
2. Suppose you have created a Terraform stack and you can see that the indentation of the Terraform code is not in the correct format. Which of the following Terraform commands would you run?
 - A. `terraform sources`
 - B. `terraform init`
 - C. `terraform fmt -recursive`
 - D. `terraform plan`
3. Which command needs to be executed to download Terraform modules from the Github source?
 - A. `terraform init`
 - B. `terraform syntax`
 - C. `terraform fmt`
 - D. `terraform plan`
4. You created a Terraform stack containing the following code

```
module "azure_stacks" {  
  source           = "../stacks"  
  resource_group_name = var.resource_group_name  
  ...
```

You executed `terraform plan` and see that it is prompting you to provide the following values:

```
$ terraform plan  
var.resource_group_name  
  A container that holds related resources for an Azure  
  solution  
Enter a value:
```

What could you have done to prevent it from prompting during runtime?

- A. Defined `resource_group_name` variable value into any file ending with `.tfvars`, `.auto.tfvars` or `.tfvars.json`
 - B. Manually provided `resource_group_name` value during runtime
 - C. Hardcoded `resource_group_name = "Terraform-test-rg"`
5. D. Run the `terraform validate` command
6. Suppose you're consuming the module and forgot to mention the version in the module code block. What could be a major problem with not defining the version in the module code?
- A. Terraform will always reference the latest version of the modules.
 - B. Terraform code may fail when we run `plan` or `apply` if previously defined module code block is not compatible with the latest version.
 - C. Terraform will upgrade the infrastructure as per the latest version of the modules.
 - D. Terraform will run `terraform apply` automatically.

Further reading

You can check out the following links for more information about the topics that were covered in this chapter:

- AWS Modules: <https://github.com/terraform-aws-modules>
- Azure Modules: <https://github.com/terraform-azurerm-modules>
- Google Modules: <https://github.com/terraform-google-modules>

10

Terraform Cloud and Terraform Enterprise

In our previous chapter, we discussed how we can handle a large-enterprise infrastructure deployment and its updates—and other tasks—using Terraform configuration code. We saw how we can effectively write Terraform modules for major cloud providers such as **Google Cloud Platform (GCP)**, **Azure**, and **Amazon Web Services (AWS)**. Later on in the chapter, we saw how we can club multiple modules together to prepare a composite module also known as a stack. Using that stack, we managed to provision a large-enterprise infrastructure.

In this chapter, we are going to introduce different versions of the Terraform product, such as Terraform Cloud and Terraform Enterprise. We will discuss Terraform Sentinel, which allows you to implement policies as code, and will further see which features are present in Terraform Cloud and Terraform Enterprise as compared to the **Terraform command-line interface (Terraform CLI)**, to give us an insight into using Terraform Cloud and Terraform Enterprise.

The following topics will be covered in this chapter:

- Introducing Terraform Cloud
- Understanding Terraform Enterprise
- Overviewing Terraform Sentinel
- Comparing different Terraform features

Technical requirements

To follow this chapter, you need to have an understanding of Terraform CLI and its workflow, as well as a basic grasp of how to write a Terraform configuration file. You can find all the code used in this chapter at the following GitHub link: <https://github.com/PacktPublishing/HashiCorp-Infrastructure-Automation-Certification-Guide/tree/master/chapter10>.

Check out the following link to see the Code in Action video:

<https://bit.ly/3qYE8aq>

Introducing Terraform Cloud

Terraform Cloud is one of Terraform's **software-as-a-service (SaaS)** offerings. It is a hosted service that can be accessed from <https://app.terraform.io/>. Terraform Cloud provides you with the flexibility to run your Terraform configuration code remotely, but not only this: it also allows you to run configuration code locally and get this stored in the source control repository. If you need to work closely with many colleagues within your team or across other teams, then Terraform Cloud provides great collaboration features. It can hold Terraform state files and secret data safely and securely, allowing you to access these easily. It also provides a private space where you can publish all your modules and consume them as per your requirements, giving you an opportunity to write a policy for governing the code sitting inside the Terraform configuration file. As you may have noticed when using open source Terraform CLI, it uses a persistent working directory that may introduce a single point of failure, whereas when you start using Terraform Cloud, in order to execute Terraform configuration files, Terraform provisions **virtual machines (VMs)** in their cloud infrastructure and then destroys them once execution is completed. Completing execution in the backend is also called **remote Terraform execution** or a **remote operation**.

Terraform Cloud has defined **workspaces** instead of any persistent directories. A Terraform Cloud workspace contains everything that it requires for provisioning the infrastructure. The state file in Terraform Cloud gets stored in the remote storage, which has a direct connection to the workspace so that Terraform will be able to access that state file while executing its workflow. Terraform Cloud helps you to exchange the output of one workspace to another, using `terraform_remote_state` data sources. We don't need to have any additional authentication when we want to use the output of one workspace in another workspace. If there is some sort of dependency from one workspace to another, then we can define run triggers just to ensure that when the upstream workspace receives changes, it then goes and makes the necessary updates in the downstream workspace. To read more about the `run trigger` command, you can visit <https://www.terraform.io/docs/cloud/workspaces/run-triggers.html>.

As with any other code, **infrastructure as code (IaC)**—that is, Terraform configuration code—can also be placed inside version control tools, and Terraform Cloud supports integration with these tools. Integrating Terraform Cloud with version control tools is totally optional; it's not mandatory. If you are planning to use version control tools with Terraform Cloud, then you need to link each workspace of Terraform Cloud to the version control repository, where you can even specify the branch or subdirectory of your version control repository. The Terraform Cloud workspace will keep an eye on the linked version control repository, and if it sees any commit to the version control repository, then the Terraform Cloud workspace will execute Terraform plans by taking the latest code. Even if it observes any new pull request to the version control, then it will also execute `terraform plan` with the latest code, and reviewers can easily see the expected change from the Terraform Cloud workspace and approve the pull request accordingly.

Terraform Cloud supports the following version control repositories:

- GitHub.com
- GitHub.com (OAuth)
- GitHub Enterprise
- GitLab.com
- GitLab **Enterprise Edition (EE)** and **Community Edition (CE)**
- Bitbucket Cloud
- Bitbucket Server
- Azure DevOps Server
- Azure DevOps Services

We will not go into a detailed discussion of all the version control integrations with Terraform Cloud, but you can read about this directly at <https://www.terraform.io/docs/cloud/vcs/index.html#supported-vcs-providers>.

What about if you are using unsupported version control tools or you want to preserve the existing deployment and validation pipeline? In that case, it's better to use the **application programming interface (API)** or Terraform CLI to upload the latest version of the configuration.

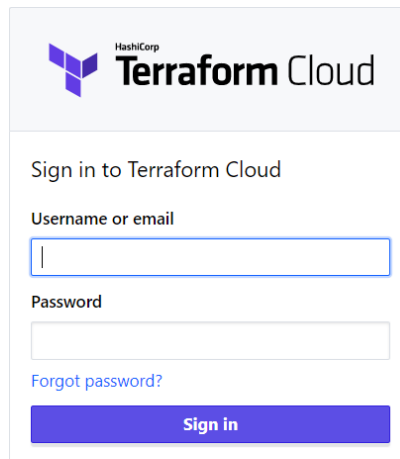
Terraform Cloud offers both free and paid plans. Free plans are good for individuals or small teams. The free version of Terraform Cloud includes remote state storage, **version control system (VCS)** integration, and remote runs. The paid version of Terraform Cloud is suitable for a larger team—that is, mainly for medium-sized businesses.

Terraform Cloud workflow

Let's discuss how you can start exploring Terraform Cloud, as follows:

1. Firstly, we need to sign up or log in by visiting <https://app.terraform.io/>, as illustrated in the following screenshot:

<https://app.terraform.io/session>



HashiCorp
Terraform Cloud

Sign in to Terraform Cloud

Username or email

Password

[Forgot password?](#)

Sign in

[Sign in with SSO.](#)

Need to sign up? Create your [free account](#).

View [Terraform Offerings](#) to find out which one is right for you.

Figure 10.1 – Terraform Cloud login page

2. Create a new organization in your profile, ensuring that you have provided a unique organization, as illustrated in the following screenshot:

Create a new organization

Organizations are privately shared spaces for teams to collaborate on infrastructure. [Learn more](#) about organizations in Terraform Cloud.

Organization name

e.g. company-name

Organization names must be unique and will be part of your resource names used in various tools, for example `terraformlab234/www-prod`.

Email address

The organization email is used for any future notifications, such as billing alerts, and the organization avatar, via [gravatar.com](#).

Create organization

Figure 10.2 – Terraform Cloud organization

3. Create a new workspace within the created organization, as illustrated in the following screenshot:

terraformlab234 / Workspaces

Workspaces + New workspace

All 0 Success 0 Error 0 Needs Attention 0 Running 0 Filter Sort Search by name Q

WORKSPACE NAME	RUN STATUS	RUN	REPO	LATEST CHANGE
You don't have any workspaces yet Create one now.				

Figure 10.3 – Terraform Cloud workspace

4. When creating a workspace in Terraform Cloud, you need to select a workflow. In our case, we are going to select a **Version control workflow** workflow option—that is, the GitHub repository, as illustrated in the following screenshot:

<https://app.terraform.io/app/terraformlab234/workspaces/new>

Create a new Workspace

Workspaces determine how Terraform Cloud organizes infrastructure. A workspace contains your Terraform configuration (infrastructure as code), shared variable values, your current and historical Terraform state, and run logs. [Learn more](#) about workspaces in Terraform Cloud.

The screenshot shows the 'Choose Type' step of the workspace creation process. It features a progress bar with four steps: 1. Choose Type (active), 2. Connect to VCS, 3. Choose a repository, and 4. Configure settings. Below the progress bar, the 'Choose your workflow' section is displayed. It contains three options, each with an icon, a title, a description, a 'Learn More' link, and a right-pointing arrow:

- Version control workflow** (Most common): Store your Terraform configuration in a git repository, and trigger runs based on pull requests and merges. Learn More
- CLI-driven workflow**: Trigger remote Terraform runs from your local command line. Learn More
- API-driven workflow**: A more advanced option. Integrate Terraform into a larger pipeline using the Terraform API. Learn More

Figure 10.4 – Terraform Cloud workflow

5. Provide the workspace name, branch name, and additional working directory or subdirectory, as illustrated in the following screenshot:

<https://app.terraform.io/app/terraformlab234/workspaces/new>

Create a new Workspace

Workspaces determine how Terraform Cloud organizes infrastructure. A workspace contains your Terraform configuration (infrastructure as code), shared variable values, your current and historical Terraform state, and run logs. [Learn more](#) about workspaces in Terraform Cloud.

Choose Type
 Connect to VCS
 Choose a repository
 4 Configure settings

Configure settings

Workspace Name

infrastructure-automation

The name of your workspace is unique and used in tools, routing, and UI. Dashes, underscores, and alphanumeric characters are permitted. [Learn more about naming workspaces](#).

[^ Advanced options](#)

Terraform Working Directory

chapter10/terraform-cloud


The directory that Terraform will execute within. This defaults to the root of your repository and is typically set to a subdirectory matching the environment when multiple environments exist within the same repository.

Terraform will change into the `chapter10/terraform-cloud` directory prior to executing any operation. Any modules utilized can be referenced outside of this directory.

Automatic Run Triggering

Choose when runs should be triggered by VCS changes.

Always trigger runs
 Only trigger runs when files in specified paths change

chapter10/terraform-cloud Working Directory 

e.g. ./modules Add path

VCS branch

master

The branch from which to import new versions. This defaults to the value your version control provides as the default branch for this repository.

Include submodules on clone

Checking this box will perform a recursive clone of your repositories submodules, making them available in the resulting slug containing your Terraform configuration. Recursive clone is performed with `--depth 1`.

Create workspace Cancel

Figure 10.5 – Terraform Cloud VCS

6. If you have any authentication access key variables or input variables in the configuration files, you can configure these, as illustrated in the following screenshot:

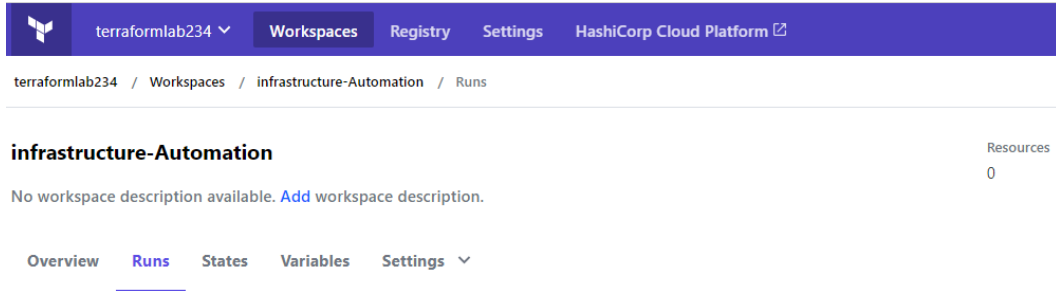


Figure 10.6 – Terraform Cloud Configure variables button

7. Provide all the keys and respective values of the variables you declared in the Terraform configuration file, as illustrated in the following screenshot:

Variables

These variables are used for all plans and applies in this workspace. Workspaces using Terraform 0.10.0 or later can also load default values from any `*.auto.tfva`

Sensitive variables are hidden from view in the UI and API, and can't be edited. (To change a sensitive variable, delete and replace it.) Sensitive variables can still appear in output them.

When setting many variables at once, the [Terraform Cloud Provider](#) or the [variables API](#) can often save time.

Terraform Variables

These [Terraform variables](#) are set using a `terraform.tfvars` file. To use interpolation or set a non-string value for a variable, click its HCL checkbox.

Key	Value
rgname Provide resource group name	terraform-lab-rg
rglocation provide name of the location	eastus

[+ Add variable](#)

Environment Variables

These variables are set in Terraform's shell environment using `export`.

Key	Value
ARM_CLIENT_ID	607907bc-e557-419e-81e1-a3e10e41644a
ARM_CLIENT_SECRET <small>SENSITIVE</small>	<i>Sensitive - write only</i>
ARM_SUBSCRIPTION_ID	97c3799f-2753-40b7-a4bd-157ab464d8fe
ARM_TENANT_ID	abfa0b4e-cdda-49d0-8fc5-24f185a5c497

[+ Add variable](#)

Figure 10.7 – Terraform Cloud variables values

8. For a better understanding of this, we have written a configuration file for creating an Azure resource group and placed it in our GitHub repository. The following files are present under the `terraform-cloud` directory of the `chapter10` folder:

```
inmishrar@terraform-vm: ~/HashiCorp-Infrastructure-
Automation-Certification-Guide/chapter10# tree
.
└── terraform-cloud
    ├── main.tf
    ├── providers.tf
    └── variables.tf
```

9. As we have already defined an `auto-trigger` in the Terraform Cloud console, it will go ahead and run `terraform plan` in the backend for us and will ask for confirmation for `terraform apply`, as illustrated in the following screenshot:

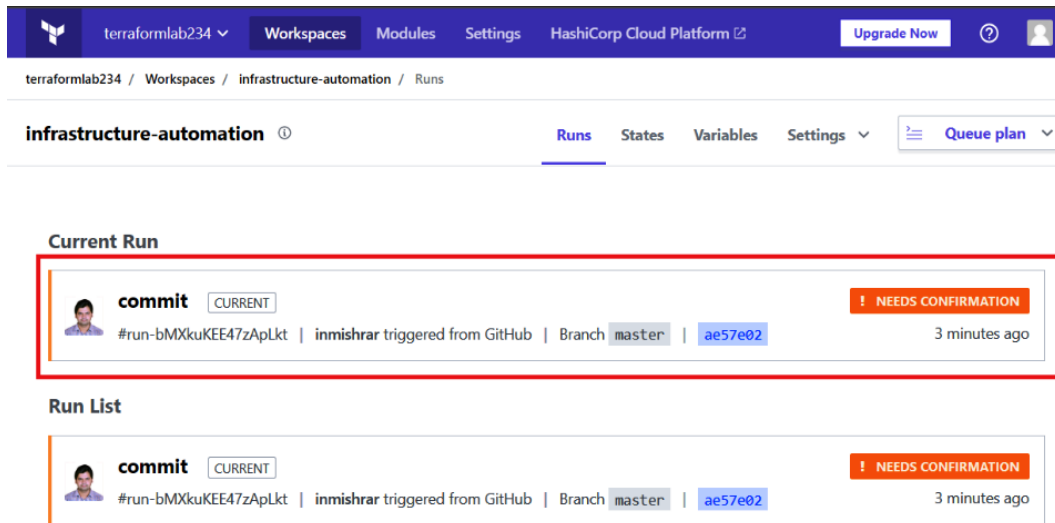


Figure 10.8 – Terraform Cloud plan phase

10. After looking at the `terraform plan` output on the terminal of the Terraform Cloud console, if you want to, you can choose **Discard Run** or **Confirm & Apply**, as illustrated in the following screenshot:

The screenshot displays the Terraform Cloud interface for a workspace named 'infrastructure-automation'. The current run is titled 'commit' and is in a 'CURRENT' state. A notification indicates that 'inmishrar' triggered a run from GitHub 13 minutes ago. The plan phase is 'Plan finished' 13 minutes ago, with resources: 1 to add, 0 to change, 0 to destroy. The plan output shows the configuration of an Azure Resource Group:

```

Terraform v1.0.1
on linux_amd64
Configuring remote state backend...
Initializing Terraform configuration...

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# azurerm_resource_group.terraform-rg will be created
+ resource "azurerm_resource_group" "terraform-rg" {
+   id           = (known after apply)
+   location    = "eastus"
+   name        = "terraform-lab-rg"
}

Plan: 1 to add, 0 to change, 0 to destroy.
    
```

Below the plan output, the 'Apply pending' section is highlighted with a red box. It contains the following text and buttons:

Needs Confirmation: Check the plan and confirm to apply it, or discard the run.

Buttons: **Confirm & Apply**, **Discard Run**, **Add Comment**

Figure 10.9 – Terraform Cloud plan output

11. Finally, we can see in the following screenshot how easily we managed to create an Azure resource group:

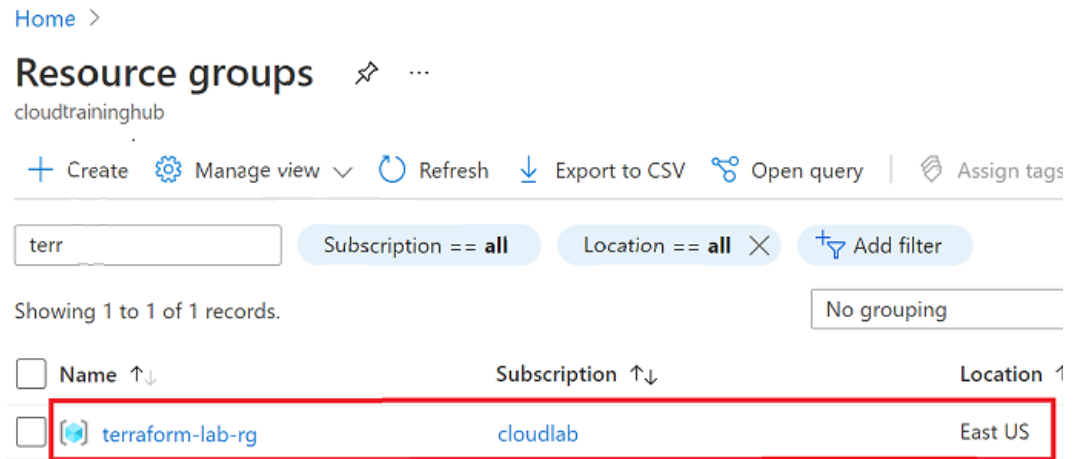


Figure 10.10 – Terraform Cloud apply output

12. In rare cases, if you need to destroy the infrastructure you provisioned using Terraform Cloud, then under the **Settings** option you can go and select the **Destruction and Deletion** option. From this page, you can delete both the infrastructure and the Terraform Cloud workspace, as illustrated in the following screenshot:

terraformlab234 / Workspaces / infrastructure-automation / Settings / Destruction and Deletion

infrastructure-automation ⓘ Runs States Variables Settings ▾ Queue plan ▾

Destruction and Deletion

There are two independent steps for destroying this workspace and any infrastructure associated with it. First, any Terraform infrastructure should be destroyed. Second, the workspace in Terraform Cloud, including any variables, settings, and alert history can be deleted.

Destroy infrastructure

Allow destroy plans
When enabled, this setting allows a destroy plan to be created and applied. This also applies when using the CLI.

Manually destroy

Queuing a destroy plan will redirect to a new plan that will destroy all of the infrastructure managed by Terraform. It is equivalent to running `terraform plan -destroy -out=destroy.tfplan` followed by `terraform apply destroy.tfplan` locally.

[Queue destroy plan](#)

Delete Workspace

Deleting a workspace will remove any variables, settings, alert history, run history, and state related to it. This **will not** remove any infrastructure managed by this workspace. If needed, destroy the infrastructure prior to deleting the workspace.

[Delete from Terraform Cloud](#)

Figure 10.11 – Terraform Cloud Destroy infrastructure option

Thus, we have learned how we can use the Terraform Cloud console for resource deployment.

Important note

While defining variables in Terraform Cloud, you have an option to set a variable value as sensitive. If you check a value as sensitive, that value will then be seen in encrypted format. This is good for storing secret values that are confidential. In *Step 7*, we have stored the `ARM_CLIENT_SECRET` value as sensitive.

From this entire section, you have gained an understanding of what Terraform Cloud is and how it would be effective from a business point of view when working with a small number of people. Moving on further, we are going to discuss the Terraform Enterprise product.

Understanding Terraform Enterprise

Terraform Enterprise is a HashiCorp product that is a self-hosted variant of Terraform Cloud. It is a private instance/server that has a Terraform Cloud application, without having any resource limit and with some additional features such as audit logging and **Security Assertion Markup Language (SAML) single sign-on (SSO)**. For a detailed understanding about different supported features of Terraform Enterprise, you can refer to <https://www.hashicorp.com/products/terraform/pricing>. This product is very good for large enterprises with higher numbers of customers. Large-enterprise customers can procure a Terraform Enterprise license from HashiCorp. The deployment of the Terraform Enterprise installer totally depends on the customer and the type of reference architecture they want to refer to Terraform Enterprise. Based on customers' experiences, Terraform has published some reference architecture documentation that can be found at <https://www.terraform.io/docs/enterprise/before-installing/reference-architecture/index.html>.

Terraform Enterprise can be installed on **Red Hat Enterprise Linux (RHEL)** and **CentOS**. For RHEL-specific requirements for Terraform Enterprise, you can visit <https://www.terraform.io/docs/enterprise/before-installing/rhel-requirements.html>, and for CentOS-specific requirements, go to <https://www.terraform.io/docs/enterprise/before-installing/centos-requirements.html>.

Most of the time, many organizations start with Terraform open source software to build and manage their infrastructure into clouds, using IaC. Over time, challenges mount up when your team size increases. You need to start thinking about collaboration, policy, security, and governance. Several questions come to mind, such as the following:

- When you have many members in a team, how will provisioning of workflows get impacted?
- How will you establish collaboration in a team?
- How will the team create and manage infrastructure state?
- How can developers enable their self-service infrastructure?
- How will you enforce provisioning rules and best practices within Terraform?

- How do administrators control **role-based access control (RBAC)** on each of the infrastructure teams (for example: network, database, monitoring)?
- How can Terraform help you to enable a secure multi-cloud environment?

Most of these questions will get answered when we start using HashiCorp's Terraform Enterprise product.

Here are a few of the key benefits we can expect when using Terraform Enterprise:

- Operational efficiency
- Risk reduction
- Collaboration capabilities
- Controlling cloud costs
- Governance and policy capabilities

These key benefits will help us to go ahead and start using Terraform Enterprise.

From this section, you have gained a brief understanding of Terraform Enterprise. Discussing Terraform Enterprise in detail is somewhat beyond the scope of this book, so we have kept this as a short overview. We are now going to discuss *policy as code* with Terraform Sentinel.

Overviewing Terraform Sentinel

Terraform Sentinel is a feature that is only available in the paid version of HashiCorp products such as Vault Enterprise, Nomad Enterprise, Consul Enterprise, Terraform Cloud, or Terraform Enterprise. This is basically a well-defined framework written in a code format—that is, policy as code. Terraform Sentinel has its own standard language of writing: Sentinel language. Don't worry, as it's not difficult to learn this language: anyone can learn it in just an hour, and you don't require any sort of programming language experience for this. Terraform Sentinel helps to restrict or control the behavior of the infrastructure before it actually gets deployed. Sentinel checks for defined governance requirements, and this whole flow can be controlled and automated by placing them in the VCS, as illustrated in the following diagram:

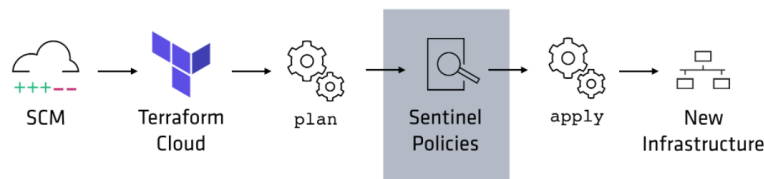


Figure 10.12 – Terraform Sentinel phase

The Sentinel CLI will run and validate the defined Sentinel policies. Sentinel policies are enforced by Terraform in between the `plan` and `apply` phases so that Terraform will prevent the deployment of the infrastructure until it is overridden by an authorized user or `terraform plan` passes through all the defined requirements checked in the Sentinel policies; this totally depends on what sort of enforcement level has been defined in the policy code. For more understanding about enforcement levels, you can refer to <https://www.terraform.io/docs/cloud/sentinel/manage-policies.html#enforcement-levels>.

If you want to have a Sentinel CLI installed on your system, then you can download this from <https://docs.hashicorp.com/sentinel/downloads>.

Once you have downloaded Sentinel, you can validate it from the CLI by simply typing `sentinel`, as illustrated in the following code snippet:

```
C:\>sentinel
Usage: sentinel [--version] [--help] <command> [<args>]
Available commands are:
    apply      Execute a policy and output the result
    fmt        Format Sentinel policy to a canonical format
    test       Test policies
    version    Prints the Sentinel runtime version
C:\>
```

For an explanation of Sentinel policies, we have taken an example to validate whether an *Azure resource group has tags or not*. This policy will restrict the provisioning of a resource group in Azure. There are many ways of placing Sentinel policies and testing them, but we are discussing how to place Sentinel policies through a VCS and using Terraform Cloud, as follows:

1. Create a policy set in Terraform Cloud and define the directory and subdirectory from where it should fetch Sentinel policies, as illustrated in the following screenshot:

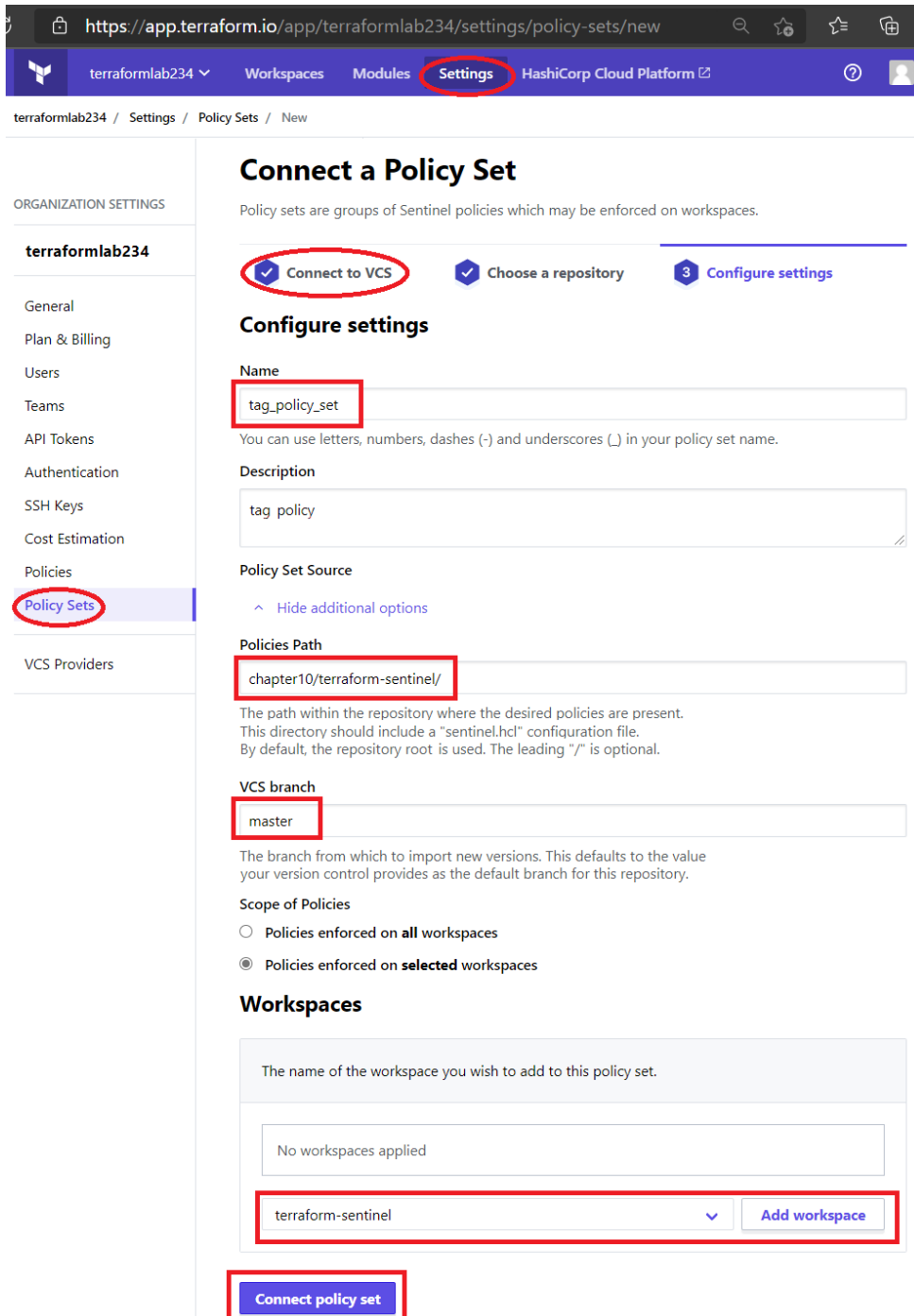


Figure 10.13 – Terraform Sentinel policy set

2. The Sentinel policy set requires a configuration filename of `sentinel.hcl`; this can be found inside the `terraform-sentinel` directory of the `chapter10` folder. The following code is present in the `sentinel.hcl` file:

```
policy "azure_tags" {  
  source          = "./azure_tags.sentinel"  
  enforcement_level = "hard-mandatory"  
}
```

We defined the `source` and gave the local path and enforcement level as `hard-mandatory` so that policies must pass in order for the run to continue to the `Confirm & Apply` state and no-one can bypass this enforcement. For more details about `sentinel.hcl`, you can visit <https://www.terraform.io/docs/cloud/sentinel/manage-policies.html>.

3. To test whether our Sentinel policy failed or passed, we firstly placed the following code block in the `main.tf` file present inside the `terraform-sentinel` directory, where we had defined a `tags` argument block:

```
resource "azurearm_resource_group" "terraform-rg" {  
  name      = var.rgname  
  location = var.rglocation  
  tags = {  
    "environment" = "test"  
    "costcenter"  = "terraform-sentinel"  
  }  
}
```

We can see in the following screenshot that the Sentinel policy was able to test the *presence of tags* and provided us with a valid output:

The screenshot displays the Terraform Sentinel web interface. At the top, the browser address bar shows the URL: `https://app.terraform.io/app/terraformlab234/workspaces/terraform-sentinel/`. Below the navigation bar, the main content area shows a list of runs. The most recent run, titled "updated policy" (status: APPLIED), is highlighted. It shows a sequence of events: "Plan finished", "Cost estimation finished", and "Policy check passed". The "Policy check passed" event is expanded, showing a detailed log. The log text, highlighted with a red border, reads: "This result means that all Sentinel policies passed and the protected behavior is allowed. 1 policies evaluated. ## Policy 1: tags_policy_set/azure_tags (hard-mandatory) Result: true ./azure_tags.sentinel:14:1 - Rule 'main' Value: true ./azure_tags.sentinel:8:1 - Rule 'azure_tags' Value: true". Below the log, there are buttons for "View raw log", "Top", "Bottom", "Expand", and "Full screen". At the bottom of the run details, there is a "Comment" section with the text "Comment: Leave feedback or record a decision." and an "Add Comment" button.

Figure 10.14 – Terraform Sentinel policy pass result

4. Now, let's see whether we can stop the provisioning of the Azure resource group if *tags are not present*. For this, we made changes in the `main.tf` file and placed the following code block within it:

```
resource "azurerm_resource_group" "terraform-rg" {  
  name      = var.rgname  
  location = var.rglocation  
  # tags = {  
  #   "environment" = "test"  
  #   "costcenter"  = "terraform-sentinel"  
  # }  
}
```

5. We commented the `tags` argument and its values in the `main.tf` file and then pushed the latest code to our GitHub repository. After that, we queued the `plan` workflow in Terraform Cloud and were able to see a **Policy check hard failed** message, as illustrated in the following screenshot:

✖ ERRORED commented tags in main.tf file CURRENT

inmishrar triggered a run from GitHub a few seconds ago Run Details ▾

Plan finished a few seconds ago Resources: 1 to add, 0 to change, 0 to destroy ▾

Cost estimation finished a few seconds ago Resources: 0 of 0 estimated · \$0.00/mo · +\$0.00 ▾

Policy check hard failed a few seconds ago Policies: 0 passed, 1 hard failed ▲

Queued a few seconds ago > Hard failed a few seconds ago

failed tags_policy_set/azure_tags

[View raw log](#)
[↑ Top](#)
[↓ Bottom](#)
[⌵ Expand](#)
[↗ Full screen](#)

```

violation(s), which is usually indicated by a rule with a false boolean value,
or non-zero collection data.

1 policies evaluated.

## Policy 1: tags_policy_set/azure_tags (hard-mandatory)
Result: false

./azure_tags.sentinel:14:1 - Rule "main"
Value:
  false

./azure_tags.sentinel:8:1 - Rule "azure_tags"
Value:
  false
    
```

— Apply will not run

Figure 10.15 – Terraform Sentinel policy fail result

You may have noticed that our created Sentinel policy was able to validate and stopped the flow immediately after the `terraform plan` phase, and this will therefore not let Terraform run the `terraform apply` phase.

In this section, we learned about Terraform Sentinel policies and saw how easily we can write a policy that can restrict resources before they get provisioned. For a detailed understanding of Terraform Sentinel, you can visit <https://www.hashicorp.com/resources/writing-and-testing-sentinel-policies-for-terraform>. We are now going to discuss feature differences between Terraform Cloud, Terraform Enterprise, and Terraform CLI.

Comparing different Terraform features

We have already discussed many things about Terraform CLI throughout this entire book, and in this chapter, we started discussing Terraform Cloud and Terraform Enterprise and demonstrated a few examples so that you can get an idea of how they are different in their features. The following screenshot gives you a brief idea about feature differences in different Terraform products:

Features	Terraform CLI	Terraform Cloud			Terraform Enterprise
		FREE	TEAM and GOVERNANCE	BUSINESS	
IaC	YES	YES	YES	YES	YES
Remote State		YES	YES	YES	YES
VCS Connection		YES	YES	YES	YES
Workspace Management		YES	YES	YES	YES
Secure Variable Storage		YES	YES	YES	YES
Remote Runs		YES	YES	YES	YES
Private Module Registry		YES	YES	YES	YES
Team Management			YES	YES	YES
Sentinel Policy as Code Management			YES	YES	YES
Cost Estimation			YES	YES	YES
SSO				YES	YES
Audit Logging				YES	YES

Features	Terraform CLI	Terraform Cloud			Terraform Enterprise
		FREE	TEAM and GOVERNANCE	BUSINESS	
Self-Hosted Agents				YES	
Configuration Designer				YES	YES
ServiceNow Integration				YES	YES
Concurrent Runs		Up to 1	Up to 2	Unlimited	Unlimited
Operations	Local CLI	Cloud	Cloud	Cloud	Private

For details about feature and pricing differences, you can visit the Terraform documentation at <https://www.datocms-assets.com/2885/1602500234-terraform-full-feature-pricing-tablev2-1.pdf>.

Thus, from this section, we have learned about feature differences between different Terraform products, and we also learned in which scenarios these Terraform products can be used.

Summary

In this chapter, we discussed different HashiCorp products under the umbrella of Terraform, such as Terraform Cloud and Terraform Enterprise. We also discussed how Terraform Cloud or Terraform Enterprise can help you to collaborate between teams. We also understood the advantages we can expect when using Terraform Cloud and Terraform Enterprise. We further discussed Terraform Sentinel (that is, policy as code), which can be introduced in between `terraform plan` and `terraform apply` phases so that Terraform performs a precheck, and if the configuration passes through all the necessary checks or the policy gets overridden by authorized users, only then will the `terraform apply` phase get executed. This whole chapter should help you in selecting the right Terraform products, depending upon the use case, and shows how you can implement a policy check before an actual infrastructure gets provisioned.

In our next chapter, we will discuss some Terraform acronyms in brief.

Questions

The answers to the following questions can be found in the *Assessments* section at the end of this book:

1. Terraform Sentinel needs to be placed where?
 - A. After the `apply` phase
 - B. After the `destroy` phase
 - C. In between the `plan` and `apply` phases
 - D. After the `init` phase
2. Which of the following features are available in both Terraform Cloud and Terraform Enterprise?
 - A. Terraform **JavaScript Object Notation (JSON)** files
 - B. Terraform Sentinel
 - C. `terraform fmt`
 - D. `terraform plan`
3. What does Terraform Cloud use to keep the state file?
 - A. Local directory
 - B. Remote backend in Terraform Cloud itself
 - C. Remote backend in GitHub
 - D. Local backend
4. There is a project team and they are using Terraform to provision their infrastructure, and now they want to provision a VM in Azure in the `West Europe` location, which is not allowed as per your company standards. You need to ensure that they are not able to create a VM in that specified location. How can you stop them?
 - A. Create a Sentinel policy.
 - B. Ask them to run `terraform plan` and share output from the terminal.
 - C. Ask them to send a Terraform state file.
 - D. Just ignore them and let them do what they want to.

Further reading

You can check out the following links for more information about the topics that were covered in this chapter:

- Terraform Cloud and Terraform Enterprise: <https://www.terraform.io/docs/cloud/index.html>
- Terraform Sentinel policies: <https://github.com/hashicorp/terraform-guides/tree/master/governance>
- Terraform Enterprise: <https://www.terraform.io/docs/enterprise/index.html>
- Terraform Enterprise key benefits: <https://www.hashicorp.com/resources/why-consider-terraform-enterprise-over-open-source>
- Terraform Cloud overview: <https://www.terraform.io/docs/cloud/overview.html>

Terraform Glossary

In this glossary, we are going to list all the acronyms and key terms used throughout this book. This will help to clarify all the technical terms. These terms should assist in helping you to get to grips with the different conversations currently taking place in the Terraform community:

- **API:** An application programming interface. This is an interface that has been designed to manipulate some functionality of an application. Terraform uses cloud or on-premises API software to manage infrastructure. Terraform resources are responsible for mapping to each API call that is required to create, update, or delete the infrastructure.

Terraform Cloud also has APIs that can be used to easily manage policies and workspaces.

- **Apply:** This is one of the phases in the Terraform life cycle that helps Terraform to perform implementation by accepting planned changes, or to make real changes to the Terraform providers using their resource APIs.
- **Azure:** This is a public cloud provided by Microsoft.
- **AzureRM:** This is one of the Terraform providers specific to Azure.
- **AWS:** This is a public cloud provided by Amazon.

- **Arguments:** When we write a Terraform configuration file, arguments take the form `<IDENTIFIER> = <EXPRESSION>` and are defined by a block of resources and data sources. Generally, they hold the properties of the Terraform providers.
- **Attribute:** An attribute is a property of an object that can be exported. For example, we can get an attribute value in a defined way: `aws_instance.eg.id`.
- **Backend:** A backend is required to store the Terraform state files, so it could be remote or local. The choice of backend determines how the state is stored and where Terraform will execute.
- **Block:** The HashiCorp configuration syntax used to create a container that holds all the defined objects like a resource. A block contains one or more labels and a body containing name and value pairs:

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {  
  # Block body  
  <IDENTIFIER> = <EXPRESSION> # Argument
```

- **CI/CD:** Continuous Integration/Continuous Delivery. CI/CD involves operating principles and practices that enable a reliable sequence in terms of the development and deployment of cloud resources, typically an application. This helps as regards complete automation. In our course, we used Azure DevOps Pipeline to explain CI/CD concepts using Infrastructure as Code using Terraform.
- **CLI:** The command-line interface. We can run Terraform commands on any terminal, such as a Unix shell or the Windows CLI. The Terraform CLI is an open source binary. In other words, `terraform.exe` can be downloaded and placed in local system environment variables and Terraform commands can start to be used.
- **Clone:** Cloning means copying files to the local system so that it will be easy for us to make changes and push them back to the repository. It is basically a `git clone` command that helps to perform cloning from the source control repository to the local system.
- **Configuration language:** Terraform files used to have code written in declarative format describing the desired state of the infrastructure. It is mainly referred to as **HashiCorp Configuration Language (HCL)**.
- **Data source:** Like a Terraform resource code block, we have a data source that generally helps you to fetch information regarding the already existing infrastructure that has been deployed by some other means, for example, manually, or using any other Terraform configuration code. In places, you may also find it referred to as a data resource.

-
- **Fork:** This is like a clone where one copy of the repository gets cloned into the VCS server. It might be the case that you are copying content and history from one repository to another. For example, suppose you need to work on the Microsoft repository. What you can do is that you can clone it to your local VCS so that it will be easy for you to work on it. You would be getting the full option of copying content to your local VCS with all its branches. The main goal of a fork is to copy a source repository to another repository.
 - **Git:** A distributed version control system that will hold all the change history that has been performed on any file or folder defined in the source code. This is beneficial when you are working with many developers, as they would easily be able to coordinate with one another.
 - **HCL:** HashiCorp Configuration Language. This is a structured configuration syntax that is used to write Terraform code. It uses specific block types, such as a resource, variable, provider, and built-in function. It is written as a named value pair.
 - **IaC:** Infrastructure as Code is a way of writing infrastructure in code format and keeping it in a file so that it will be easy to manage infrastructure, for example, Terraform configuration files.
 - **Input variables:** This is a method of declaring a variable in Terraform and taking a value from the user.
 - **Locking:** A Terraform state file is used to go into a locked state so that it can avoid another Terraform process, such as `apply`, when a Terraform operation is already running.
 - **Log:** This is a text-based output that gets printed to `stderr` following the execution of any Terraform operations, such as `plan` and `apply`.
 - **Module:** A Terraform module is more like a code function. You have input and a module can provide output, but all processes are internal to the module. It is like a self-defined container where it has all the Terraform configuration code that can be used to manage the infrastructure. Other Terraform configurations can call the module that tells Terraform to manage a resource that is defined in the module configuration code.
 - **Output values:** Suppose we want to see what Terraform operation had been performed, irrespective of whether a defined resource was created. We can see them during the runtime itself by defining them in the output file as an output value.
 - **Plan:** This is one of the Terraform workflow operations where Terraform compares the infrastructure's real state to the configuration, and is shown to the user in a readable format, about the changes it is going to perform to match the desired state.

- **Policy:** This is basically a rule that is enforced by Terraform to validate the plan, irrespective of whether the resources comply with company policy.
- **Policy set:** A list of policies defined to be enforced globally or in relation to a specific workspace.
- **Provider:** This is a plugin for Terraform that holds collections of the available resources. There are many Terraform providers, including AWS, Azure, and GCP. Terraform has already published a list of available providers that can be found at <https://www.terraform.io/docs/providers/index.html>.
- **Remote backend:** Terraform state files can be stored in defined storage, such as S3 and Google Cloud Storage. If we want to store the state file securely, then it's desirable to configure the remote backend.
- **Repository:** This is a place where code files can be kept and managed. Primarily, this will be any version control system that can hold the entire history of changes to the file. A repository is mainly a Git repository that will retain all Terraform configuration files except secrets.
- **Resource:** In the Terraform configuration file, we defined a resource code block that describes one or more infrastructure objects. A resource block instructs Terraform to manage the defined resource. Terraform uses cloud provider APIs to create, edit, and destroy resources.
- **Sentinel:** Sentinel is a feature available in all paid HashiCorp products, including Terraform Cloud, Terraform Enterprise, HashiCorp Enterprise Vault, and HashiCorp Enterprise Consul, where we can define policy as code.
- **State:** Terraform generates a state file in a JSON format where it holds all the information about the defined infrastructure and maps those resources to the real world. Without state, Terraform can't know which resources got provisioned during the previous run. So, it is very important to know what action has been performed during the previous run if you are working with many people. That is why Terraform generates state files and records all the defined infrastructure resources in the state file.
- **VCS:** A version control system, such as Git. This is a software application that tracks changes to collections of files and helps you to monitor changes, undo changes, or combine them.
- **Workspace:** In the Terraform CLI, a workspace is an isolated instance that can be used to deploy multiple environments using a single Terraform configuration file.

Assessments

Chapter 1

1. Correct answer: B

Explanation: As covered in the *What is Terraform?* section, Terraform is an orchestration tool used for infrastructure provisioning and its efficient maintenance.

2. Correct answers: A and B

Explanation: As covered in the *Terraform architecture* section, where we discussed what a Terraform plugin is, it contains both Terraform providers and provisioners.

3. Correct answer: B

Explanation: As covered in the *A comparison with other IaC* section, we saw how Terraform differs from other IaC, such as an ARM template and CloudFormation, and we learned that Terraform configuration files are written in HCL.

4. Correct answer: D

Explanation: Given that we already know that *Azure*, *AWS*, and *GCP* are providers, by following this logic, *SAP* is the only remaining option that is not a Terraform provider. In the future, you may well expect Terraform to have *SAP* providers as well, but this isn't the case at present.

5. Correct answer: B

Explanation: In the *Introduction to Terraform* section, we discussed what Terraform is, and you got to know which company introduced Terraform to the market – none other than HashiCorp.

Chapter 2

1. Correct answer: B

Explanation: You can check the version of Terraform using any of these commands: `terraform version`, `terraform -v`, `terraform -version`, or `terraform --version`.

2. Correct answers: A and B

Explanation: There are multiple ways of opening help options in Terraform, such as `terraform -h`, `terraform -help`, `terraform -help`, `terraform help`, and so on.

3. Correct answer: B

Explanation: There is an executable file of Terraform for every operating system, such as Windows, Linux, macOS, and so on, which is available on the Terraform website at <https://www.terraform.io/downloads.html>. So, you can directly download it based on your operating system and architecture. You don't need the Go language library for it.

Chapter 3

1. Correct answer: A

Explanation: Terraform configuration can be written in JSON. For more information, you can visit <https://www.terraform.io/docs/configuration/syntax-json.html>.

2. Correct answer: D

Explanation: An `alias` meta-argument is used when you are using the same Terraform provider with different configurations for different resources. For more information about multiple providers, please refer to <https://www.terraform.io/docs/configuration/providers.html#alias-multiple-provider-instances>.

3. Correct answer: B

Explanation: A resource code block is written by referring to the actual service name and then the local name. You can read more about this at <https://www.terraform.io/docs/configuration/resources.html>.

4. Correct answer: C

Explanation: Data source code helps you to learn about your existing infrastructure and using that data source, you can fetch certain output that you can consume in your other configuration code.

5. Correct answer: B

Explanation: By defining input variables in the Terraform configuration code, you can make configuration code reusable and dynamic.

Chapter 4

1. Correct answer: A

Explanation: The Terraform support environment variable, `TF_LOG`, should be enabled with some trace label so that Terraform can start collecting logs and storing them in the defined path, `TF_LOG_PATH`. For more information, you can read the Terraform debugging page at <https://www.terraform.io/docs/internals/debugging.html>.

2. Correct answer: C

Explanation: Terraform has inbuilt functions and `max` is one of them, so the maximum value among the given number is 10. For more information about Terraform functions, you can refer to <https://www.terraform.io/docs/configuration/functions.html>.

3. Correct answer: D

Explanation: GitHub is not a supported backend type. For more information about Terraform backends, you can read <https://www.terraform.io/docs/backends/types/index.html>.

4. Correct answer: C

Explanation: Terraform provisioners can be used to run some sort of script, whether it is on a local or remote machine, but always remember that a provisioner should be used only when you don't have any other option.

5. Correct answer: D

Explanation: A dynamic expression of Terraform can help you to iterate over the inline code inside a resource code block. You can read about different Terraform iterations at <https://www.hashicorp.com/blog/hashicorp-terraform-0-12-preview-for-and-for-each>, <https://www.terraform.io/docs/language/expressions/index.html> and <https://learn.hashicorp.com/tutorials/terraform/count?in=terraform/configuration-language>.

Chapter 5

1. Correct answer: B

Explanation: The `force-unlock` command is used for unlocking the state file. For more information, you can read <https://www.terraform.io/docs/commands/force-unlock.html>.

2. Correct answer: C

Explanation: The `terraform taint` command marks a Terraform managed resource as a taint and when you run `apply` next time, it will destroy and recreate that specific resource. For more information, head to <https://www.terraform.io/docs/commands/taint.html>.

3. Correct answer: B

Explanation: `terraform fmt` rewrites the configuration files to a canonical format and style. For information, you can read <https://www.terraform.io/docs/commands/fmt.html>.

4. Correct answer: A

Explanation: Using `terraform import`, you can bring any resources that already exist to the state file, and once you have that resource in the state file, the life cycle of that resource can be managed by Terraform.

5. Correct answer: A

Explanation: The `terraform validate` command will help you to check any kind of syntax error in the configuration file.

Chapter 6

1. Correct answers: A, B, and D

Explanation: The `terraform init` command is used to perform initialization and download the respective modules, providers, and plugins. It also initializes the backend config. You can read about `terraform init` at <https://www.terraform.io/docs/commands/init.html>.

2. Correct answer: C

Explanation: The `terraform plan` lets you know what resources it is going to update or create to achieve the desired state defined in the configuration file. For more information, you can read <https://www.terraform.io/docs/commands/plan.html>.

3. Correct answers: A, B, and D

Explanation: Terraform can take an input variable value in multiple ways.

4. Correct answer: B

Explanation: The `terraform validate` command will help you check if there is any sort of syntax error.

5. Correct answer: C

Explanation: The `terraform destroy` command will help you to delete infrastructure that has been provisioned using Terraform. For more details, you can read <https://www.terraform.io/docs/commands/destroy.html>.

Chapter 7

1. Correct answer: C

Explanation: The Terraform module supports a local, private, or public registry. The source URL should be defined correctly. For more information, you can read <https://www.terraform.io/docs/modules/sources.html>.

2. Correct answer: A

Explanation: When we run `terraform init`, it downloads files to the local directory and stores them inside the `.terraform` folder.

3. Correct answer: A

Explanation: When we want to get output from the module, then we need to define `module.<modulename>.<outputname>`.

4. Correct answer: C

Explanation: The Terraform module is the best option if you are working with many developers – it will be easy to write a module and use it again and again.

5. Correct answer: B

Explanation: The Terraform module has a local source path that doesn't support any versioning; it always takes the latest version.

Chapter 8

1. Correct answer: B

Explanation: We already described the Terraform style convention that each nesting level in the Terraform configuration file should have two indentation spaces. To learn more about it, you can visit <https://www.terraform.io/docs/language/syntax/style.html>.

2. Correct answer: A

Explanation: The default way of placing a comment in the Terraform configuration file is using `#`.

3. Correct answer: A

Explanation: The `terraform validate` command will help you to check any syntax error in the Terraform configuration file.

4. Correct answer: A

Explanation: If we want to see an output of the provisioned resources, then we should define them as an output so that we can see it at runtime itself. Option B is also quite close – no doubt we can see the name of the provisioned storage account by looking inside the state file but it is not the recommended way of doing it. And even option C is correct – if we run the `terraform show` command, it will display everything that is there inside the `state` file, but as the question is expecting to have a specific output, that is, only the `storage account` name, that also during the runtime itself, so option C can't be considered as a correct answer.

5. Correct answer: C

Explanation: By default, Terraform will look for `terraform.tfvars` or any filename ending with `.tfvars` or `.auto.tfvars` for its input variable value. There's no doubt that Terraform can read input variable values from any other files, but for that, you would be required to define the filename with `terraform apply -var-file=<filename>`.

Chapter 9

1. Correct answer: B

Explanation: We already know that modules are written when we want to perform some repetition of infrastructure provisioning or updating using modules and modules can be combined together to form a stack.

2. Correct answer: C

Explanation: The `terraform fmt` command helps you to arrange all the Terraform IaC code into the canonical format and `terraform fmt -recursive` will help you to perform the formatting of all the Terraform configuration files down the line in all the directories from where you execute that command.

3. Correct answer: A

Explanation: The `terraform init` command helps you to download Terraform modules to the `.terraform` directory inside the current working directory.

4. Correct answer: A

Explanation: Input variable values can be given by defining a default value in the variable declaration file itself, by providing those values from any file ending with `.tfvars`, `.auto.tfvars`, or `.tfvars.json`, by providing values during the runtime, or you can set variable values in the system environment variables/path.

5. Correct answer: B

Explanation: We define versioning in the module code so that our infrastructure continues using that version and if we forget, then it will always download the latest version if we run `terraform init`. So to restrict the module to a specific version, we need to provide version constraints.

Chapter 10

1. Correct answer: C

Explanation: Terraform Sentinel is a policy-as-code framework that is generally used to restrict infrastructure deployment so deployment happens only when we execute the `apply` phase. So Terraform Sentinel is placed in between `plan` and `apply`.

2. Correct answer: B

Explanation: Terraform Sentinel is a policy-as-code framework supported by both the paid version of Terraform Cloud and Terraform Enterprise.

3. Correct answer: B

Explanation: Terraform Cloud keeps the state file in the cloud itself. We can define it as a remote backend and point it to Terraform Cloud.

4. Correct answer: A

Explanation: By creating a Terraform Sentinel policy, we can restrict and force the user to follow the defined governance in the Sentinel policy.



Packt . com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

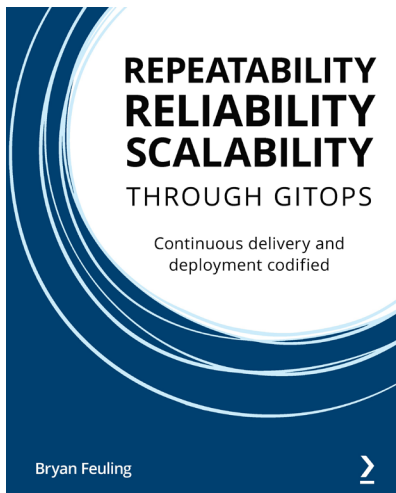
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print,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

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [packt . com](http://packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub . com](mailto:customercare@packtpub.com) for more details.

At [www . packt . com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Repeatability, Reliability, and Scalability through GitOps

Bryan Feuling

ISBN: 978-1-80107-779-8

- Explore a variety of common industry tools for GitOps
- Understand continuous deployment, continuous delivery, and why they are important
- Gain a practical understanding of using GitOps as an engineering organization
- Become well-versed with using GitOps and Kubernetes together
- Leverage Git events for automated deployments
- Implement GitOps best practices and find out how to avoid GitOps pitfalls



Google Cloud for DevOps Engineers

Sandeep Madamanchi

ISBN: 978-1-83921-801-9

- Categorize user journeys and explore different ways to measure SLIs
- Explore the four golden signals for monitoring a user-facing system
- Understand psychological safety along with other SRE cultural practices
- Create containers with build triggers and manual invocations
- Delve into Kubernetes workloads and potential deployment strategies
- Secure GKE clusters via private clusters, Binary Authorization, and shielded GKE nodes
- Get to grips with monitoring, Metrics Explorer, uptime checks, and alerting
- Discover how logs are ingested via the Cloud Logging API

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.

Share Your Thoughts

Now you've finished *HashiCorp Infrastructure Automation Certification Guide*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

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

Index

A

- access key ID, and secret
 - using, for authentication 144, 145
- Amazon Web Services (AWS)
 - about 305
 - Terraform configuration files,
 - writing for 247-250
 - Terraform modules, writing for 214-220
 - Terraform stacks, writing for 269-273
 - used, for authenticating Terraform 144
- application programming interface (API) 198, 282, 305
- apply phase 305
- arguments 306
- attribute 306
- AWS data sources 80, 81
- AWS input variables 67, 68
- AWS output 73-75
- AWS provider
 - about 57, 58
 - arguments 58
- AWS provider, authentication option
 - reference link 58
- AWS resources
 - about 62
 - reference link 62
- AWS S3 92
- AWS services
 - provisioning, with Terraform 146, 147
- Azure
 - about 305
 - Terraform CLI, authenticating to 133
 - Terraform configuration files,
 - writing for 250-254
 - Terraform modules, writing for 205-214
 - Terraform stacks, writing for 273-276
- Azure ARM templates, versus Terraform
 - about 20
 - Azure portal support 22
 - cross-platform 20
 - error message understandability 22
 - infrastructure state management 22
 - language 21
 - license and support 21
 - modularity 21
 - readability 21
 - workflow 22
- Azure data sources 79, 80

- Azure DevOps
 - using, with Terraform 180-189
- Azure input variables 64-67
- Azure output 71-73
- Azure Resource Manager 205
- Azure Resource Manager
 - (ARM) provider block
 - reference link 138
- Azure resources
 - about 60
 - blue highlighted text 62
 - features block 54
 - green highlighted text 62
 - key_vault block 54
 - log_analytics_workspace block 55
 - reference link 61
 - template_deployment block 54
 - virtual_machine block 55
 - virtual_machine_scale_set block 55
- AzureRM 305
- AzureRM provider
 - about 52-57
 - arguments 52
- Azure services
 - provisioning, with Terraform 140-143
- Azure Storage 92
- Azure subscription sign-up
 - reference link 133

B

- backend 306
- Bitbucket 198
- block 306
- blue highlighted text 62

C

- CentOS 292
- clone 306
- CloudFormation, versus Terraform
 - about 16
 - AWS console support 18
 - cross-platform 16
 - error message understandability 19
 - infrastructure state management 19
 - language 16
 - license and support 18
 - modularity 17
 - readability 18
 - validation 17, 20
 - workflow 18, 19
- command-line interface (CLI) 306
- Community Edition (CE) 281
- complex types 241
- Configuration Language 306
- content delivery network (CDN) 10
- continuous integration/continuous deployment (CI/CD) 6
- Continuous Integration/Continuous Delivery (CI/CD) 306
- continuous integration/continuous deployment (CI/CD) tool 180
- count expression 106-112

D

- data source 306
- depends_on argument 78

E

- Enterprise Edition (EE) 281

F

file provisioner 102-104
for_each expression 113-119
for expression 119, 120
fork 307
fully qualified domain name (FQDN) 71

G

GCP data sources 81
GCP input variables 68-70
GCP output 75, 76
GCP services
 provisioning, with Terraform 152, 153
Generic Git repository 198, 199
Generic Mercurial repository 199
Git 307
GitHub 198
Google App Engine 63
Google Cloud Deployment
 Manager, versus Terraform
 about 23
 cross-platform 23
 error message understandability 26
 Google console support 25
 infrastructure state management 26
 language 23
 maintainability 25
 modularity 24
 readability 24
 validation 24
 workflow 26
Google Cloud Platform (GCP)
 Terraform, authenticating to 148
 Terraform configuration files,
 writing for 244-247

 Terraform modules, writing for 220-224
 Terraform stacks, writing for 259-269
Google Cloud Storage (GCS) 92
Google Cloud Storage (GCS) buckets 201
Google provider
 about 59, 60
 arguments 59
Google service account
 authenticating 148-151
Google Terraform resource 63
Graphviz
 URL 155
green highlighted text 62

H

HashiCorp, backend classes
 enhanced 91
 standard 91
HashiCorp Configuration File (HCL) 163
HashiCorp Configuration Language
 (HCL) 16, 306, 307
Heroku app 9, 10
HTTPS Secure (HTTPS) 198
HyperText Transfer Protocol
 (HTTP) URLs 199

I

IAM accounts
 reference link 147
Identity and Access Management
 (IAM) 204
Identity and Access Management
 (IAM) roles, Azure
 reference link 138

I
Infrastructure as Code (IaC)

about 4, 5, 87, 281, 307

advantages 5-7

input variables 307

integrated development

environment (IDE) 186

J

JSON configuration syntax 239-241

L

Linux

Terraform, downloading 39-41

Terraform, installing on 39

local backend 89, 90

local-exec provisioner 99-102

locked state 88

locking 307

log 307

M

macOS

Terraform, downloading 42-44

Terraform, installing on 42

module 307

module composition 194

N

Network Security Group (NSG) 105

O

output values 307

P

plan 307

Platforms as a Service (PaaS) 9

policy 308

policy set 308

primitive types 241

provider

about 51, 308

URL 308

Q

quality analyst (QA) 6

R

Red Hat Enterprise Linux (RHEL) 292

reference architecture documentation,

Terraform Enterprise

reference link 292

remote backend

about 91-96, 308

advantages 91

remote-exec provisioner

about 104, 105

reference link 105

remote operation 280

Remote Procedure Calls (RPCs) 27

remote state 88

remote Terraform execution 280

repository 308

resources 60, 308

role-based access control (RBAC) 293

run trigger command

reference link 281

S

- Secure Shell (SSH) 198
- Security Assertion Markup Language (SAML) 292
- Sentinel 308
- Sentinel CLI
 - download link 294
- Sentinel policies
 - reference link 294
- Service Principal, and Client Secret
 - used, for authenticating Terraform CLI to Azure 134-140
- Simple Storage Service (S3) buckets 200
- single sign-on (SSO) 292
- software-as-a-service (SaaS) 197, 280
- Software-Defined Networking (SDN) 13
- software development kit (SDK) 201
- stack 279
- state 308
- Subject Matter Experts (SMEs) 6
- subnet 194

T

- Terraform
 - about 7
 - architecture 27
 - authenticating, to GCP 148
 - authenticating, with AWS 144
 - Azure DevOps, using with 180-186
 - downloading, on Linux 39-42
 - downloading, on macOS 42-44
 - downloading, on Windows 34-39
 - download link 34, 39, 42, 44
 - installing, on Linux 39
 - installing, on macOS 42
 - installing, on Windows 34
 - life cycle 162
 - used, for provisioning AWS services 146, 147
 - used, for provisioning Azure services 140-143
 - used, for provisioning GCP services 152, 153
- Terraform backend 86
- Terraform backend, types
 - about 88
 - local backend 89, 90
 - remote backend 91-96
- Terraform binary
 - download link 42
- Terraform CLI
 - about 130
 - authenticating, to Azure 133
- Terraform CLI, commands
 - about 154
 - reference link 158
 - terraform console 154
 - terraform fmt 154
 - terraform force-unlock 158
 - terraform graph 155, 156
 - terraform import 158
 - terraform output 156
 - terraform refresh 157
 - terraform show 157
 - terraform taint 157
 - terraform validate 158
 - terraform workspace 157
- Terraform Cloud
 - about 280-282
 - URL 280, 282
 - workflow 282-291
- Terraform Cloud, version
 - control integration
 - reference link 282

- Terraform community providers
 - reference link 51
- Terraform configuration files
 - about 234
 - data types 241, 242
 - JSON configuration syntax 239-241
 - native configuration syntax 234-236
 - override file 236-239
 - style conventions 242, 243
 - writing, for AWS 248-250
 - writing, for Azure 250-254
 - writing, for GCP 244-247
- Terraform configuration files, for AWS
 - key practices 250
- Terraform configuration files, for Azure
 - key practices 253, 254
- Terraform configuration files, for GCP
 - key practices 245, 246
- Terraform Core 28
- Terraform data 79
- Terraform data sources
 - about 79
 - AWS data sources 80, 81
 - Azure data sources 79, 80
 - GCP data sources 81
 - reference link 79
- Terraform debugging 123, 125
- Terraform documentation
 - reference link 301
 - URL 262
- Terraform Enterprise
 - about 292, 293
 - URL 292
- Terraform features
 - comparing 300, 301
- Terraform, features
 - about 8
 - changing automation 9
 - execution plans 8
 - infrastructure as code 8
 - resource graph 9
- Terraform functions 121-123
- Terraform graph
 - subcommands, viewing 132
- Terraform language-style conventions
 - reference link 154
- Terraform life cycle 162
- Terraform life cycle workflow
 - terraform apply 175-177
 - terraform destroy 177-180
 - terraform init 163-167
 - terraform plan 169-174
 - terraform validate 167-169
- Terraform loops
 - about 105
 - count expression 106-112
 - for_each expression 113-119
 - for expression 119, 120
- Terraform modules
 - about 194-196
 - key requirements 225
 - publishing 225-227
 - source argument 196
 - version argument 201-204
 - writing, for AWS 214-220
 - writing, for Azure 205-214
 - writing, for GCP 220-224
- Terraform modules, source argument
 - Bitbucket 198
 - GCS bucket 201
 - Generic Git repository 198, 199
 - Generic Mercurial repository 199
 - GitHub 198
 - HTTP URLs 199
 - S3 bucket 200
 - Terraform Registry 197

- Terraform native configuration
 - syntax 234-236
- Terraform output
 - about 70
 - AWS output 73-75
 - Azure output 71-73
 - GCP output 75, 76
 - optional arguments 77, 78
- terraform output command
 - reference link 156
- Terraform override file 236-239
- Terraform plugins
 - about 28
 - plugin locations 29, 30
 - selecting 30
 - upgrading 30, 31
- Terraform providers
 - about 50, 51
 - AWS provider 57, 58
 - AzureRM provider 52-57
 - Google provider 59, 60
- Terraform provisioners
 - about 96
 - types 99
 - use cases 97, 98
- terraform refresh command
 - reference link 157
- Terraform Registry
 - about 51, 197
 - reference link 51
 - URL 260
- Terraform resource code
 - block, AWS provider
 - reference link 146
- Terraform resources
 - about 60
 - AWS resources 62
 - Azure resources 60
 - Google resources 63
- Terraform Sentinel
 - overview 293-300
 - reference link 300
- Terraform stacks
 - about 258, 259
 - writing, for AWS 269-273
 - writing, for Azure 273-276
 - writing, for GCP 259-269
- Terraform state file
 - about 86, 87
 - purpose 87
- Terraform state file, benefits
 - metadata 87
 - performance 88
 - real world, mapping to 87
 - syncing 88
- terraform taint command
 - reference link 157
- Terraform, use cases
 - about 9
 - disposable environments 12
 - Heroku app setup 9, 10
 - multi-cloud deployment 14, 15
 - multi-tier applications 10
 - resource schedulers 14
 - self-service clusters 11
 - Software-Defined Networking (SDN) 13
 - software demos 11
- Terraform v1.0.0
 - download link 40
- Terraform variables
 - about 63, 64
 - AWS input variables 67, 68
 - Azure input variables 64-67
 - GCP input variables 68-70

- Terraform workflows
 - commands 130
- terraform workspace
 - reference link 157
- types, Terraform provisioners
 - file provisioner 102-104
 - local-exec provisioner 99-102
 - remote-exec provisioner 104, 105

U

- Uniform Resource Locator (URL) 196
- unzip 40

V

- Vagrant 11
- version control system (VCS) 282, 308
- virtual machines (VMs) 280
- virtual private cloud (VPC) 194
- Visual Studio Code (VS Code) 186

W

- Windows
 - Terraform, downloading 34-39
 - Terraform, installing on 34
- workspaces 281, 308

